Université Clermont Auvergne
École doctorale des sciences pour l'ingénieur

Thèse présentée par

Alexis Pereda

en vue de l'obtention du grade de

Docteur d'université

Spécialité : informatique

Application de la métaprogrammation template à la conception de bibliothèques actives de parallélisation assistée

Soutenue publiquement le 01/07/2021 devant le jury composé de :

Alexandre Guitton	*Professeur des universités*	Université Clermont Auvergne	Président
Joël Falcou	*Maître de conférences HDR*	Université Paris-Saclay	Rapporteur
Françoise Baude	*Professeur des universités*	Université Côte d'Azur	Rapportrice
Éric Innocenti	*Maître de conférences*	Université de Corse	Examinateur
Bruno Bachelet	*Maître de conférences HDR*	Université Clermont Auvergne	Directeur de thèse
David Hill	*Professeur des universités*	Université Clermont Auvergne	Directeur de thèse
Claude Mazel	*Maître de conférences*	Université Clermont Auvergne	Invité
Jian-Jin Li	*Maître de conférences*	Université Clermont Auvergne	Invitée

Remerciements

Je remercie avant tout mes directeurs de thèse, Bruno Bachelet et David Hill. Bruno pour avoir proposé ce sujet de thèse (spécialement pour la partie « métaprogrammation ») et m'avoir choisi pour travailler dessus, mais surtout pour son aide, sa disponibilité, sa patience et sa confiance indéfectible durant ces quelques années. David pour avoir rendu cette thèse possible et pour son aide indispensable quant à certaines difficultés... scientifiques et administratives. Je remercie Claude Mazel, co-encadrant qui n'aura eu de cesse de me vitupérer pour d'importantes vétilles, aussi bien pour ses relectures que pour les nombreuses discussions que nous avons eues, évidemment toujours expressément au sujet de la thèse.

Je remercie également Joël Falcou, Françoise Baude, Éric Innocenti, Alexandre Guitton ainsi que Jian-Jin Li pour avoir fait partie de mon jury de thèse, et en particulier Joël et Françoise pour leur travail de rapporteurs.

Je remercie aussi l'ensemble des personnes travaillant au LIMOS et à l'ISIMA. Loïc, notamment pour m'avoir épargné, de force quelques fois, certaines besognes en faveur de l'avancement de la thèse ; Yves-Jean qui a toujours pris le temps de discuter de mathématiques, y compris quelquefois utiles à mes travaux ; Fréd avec qui j'ai pu attendre Kaamelott ; l'équipe technique et le personnel administratif qui m'ont facilité bien des démarches.

Merci aux quelques docteurs – ou presque – que j'ai cotoyé durant cette thèse pour l'avoir en partie animée. David et Kévin pour les multiples discussions autour de problèmes variés et les *occasionnelles* parties de billard ; Théo, futur artiste qui m'a évité bien des soliloques ; et bien sûr Cyrille avec qui j'ai pu explorer différents domaines de l'informatique, des autres sciences et de tant d'autres thèmes.

ICAST2019を開催した熊本大学とそれにかかわった人たちに感謝します。特に岸田先生には滞在中お世話になり、初めての日本訪問を楽しむことができました。

Merci au théorème fondamental de l'ingénierie logicielle, à savoir que tout problème peut être résolu en ajoutant un niveau d'indirection, d'être vrai, au moins dans le cadre des développements effectués durant cette thèse.

J'étends enfin ces remerciements à mes proches, nommément Gwénaëlle qui a dû accepter un emploi du temps chaotique, et m'a malgré cela soutenu inlassablement.

Ce document a été écrit en utilisant LaTeX et en profitant de nombreux paquets sans lesquels ce travail de rédaction aurait été bien moins appréciable. Je tiens donc à remercier toute personne ayant participé au développement de ces outils, et plus généralement à l'expansion du logiciel libre.

Résumé

L'écriture de programmes parallèles, par opposition aux programmes « classiques » séquentiels et n'utilisant donc qu'un processeur, est devenue une nécessité. En effet, si jusqu'au début du millénaire la puissance de calcul des ordinateurs dépendait principalement de la fréquence du processeur, elle est maintenant liée au nombre de cœurs de calcul qui sont de plus en plus nombreux. Pourtant, à cause des difficultés introduites par la parallélisation, la plupart des programmes sont toujours écrits de manière « classique ». En particulier, il peut être compliqué de déterminer, étant donné une sous partie d'un programme, si la parallélisation est possible, c'est-à-dire si elle n'introduira pas un changement de comportement du programme. Cependant, même en sachant précisément ce qui peut être parallélisé, le faire correctement est aussi une tâche difficile. Cette thèse présente deux approches pour simplifier l'écriture de programmes parallèles.

Nous proposons une bibliothèque active – par métaprogrammation template, elle agit durant la compilation – qui acquiert des informations à propos d'une portion de programme, correspondant à une boucle, au moyen de patrons d'expression. Celles-ci sont utilisées pour analyser les instructions et identifier lesquelles peuvent être exécutées en parallèle. Cette analyse repose sur deux niveaux de connaissance : l'ensemble des variables utilisées en distinguant les accès en lecture de ceux en écriture, et, puisqu'il s'agit souvent de tableaux, des fonctions d'indice. Les variables et leur mode d'accès permettent de savoir quelles instructions sont interdépendantes tandis que les fonctions d'indice nous servent à déterminer, pour un groupe d'instructions, s'il est possible de procéder à une exécution parallèle des itérations de la boucle. L'objectif de cette bibliothèque est de proposer un cadre, à la fois pour les développeurs afin d'écrire des boucles qui seront automatiquement exécutées en parallèle si cela est possible, mais aussi à un niveau plus élevé pour intégrer de nouvelles méthodes de parallélisation ou d'autres règles à utiliser pour l'analyse.

Nous proposons également une seconde bibliothèque, active elle aussi, orientée sur la parallélisation assistée en utilisant la technique des squelettes algorithmiques comme interface pour le développeur. Celle-ci permet de représenter des algorithmes complets comme des assemblages de motifs d'exécution : séquence de tâches ; exécution répétée d'une tâche en parallèle ; ... En utilisant cette connaissance, nous pouvons mettre en place des choix dans la manière de répartir les tâches exécutées en parallèle sur les différents processeurs. Par ailleurs, nous avons choisi d'expliciter l'expression de la transmission des données entre les tâches, contrairement à ce qui est habituellement fait. Grâce à cela, la bibliothèque automatise notamment la transmission de paramètres qui ne doivent pas être partagés par des tâches parallèles. Cela nous permet en particulier de garantir la répétabilité des exécutions y compris lorsque, par exemple, les tâches utilisent des nombres pseudo-aléatoires. En tenant compte de la politique d'exécution choisie et des nombres de processeurs possibles, nous réduisons la quantité nécessaire de ces paramètres ne devant pas être partagés. Ainsi, cette seconde bibliothèque propose elle aussi un cadre de programmation à plusieurs niveaux. Celle-ci est extensible au niveau de ses politiques d'exécution ou des motifs pour la construction des squelettes algorithmiques. On peut l'utiliser pour définir une variété de squelettes algorithmiques, lesquels serviront ensuite à un développeur pour écrire des programmes dont la parallélisation sera facilitée.

Mots-clés : métaprogrammation template ; parallélisation assistée ; parallélisation automatique ; bibliothèques actives ; squelettes algorithmiques ; répétabilité.

Abstract

Hardware performance has been increasing through the addition of computing cores rather than through increasing their frequency since the early 2000s. This means that parallel programming is no longer optional should you need to make the best use of the hardware at your disposal. Yet many programs are still written sequentially: parallel programming introduces numerous difficulties. Amongst these, it is notably hard to determine whether a sequence of a program can be executed in parallel, i.e. preserving its behaviour as well as its overall result. Even knowing that it is possible to parallelise a piece of code, doing it correctly is another problem. In this thesis, we present two approaches to make writing parallel software easier.

We present an active library (using C++ template metaprogramming to operate during the compilation process) whose purpose is to analyse and parallelise loops. To do so, it builds a representation of each processed loop using expression templates through an embedded language. This allows to know which variables are used and how they are used. For the case of arrays, which are common within loops, it also acquires the index functions. The analysis of this information enables the library to identify which instructions in the loop can be run in parallel. Interdependent instructions are detected by knowing the variables and their access mode for each instruction. Given a group of interdependent instructions and the known index functions, the library decides if the instructions can be run in parallel or not. We want this library to help developers writing loops that will be automatically parallelised whenever possible and run sequentially as without the library otherwise. Another focus is to provide this to serve as a framework to integrate new methods for parallelising programs and extended analysis rules.

We introduce another active library that aims to help developers by assisting them in writing parallel software instead of fully automating it. This library uses algorithmic skeletons to let the developer describe its algorithms with both its sequential and parallel parts by assembling atomic execution patterns such as a series of tasks or a parallel execution of a repeated task. This description includes the data flow, that is how parameters and function returns are transmitted. Usually, this is automatically set by the algorithmic skeleton library, however it gives the developer greater flexibility and it makes it possible, amongst other things, for our library to automatically transmit special parameters that must not be shared between parallel tasks. One feature that this allows is to ensure repeatability from one execution to another even for stochastic algorithms. Considering the distribution of tasks on the different cores, we even reduce the number of these non-shared parameters. Once again, this library provides a framework at several levels. Low-level extensions consist of the implementation of new execution patterns to be used to build skeletons. Another low-level axis is the integration of new execution policies that decide how tasks are distributed on the available computing cores. High-level additions will be libraries using ours to offer ready-to-use algorithmic skeletons for various fields.

Keywords: template metaprogramming; assisted parallelisation; automatic parallelisation; active libraries; algorithmic skeletons; repeatability.

Avant-propos

Ce document présente les travaux effectués durant ma thèse. Les projets correspondants sont accessibles à l'adresse `https://phd.pereda.fr/dev`. Parmi ces projets se trouvent les deux bibliothèques principales :

— `https://phd.pereda.fr/dev/pfor` qui est présentée dans le chapitre 4 ;

— `https://phd.pereda.fr/dev/alsk` qui est présentée dans le chapitre 5.

De nombreux extraits de code source sont étudiés. Ceux-ci sont la plupart du temps simplifiés pour se concentrer sur les points intéressants. En général, le langage de programmation est indiqué et, en particulier pour le C et le C++, le standard à partir duquel le code est valide. Ces indications sont en dessous à droite des extraits de code.

```
// source code
```

(⩾ C++14)

Ce document a été compilé spécifiquement pour impression. La version numérique comporte des figures dynamiques. Celles-ci ont toujours un rendu par défaut qui est le seul dont il est question au sein du texte. Les éléments dynamiques peuvent apporter une aide à la compréhension ou des éléments supplémentaires mais ne sont en aucun cas indispensables à la lecture.

Table des matières

Table des figures

Liste des tableaux

Liste des algorithmes

Liste des extraits de code

Introduction

L'informatique est employée dans de nombreux domaines, que ce soit comme sujet d'étude ou comme outil d'application, notamment en sciences, y compris dans le cadre de démonstrations mathématiques. L'évolution de celle-ci repose de manière croissante sur la puissance de calcul qu'apportent les ordinateurs. Or, jusqu'au début des années 2000, la puissance d'un ordinateur dépendait sensiblement de la fréquence du processeur qui effectue les calculs, laquelle a été confrontée à une limite physique au delà de laquelle les fuites de courants étaient trop importantes, allant même jusqu'à faire fondre les processeurs [Nam Sung Kim et al. 2003].

Depuis, pour outrepasser cette limite, les processeurs se voient pourvus de cœurs de calcul de plus en plus nombreux. En dehors de notions élémentaires, l'utilisation de l'informatique comme outil pour résoudre des problèmes nécessite surtout des connaissances dans le domaine du problème. Mais l'utilisation concurrente de plusieurs cœurs de calcul introduit des difficultés propres à l'informatique et donc une expertise que n'ont pas la plupart de ses utilisateurs, et qu'ils ne devraient pas être contraints à maîtriser. Ces difficultés couvrent des sujets variés, incluant la connaissance du fonctionnement du matériel, de son interaction avec les couches logicielles les plus basses jusqu'à des principes propres à la programmation concurrente : gestion de la communication entre les éléments concurrents ; synchronisation du travail effectué ; implémentation de patrons de conception dédiés ; maintien de la répétabilité malgré l'utilisation de nombres pseudo-aléatoires ; ... Du fait de ces difficultés, une majorité de programmes sont développés sans bénéficier réellement du potentiel du matériel à disposition.

Des solutions ont naturellement été étudiées, permettant d'abord l'utilisation des mêmes primitives indépendamment de l'architecture exacte sur laquelle est déployé le programme, là où initialement chaque système fournissait sa propre interface. Des instructions spécifiques, mettant en place tout ce qui est nécessaire à la parallélisation, ont été introduites, notamment au sein de certains compilateurs. Cette technique va jusqu'à la création de langages dédiés exposant des syntaxes adaptées. Tout cela a participé à assister les développeurs dans l'écriture de programmes s'exécutant en parallèle sur plusieurs processeurs.

Il s'est également agit de proposer l'automatisation de la génération de programmes parallèles à partir de leur écriture séquentielle classique. Différentes techniques existent, dont, à nouveau, l'exploitation des compilateurs pour analyser le code source et produire, sans que soit nécessaire une intervention de la part du développeur, un programme dont l'exécution est parallèle [Zima et al. 1988 ; Blume et al. 1995 ; Ahmad et al. 1997 ; Fonseca et al. 2016]. Similairement, des langages de programmation se sont mis à intégrer des constructions orientées vers la parallélisation et plus ou moins transparentes pour le développeur [Roscoe et Charles Antony Richard Hoare 1988 ; Loveman 1993 ; Chamberlain et al. 2007 ; Rieger et al. 2019].

Autant pour l'implémentation d'outils simplifiant la mise en œuvre de la parallélisation que pour son automatisation, de nombreuses bibliothèques logicielles sont disponibles et permettent

d'apporter à l'existant – langages et compilateurs – de nouvelles possibilités [Kuchen 2002 ; Chan et Abdelrahman 2004 ; Joel Falcou et al. 2008 ; Videau et al. 2018 ; Ernstsson et al. 2018]. Il est même possible, dans certains cas, qu'une bibliothèque logicielle (indépendante du compilateur) agisse pratiquement aussi profondément que le peut une extension de compilateur (à l'inverse, spécifique à celui-ci). Ces bibliothèques, dites actives [Veldhuizen et Gannon 1998], ouvrent la voie à de nouvelles façons de proposer des outils aux développeurs.

Les solutions aux problèmes de la parallélisation basées sur des bibliothèques logicielles actives sont relativement peu nombreuses par rapport aux autres approches. Elles sont plus contraignantes à produire qu'une extension de compilateur. Il est notamment difficile d'en écrire une qui ne cause effectivement pas un surcoût rédhibitoire en temps d'exécution, en mémoire utilisée ou encore en temps de compilation, comparé à une implémentation manuelle ou à ce qui se fait au niveau des compilateurs. Selon le cadre dans lequel la bibliothèque active est conçue, il peut également être particulièrement difficile de déboguer.

Le C++ et ses templates, supportant une forme de métaprogrammation à la base de ce qui permet l'élaboration de bibliothèques actives au sein de ce langage, est connu pour cela. En partie parce que ces templates n'ont pas été pensé initialement pour cela (ils servent avant tout à la dimension générique du langage), il est difficile de suivre ce qu'ils causent durant la compilation. Malgré cette limite, qui tend par ailleurs à s'amoindrir avec les évolutions du langage, cette forme de métaprogrammation est très employée car elle permet, lorsqu'elle est bien utilisée, d'atteindre des résultats très proches de ce que l'on obtiendrait en écrivant soi-même le code produit.

Avec cette thèse, nous voulons proposer principalement deux nouveaux outils. Le premier est un outil de parallélisation automatique, avec une bibliothèque active détectant elle-même ce qui peut validement être exécuté en parallèle et qui génère le code correspondant. La détection doit reposer sur des mécanismes extensibles afin de permettre l'ajout de nouvelles règles, déterminant, à partir des accès aux variables, ce qu'il est possible de faire de chaque instruction. Similairement, la génération de code doit avoir la capacité de fonctionner indépendamment de la méthode de parallélisation sous-jacente. Pour cela, il faut donc voir la manière de paralléliser comme une simple configuration qui peut être indiquée à la bibliothèque plutôt que comme un élément qui lui est central. À l'inverse, le fonctionnement de la bibliothèque ne doit pas affecter négativement et significativement la performance du programme par rapport à ce qui peut être accompli similairement « à la main ». Cela doit en particulier être vrai lorsque son utilisation aboutit à un programme non parallélisé.

Le second outil relève, quant à lui, de la parallélisation assistée, laquelle laisse au moins à la charge du développeur le soin de décider quelles sections du programme doivent être parallélisées avec une deuxième bibliothèque, également active. Pour ceci, nous proposons une nouvelle vision des squelettes algorithmiques avec davantage de contrôle de la part du développeur. Un but est de permettre une utilisation très flexible des squelettes algorithmiques, par exemple sur l'aspect de la transmission des paramètres entre les différentes parties d'un squelette. Un autre but est l'analyse des squelettes algorithmiques durant la compilation pour en déduire, par exemple, des optimisations possibles de celui-ci ou des schémas efficaces d'exécution parallèle. La parallélisation, particulièrement lorsqu'elle est combinée à l'utilisation de nombres pseudo-aléatoires, rend plus difficile la préservation de la répétabilité d'un programme, c'est-à-dire la capacité à exécuter plusieurs fois ce programme dans les mêmes conditions en obtenant systématiquement le même résultat. Sans répétabilité, la capacité de débogage est illusoire, et pour cette raison, nous voulons proposer un moyen de garantir cette répétabilité grâce aux squelettes algorithmiques.

Ce document présente les travaux effectués au titre de cette thèse. Pour ce faire, il est scindé en 5 chapitres. Le chapitre 1 traite de la parallélisation en partant des notions élémentaires matérielles et logicielles et en allant jusqu'aux concepts spécifiques de plus haut niveau qui permettent de résoudre des problèmes introduits par cette discipline. Ce chapitre détaille ensuite différentes couches supérieures qui simplifient l'application de la parallélisation, principalement sur les deux aspects que sont la parallélisation automatique et la parallélisation assistée.

Le chapitre 2 introduit un paradigme de programmation nommé « généricité », en particulier en C++. En effet, la généricité de ce langage fonctionnant durant la compilation grâce, entre autres, aux templates, est à la base de la métaprogrammation template. Y sont donc traitées la création et l'utilisation de templates, ainsi que leur spécialisation et les contraintes de sélection associées à celles-ci. Ce chapitre aborde aussi des problématiques spécifiques à cette généricité et les solutions qui y répondent.

Le chapitre 3 détaille ensuite la métaprogrammation, notamment celle du C++ avec la notion de template, les travaux effectués pour cette thèse reposant tous sur cette possibilité du langage C++. En utilisant les notions de généricité du chapitre précédent, il présente l'implémentation de différents algorithmes dont le déroulement s'effectue durant la compilation du programme. Il progresse sur ce sujet jusqu'aux patrons d'expression, une technique en métaprogrammation template permettant la représentation et l'utilisation, durant la compilation, de portions de code source.

Le chapitre 4 présente une bibliothèque active en C++ et utilisant la métaprogrammation template pour la parallélisation automatique de boucles. Cette proposition est expliquée en commençant par le détail des tests effectués par la bibliothèque pour déterminer l'indépendance d'instructions et leur capacité à être exécutées en parallèle sans créer de conflit d'accès aux données. Ensuite, le chapitre traite de la détection des données nécessaires à l'application de ces tests, qui repose sur des patrons d'expression, et qui permet d'acquérir une vision fine des instructions. La génération du code éventuellement parallèle, incluant donc l'exécution des tests durant la compilation, est subséquemment exposée. Pour finir, l'efficacité de l'utilisation de cette bibliothèque est évaluée au moyen de mesures de temps de compilation et d'exécution. Cela est effectué pour différents cas et pour des codes utilisant la bibliothèque par rapport à des codes équivalents ne l'utilisant pas.

Le cinquième et dernier chapitre présente une seconde bibliothèque active, utilisant également la métaprogrammation template du C++, cette fois-ci orientée vers la parallélisation assistée au moyen de squelettes algorithmiques. Ce chapitre introduit un problème de RO (Recherche Opérationnelle) et des métaheuristiques utilisées pour le résoudre. Ces métaheuristiques sont ensuite utilisées pour illustrer les concepts généraux de squelettes algorithmiques ainsi que ceux propres à notre proposition. Le chapitre se poursuit avec l'étude de la répartition des tâches à exécuter en parallèle en fonction du squelette algorithmique et de sa structure. Le problème de la répétabilité des exécutions parallèles en tenant compte, notamment, de l'utilisation de nombres pseudo-aléatoires est ensuite traité. Après les premières sections qui présentent chacune un aspect de la conception des squelettes algorithmiques en utilisant cette bibliothèque, ce chapitre montre comment ceux-ci peuvent être utilisés en pratique, notamment par le biais d'une syntaxe alternative définissant un langage embarqué au sein du C++. Cette bibliothèque est testée pour résoudre des instances de TSP (*Travelling Salesman Problem*) et est comparée en temps d'exécution avec des implémentations équivalentes ne l'utilisant pas.

Chapitre 1

Parallélisation automatique et assistée

1.1 Introduction

La puissance des ordinateurs croît en fonction de plusieurs critères. Si la vitesse des accès aux différentes ressources est à considérer, celle des processeurs est évidemment importante. La performance des processeurs, mesurée en particulier par leur fréquence d'horloge, depuis leur création, a augmenté exponentiellement pendant des dizaines d'années [Mack 2011] mais cette amélioration est contrainte entre autres par des limites physiques [Kish 2002]. De ce fait, les fréquences d'horloge n'ont plus guère augmenté depuis presque deux décennies. L'utilisation de plusieurs cœurs de calcul au sein des processeurs permet de pallier ce problème, nécessitant le recours aux paradigmes de programmation parallèle.

Ce chapitre présente dans un premier temps la programmation parallèle : les différents types de parallélisme et les différentes architectures ainsi que les difficultés que cela engendre. Ensuite, deux axes permettant de réduire ces difficultés sont introduits : d'abord la parallélisation automatique dont l'objectif est de produire un programme parallèle à partir d'un programme séquentiel ; puis des abstractions permettant d'aider à l'écriture de programmes parallèles qui se situent entre la parallélisation automatique et la programmation parallèle et que nous qualifions de parallélisation assistée.

1.2 Généralités

L'écriture d'un programme capable d'utiliser plusieurs processeurs est un exercice différent et plus difficile que celui d'écrire un programme « classique », c'est-à-dire n'utilisant qu'un processeur. La tâche est particulièrement ardue puisqu'il faut encore considérer la grande variété des architectures parallèles, ainsi que l'évolution des architectures existantes.

Par ailleurs, il est possible de raisonner sur l'intérêt de la parallélisation en considérant, par simplification, que tout programme peut être divisé en une sous partie séquentielle irréductible et une sous partie idéalement parallélisable. Une partie idéalement parallélisable implique que le gain obtenu en utilisant n cœurs peut être directement d'un facteur n, facteur que l'on nomme accélération (*speedup* en anglais). Pour cela sont ignorées certaines contraintes techniques telles que le surcoût induit par la gestion des tâches ou encore le risque de saturation de bus matériels pour l'accès aux données.

Amdahl [Amdahl 1967] a utilisé cette modélisation pour affirmer que l'accélération dont peut bénéficier un programme grâce à sa parallélisation sur N cœurs est limitée par sa fraction non parallélisable, notée $1 - f_a$ avec $f_a \in [0, 1]$ sa fraction parallélisable. Il en découle l'équation (1.1) (où $s_{f_a}(n)$ est l'accélération, étant donnée une fraction parallélisable f_a, en fonction du nombre de cœurs $n > 0$) qui, si l'on fait tendre n vers l'infini, borne l'accélération (équation (1.2)).

$$s_{f_a}(n) = \frac{1}{(1 - f_a) + \frac{f_a}{n}} \tag{1.1}$$

$$\lim_{n \to \infty} s_{f_a}(n) = \frac{1}{1 - f_a} \tag{1.2}$$

Ainsi, plus f_a est petit, moins la valeur de n affecte l'accélération obtenue, jusqu'au cas extrême trivial pour $f_a = 0$ qui donne $s_0(n) = 1$. Cette loi indique que même pour une valeur de f_a optimiste supérieure à 0,95, l'accélération que l'on peut espérer est seulement de 20, laissant penser que la parallélisation avec un grand nombre de cœurs ne peut être réellement efficace.

Cette analyse a été révisée par Gustafson [Gustafson 1988] qui a étudié le problème dans le sens inverse. Pour cela, il a considéré le temps d'exécution t_n du programme exécuté en parallèle avec n cœurs, et, prenant en compte la fraction parallélisable $f_g \in [0, 1]$, a calculé le temps que prendrait ce programme à s'exécuter sur un seul cœur (équation (1.3)).

$$t_1 = (1 - f_g)\, t_n + f_g\, t_n\, n \tag{1.3}$$

Il suffit alors de calculer le rapport entre t_1 et t_n pour connaître l'accélération obtenue (équation (1.4)) :

$$\begin{aligned} s_{f_g}(n) &= \frac{t_1}{t_n} && (t_1 : \text{équation (1.3)}) \\ &= \frac{(1 - f_g)\, t_n + f_g\, t_n\, n}{t_n} \\ &= (1 - f_g) + f_g\, n. \end{aligned} \tag{1.4}$$

Cette loi fait prendre conscience que dans la réalité des applications parallélisables, le volume des traitements à effectuer n'est pas fixe. Ceci implique que tant que le problème est d'une taille croissante et que l'architecture sur laquelle le programme est exécuté l'est également, l'accélération qu'il est possible d'avoir n'est pas limitée.

Comme cela a été dit, cette seconde loi découle d'une interprétation différente de la même réalité, et il est facile de démontrer qu'elles sont en fait deux versions d'une même formule, utilisant des variables différentes [Shi 1996]. Si la première, la loi d'Amdahl, considère sa fraction séquentielle $1 - f_a$ comme le rapport de temps passé dans des parties non parallélisables du programme sur le temps total lorsque celui-ci est exécuté sur un seul cœur, donc de manière séquentielle, ce n'est pas le cas de la seconde. La loi de Gustafson utilise comme fraction séquentielle le même rapport, mais pour une exécution parallèle utilisant un certain nombre de cœurs.

Cette formule reste fondée sur des données simplifiées et ne prend pas en considération des facteurs qui jouent un rôle important dans l'accélération réelle obtenue.

Elle ne doit donc pas être utilisée (quelle qu'en soit la version) comme véritable argument en faveur ou défaveur de la parallélisation mais plutôt comme un outil pour estimer le gain idéal qui pourrait en découler.

La parallélisation n'est absolument pas vouée à disparaître, au contraire, et il est nécessaire de disposer d'outils permettant, en particulier aux personnes qui ne sont pas spécialistes de ce domaine, de produire des programmes qui utilisent au maximum les possibilités offertes par la machine sur laquelle ils sont exécutés.

1.3 Parallélisme

Il convient dans un premier temps de distinguer « parallélisme » et « concurrence » bien que les deux domaines aient de nombreux points communs. Il y a concurrence lorsque plusieurs tâches sont ordonnancées indépendamment les unes des autres. Cela signifie que lorsqu'une tâche est bloquée, les autres peuvent poursuivre leur exécution. En revanche, les tâches ne sont pas nécessairement exécutées simultanément : il est possible de faire de la programmation concurrente en ne disposant que d'un cœur. Lorsqu'il s'agit de disposer de plusieurs tâches pour améliorer

les performances (par exemple dans le domaine du calcul à haute performance, en anglais HPC (*High Performance Computing*)), ce n'est pas de la programmation concurrente mais de la programmation parallèle (aussi nommée parallélisme). Parmi les premières implémentations de la concurrence sur les monoprocesseurs, on peut noter l'introduction des co-routines par [Dahl et Nygaard 1966] dans le langage Simula.

Le parallélisme peut être accompli sur un grand nombre d'architectures matérielles, chacune fonctionnant différemment et imposant donc des contraintes variées. Cette section traite dans un premier temps des différents cas relatifs à la manière d'accéder à la mémoire. Les axes sur lesquels peuvent s'appliquer le parallélisme sont ensuite présentés. Enfin, cette section explique les bases du parallélisme par les *threads*. Les points abordés dans ce chapitre sont orientés vers l'objectif général de cette thèse dont les applications sont en mémoire partagée, utilisant des *threads*.

1.3.1 Accès à la mémoire

Il faut principalement distinguer deux types d'accès à la mémoire : l'accès à une mémoire partagée et l'accès à une mémoire distribuée. Ces deux archétypes, souvent combinés dans la réalité, sont présentés dans cette section.

1.3.1.1 Mémoire partagée

La mémoire principale (en opposition aux mémoires locales des processeurs sous la forme de différents niveaux de cache) peut être unique et donc partagée par les différents processeurs, ce qui constitue l'architecture la plus répandue actuellement au sein des micro-processeurs modernes. Il s'agit alors de SMP (*Symmetric MultiProcessing*) et il existe principalement deux architectures d'accès à la mémoire principale : l'accès uniforme à la mémoire (UMA (*Uniform Memory Access*)) et l'accès non uniforme à la mémoire (NUMA (*Non-Uniform Memory Access*)).

Dans le premier cas, le temps d'accès à la mémoire ne dépend pas de quel processeur effectue cet accès : dans la figure 1.1, chaque processeur accède par le même mécanisme à la mémoire centrale (UMA). À l'inverse, la figure 1.2 schématise une architecture NUMA possible. Si toute la mémoire est directement accessible depuis chaque processeur de manière implicite, les temps d'accès peuvent être différents selon le processeur et la zone mémoire à laquelle on accède. En effet, la mémoire existe en plusieurs localités, et une mémoire plus proche sera donc plus rapide d'accès. Connaître le type d'architecture est important et permet d'observer de meilleures performances [Li et al. 2013].

Du côté logiciel, la programmation pour une machine SMP peut être faite en utilisant de multiples processus, que ce soit d'un même programme ou de différents programmes. Les processus pourront alors s'exécuter chacun sur un cœur différent et communiquer entre eux au moyen de mécanismes tels que la « communication inter-processus » (en anglais IPC (*Inter-Process Communication*)) grâce auxquels ils peuvent par exemple partager un segment de mémoire, utiliser des files de messages, ...

La création et l'utilisation d'un processus est une tâche coûteuse en temps d'exécution : deux processus ne partagent pas d'état, en particulier leur espace en mémoire. Lors de la création d'un nouveau processus, il est donc nécessaire d'allouer un nouvel espace mémoire, puis, lorsqu'une commutation de contexte survient, l'état du processus sortant doit être sauvegardé et celui du processus entrant restauré.

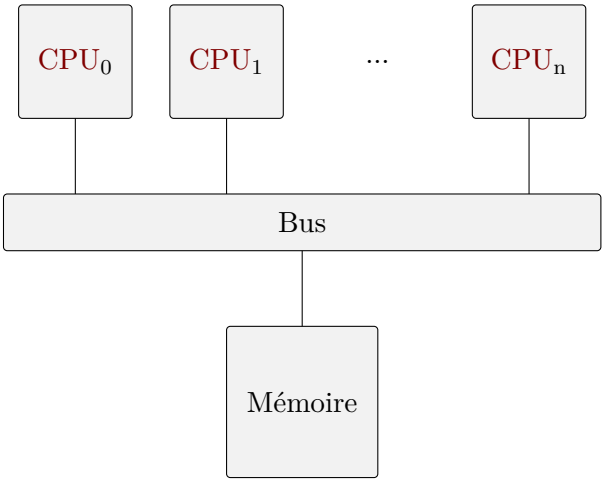


FIGURE 1.1 – Schéma d'architecture UMA à mémoire partagée

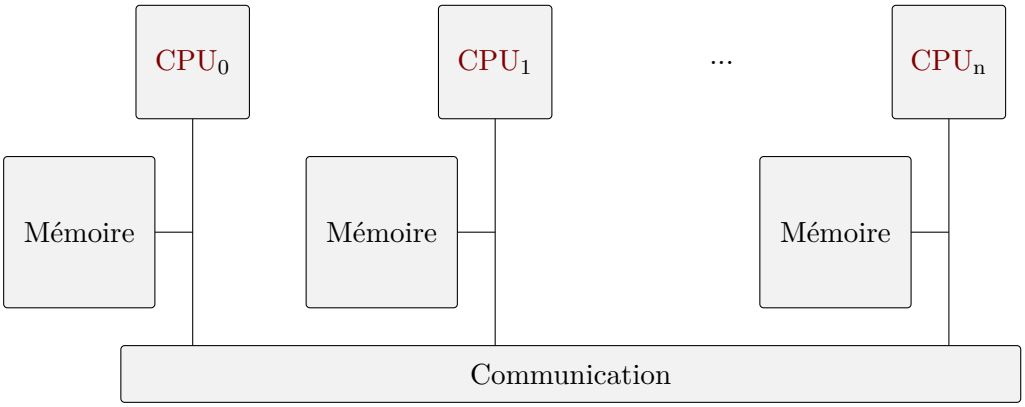


FIGURE 1.2 – Schéma d'architecture NUMA à mémoire partagée

Les processus légers, plus communément appelés *threads*, diffèrent des processus « lourds », appelés classiquement « processus », puisqu'ils partagent justement une grande partie de leur état, l'espace mémoire notamment (la pile restant propre à chaque *thread*). Ceci permet aux *threads* d'être moins coûteux en termes d'encombrement mémoire mais aussi en facilité de commutation de contexte d'exécution [Barney 2009], et en particulier lors de leur création. Ces caractéristiques justifient d'éviter l'utilisation de multiples processus pour paralléliser de manière particulièrement fine un traitement. Outre cette différence, un *thread* se comporte à l'identique d'un processus. Le fait qu'ils partagent en particulier leur espace en mémoire peut simplifier la communication entre les *threads*, mais cela soulève également certaines problématiques qui seront présentées dans ce chapitre.

De nombreux outils reposent sur les *threads* et permettent d'en simplifier l'utilisation. On peut par exemple citer OpenMP (*Open Multi-Processing*) [Dagum et Menon 1998], une API (*Application Programming Interface*) pour les langages C, C++ et Fortran ; CUDA (*Compute Unified Device Architecture*) [Luebke 2008] qui est orientée pour une catégorie de *threads* qui s'exécutent sur les nouvelles générations de processeurs graphiques (GPU (*Graphics Processing Unit*)) qui sont devenus de plus en plus généralistes ; OpenCL (*Open Computing Language*) [Munshi 2009], à la fois une API et un langage, conçu à partir du langage C, permettant la programmation pour des systèmes hétérogènes.

1.3.1.2 Mémoire distribuée

À l'opposé de la mémoire partagée, il existe la possibilité d'avoir une mémoire distribuée. Dans ce cas, chaque processeur dispose de sa propre mémoire principale privée (figure 1.3). Contrairement à une architecture NUMA à mémoire partagée, si un processeur doit accéder à la mémoire d'un autre processeur, la communication est explicite et à la charge du développeur. Pour cela, un standard a été développé au moyen de la norme MPI (*Message Passing Interface*) [Walker et Dongarra 1996]. Les deux approches ne sont pas incompatibles et il existe des systèmes adjoignant plusieurs systèmes à mémoire partagée pour en former un plus complexe à mémoire distribuée [Protic et al. 1996].

Bien que les bibliothèques produites durant cette thèse aient été conçues pour être modulaires et facilement adaptables à différents mécanismes de parallélisation, les travaux effectués n'ont été appliqués que sur des systèmes à mémoire partagée avec une architecture UMA et dont

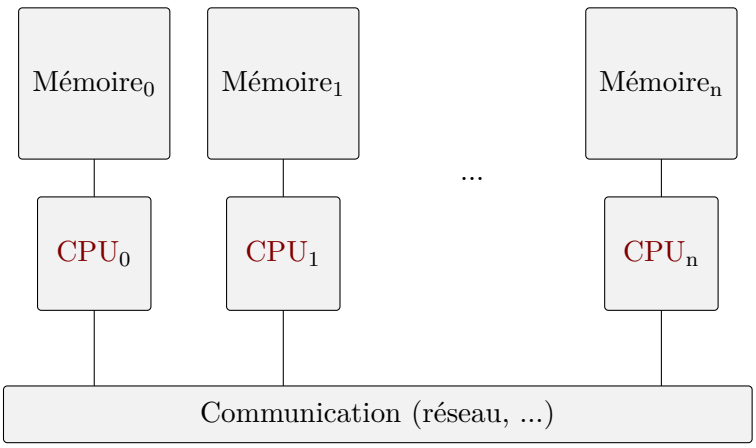


FIGURE 1.3 – Schéma d'une architecture multiprocesseurs à mémoire distribuée

l'implémentation repose sur les *threads* conformes aux normes IEEE (*Institute of Electrical and Electronics Engineers*) 1003, plus couramment désignées par POSIX (*Portable Operating System Interface*) [1], précisément POSIX.1c (ou IEEE 1003.1c). Pour cette raison, la suite de ce chapitre se concentre sur ces outils lorsqu'il s'agit de parallélisation manuelle.

1.3.2 Classification des architectures

La taxonomie de Flynn [Flynn 1972] définit 4 modèles selon qu'une ou plusieurs instructions sont appliquées à une ou plusieurs données simultanément, c'est-à-dire selon le nombre de flux d'instructions et de données. La programmation séquentielle correspond à un modèle SISD (*Single-Instruction stream – Single-Data stream*) (figure 1.4). Le cas un peu particulier de MISD (*Multiple-Instruction stream – Single-Data stream*) permet de traiter une même donnée de plusieurs manières et ne sera pas traité. Les deux modèles SIMD (*Single-Instruction stream – Multiple-Data stream*) et MIMD (*Multiple-Instruction stream – Multiple-Data stream*) couvrent les différents cas de programmation parallèle.

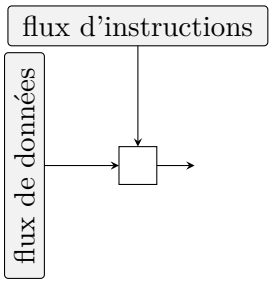


FIGURE 1.4 – Schéma du fonctionnement SISD
inspiré de https://en.wikipedia.org/wiki/File:SISD.svg

1.3.2.1 Parallélisme de donnée

Le parallélisme de donnée correspond au modèle SIMD : le flot d'exécution permet à une instruction d'être exécutée sur un ensemble de flux de données (figure 1.5). L'axe de parallélisation étant celui des données, plus il y en a à traiter, plus il est possible de bénéficier d'une forte accélération par la parallélisation. On parle d'approche vectorielle de la parallélisation telle que l'a introduite [Cray 1978]

Un exemple classique est le traitement d'image matricielle pour lequel les mêmes instructions doivent être exécutées pour tous les pixels qui peuvent donc être groupés de sorte à ce que chaque cœur traite une partie de l'image. C'est pour ce type de traitement spécifique qu'ont été conçus les GPU qui sont les processeurs correspondant au modèle SIMD les plus connus et les plus répandus après l'introduction des supercalculateurs vectoriels de marque *CRAY*. Sur ces processeurs, il est par exemple possible d'exécuter des *fragment shaders* : un programme prenant en entrée les coordonnées du pixel sur lequel il agit et retournant une couleur (des données supplémentaires telles que le temps peuvent être utilisées). Un tel programme permet d'altérer des données existantes (filtre pour rendre l'image plus floue) ou de créer entièrement une nouvelle image, par exemple par des techniques comme le lancer de rayon couplées à des fonctions de distance qui encodent les objets [Tomczak 2012] (figure 1.6).

1. Quant à la présence du X dans l'acronyme : https://stallman.org/articles/posix.html.

Les cœurs d'un GPU ne sont pas aussi généralistes que les cœurs des CPU (*Central Processing Unit*), mais ils sont en revanche très nombreux. Les autres processeurs conçus pour du parallélisme de donnée présentent généralement une structure similaire, mais possèdent des caractéristiques différentes de celles d'un CPU classique, dont une fréquence d'horloge plus faible. Ainsi, ils sont optimisés pour effectuer un traitement plus simple mais très hautement parallélisé avec une approche vectorielle.

En conséquence, les GPU ont été utilisés pour effectuer des traitements plus génériques et l'on parle alors de GPGPU (*General-Purpose computing on GPU*). S'il est possible d'exécuter de tels traitements au moyen de *shaders*, cela requiert l'encodage des données à traiter sous forme d'image sur laquelle travaillera le *shader*. Des outils ont donc été conçus pour simplifier la mise en œuvre de GPGPU. Parmi ceux-ci, C++ AMP (C++ *Accelerated Massive Parallelism*) [Lopez et al. 2016] de Microsoft (une norme couplée à une bibliothèque implémentée en C++ et fonctionnant

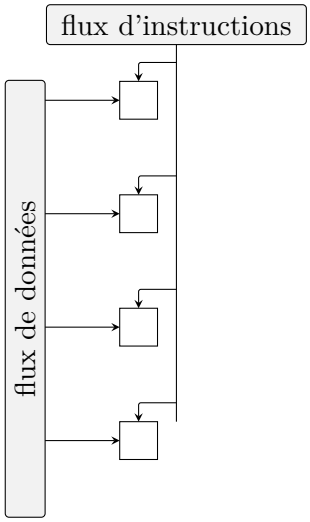


FIGURE 1.5 – Schéma du fonctionnement SIMD

inspiré de https://en.wikipedia.org/wiki/File:SIMD.svg

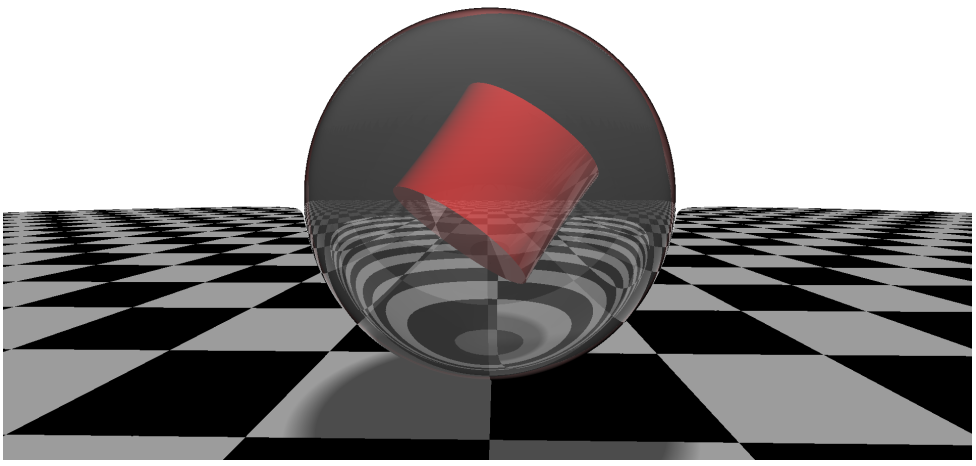


FIGURE 1.6 – Résultat d'un *fragment shader*
version complète et dynamique en ligne : https://www.shadertoy.com/view/wslBzr

au-dessus de DirectX 11), CUDA [Luebke 2008] de Nvidia (une interface de programmation d'application (en anglais, API) pouvant fonctionner avec les langages C, C++ et Fortran), OpenCL [Munshi 2009] de Apple Inc. et Khronos Group (un standard et une API pour la programmation sur des architectures hétérogènes, incluant les GPU mais aussi disponibles pour toutes sortes de FPGA (*Field-Programmable Gate Array*)).

Les CPU sont également conçus pour supporter des instructions SIMD. Pour l'architecture x86, très répandue parmi les ordinateurs personnels et serveurs, on peut citer, par ordre d'apparition, les instructions MMX (*multimedia extensions*) [Peleg et al. 1997], SSE (*Streaming SIMD Extension*) [Raman et al. 2000] et AVX (*Advanced Vector eXtensions*) [H. Zhang et al. 2018].

1.3.2.2 Parallélisme de tâche

Le parallélisme de tâche correspond au modèle MIMD : plusieurs flux d'instructions sont exécutés, chacun sur un flux propre de données (figure 1.7). La parallélisation intervient sur les deux axes : données et instructions. Ce type de parallélisme permet ainsi de bénéficier d'une accélération plus élevée lorsque le nombre de données, ou le nombre d'instructions pouvant être parallélisées, augmente.

Tout processeur disposant de plusieurs cœurs, c'est-à-dire presque la totalité des processeurs actuels pour ordinateurs personnels (incluant les téléphones) et serveurs, permet ce parallélisme.

1.3.3 Fondamentaux du parallélisme par *thread*

Cette section traite du parallélisme suivant le modèle MIMD sur une architecture à mémoire partagée avec un accès supposé uniforme à la mémoire en utilisant les *threads* POSIX, usuellement désignés par pthreads. Il s'agit de présenter les concepts fondamentaux sur lesquels reposent les outils de plus haut niveaux qui sont abordés ensuite. Les pthreads sont plus complètement détaillés dans *Programming with POSIX Threads* [Butenhof 1997]. Ce livre est une source importante de cette section.

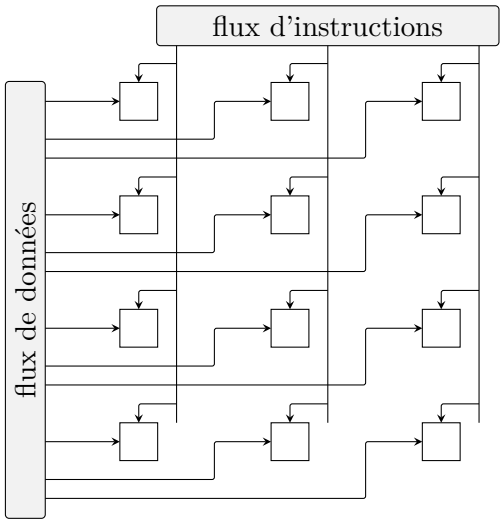


FIGURE 1.7 – Schéma du fonctionnement MIMD

inspiré de https://en.wikipedia.org/wiki/File:MIMD.svg

1.3.3.1 Gestion des *threads*

La vie d'un *thread* est similaire à celle d'un processus. La figure 1.8 représente les différents états dans lesquels peut se trouver un *thread* et comment se font les transitions entre ces états.

Lorsqu'un programme est exécuté, un nouveau processus est créé au sein duquel l'unique flot d'exécution représente alors le *thread* principal. La primitive correspondante de l'API pthread est `pthread_create`. Contrairement au mécanisme de création d'un processus qui duplique entièrement le processus père puis exécute, habituellement, une autre branche du programme, dans le cas des *threads* il ne s'agit que de l'exécution, dans un flot indépendant, d'une fonction. Le nouveau *thread* entre alors dans l'état « prêt ».

L'ordonnanceur du système d'exploitation traite alors le *thread* de la même manière qu'un processus : lorsqu'il le décidera, c'est-à-dire lorsque le *thread* en cours d'exécution quittera cet état, il passera le *thread* dans l'état « exécution ». À partir de cet état, un *thread* peut soit être préempté par l'ordonnanceur après un certain temps d'exécution, soit terminer son exécution (normalement ou en étant annulé par un autre *thread*), soit se retrouver dans l'état « bloqué » pour attendre la disponibilité d'une ressource. Le système retient la raison du blocage, et lorsque celle-ci disparaît, il change l'état du *thread* de « bloqué » à « prêt ».

Un *thread* peut être bloqué en attendant, à l'instar des processus, la possibilité de lire le contenu d'un fichier, qu'une connexion réseau soit établie, ... Cependant, le fait que tous les *threads* d'un processus partagent la même mémoire crée une autre ressource qui peut être bloquante : l'accès aux données partagées par les *threads*. Ceci induit des problématiques de synchronisations entre les *threads* qui sont l'objet de la section suivante.

La primitive `pthread_exit` permet à un *thread* de passer lui-même dans l'état « terminé ». Ceci est équivalent à atteindre la fin de la fonction qu'il devait exécuter. Un *thread* peut en terminer un autre en annulant son exécution avec la primitive `pthread_cancel`. La terminaison d'un *thread* implique la restitution au système des ressources qui lui ont été allouées. Lorsqu'un *thread* est dans l'état « terminé », il doit donc être traité pour libérer tout état lui étant propre, ce qui est accompli par la primitive `pthread_detach`. Cette fonction peut être utilisée sur un *thread* qui n'est pas terminé, auquel cas il est configuré pour libérer automatiquement ses ressources lorsqu'il se finira.

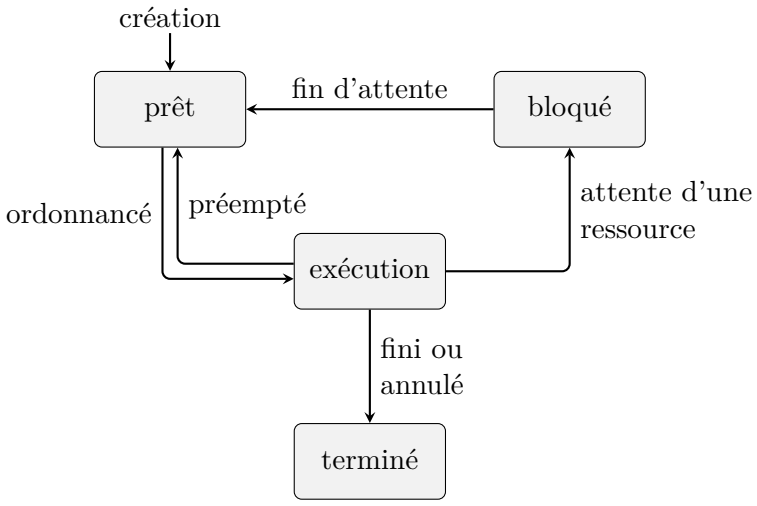


FIGURE 1.8 – États d'un *thread*

1.3.3.2 Synchronisation

Les différents *threads* d'un même processus partagent leur mémoire, à l'exception de la pile qui leur est propre. Ceci est un avantage pour la communication entre les *threads* qui peut être faite au moyen de variables partagées. En revanche, cela signifie également que les *threads* doivent être synchrones quant à leurs accès aux données communes : sans cela, il est possible d'obtenir des résultats incorrects. La figure 1.9 montre comment une insertion dans une liste chaînée peut se dérouler en trois étapes. L'insertion se fait avant le deuxième élément. La liste comporte initialement trois éléments a, b et c chaînés dans cet ordre (étape 1). Un nouveau maillon, x, est créé et pointe sur le second élément de la liste (étape 2). Enfin, le premier maillon, a, est modifié pour pointer sur le nouvel élément dans la liste (étape 3).

Sans prendre de précaution particulière, une exécution possible de deux *threads* appliquant exactement cet algorithme peut être celle représentée par la figure 1.10. Dans celle-ci, la première étape est inchangée et représente une liste chaînée composée de trois éléments. Un des deux *threads* crée un nouveau maillon, x, et le fait pointer sur le deuxième élément de la liste (étape 2). Il est alors possible que ce *thread* soit préempté par l'ordonnanceur, auquel cas un autre *thread* s'exécutant peut à son tour créer un nouveau maillon, y, et le faire pointer sur b (étape 3). Si le premier *thread* reprend son exécution à ce moment, il termine son travail en faisant pointer le premier maillon sur x (étape 4). Enfin, le second *thread* termine également son exécution en faisant pointer a sur son maillon y (étape 5). Le maillon x est définitivement perdu et la liste ne contient qu'un nouvel élément au lieu de deux.

La synchronisation des *threads* est donc indispensable lorsqu'ils agissent sur des données partagées. Les parties du code source dans lesquelles plusieurs *threads* peuvent accéder à ces données sont appelées des sections critiques. Le lecteur intéressé par ces concepts se reportera utilement à l'ouvrage de Quinn sur la programmation parallèle [Quinn 2003]. Les sections critiques participent à la fraction de temps d'exécution séquentielle du programme puisque dans ces sections, deux *threads* ne doivent pas pouvoir progresser simultanément.

Pour synchroniser plusieurs *threads*, il est possible d'utiliser les sémaphores [Dijkstra 1968]. Un sémaphore peut être décrit comme un entier non signé adjoint à une liste sur lequel deux opérations peuvent être réalisées :

— incrémenter l'entier du sémaphore de 1, nommée V ;
— décrémenter dès que possible l'entier du sémaphore (il ne doit pas valoir 0), nommée P.

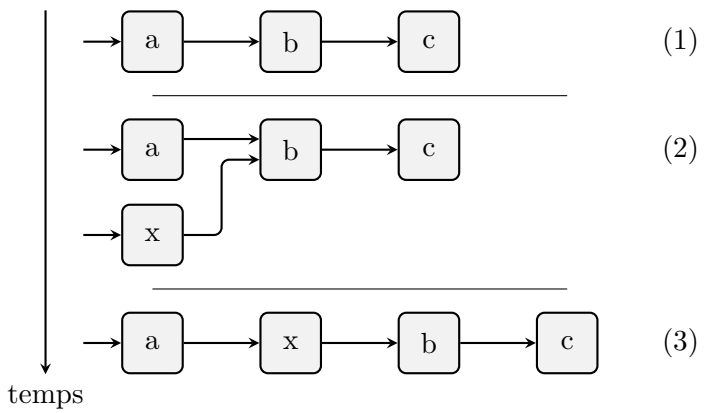


FIGURE 1.9 – Insertion dans une liste chaînée d'un élément en deuxième position

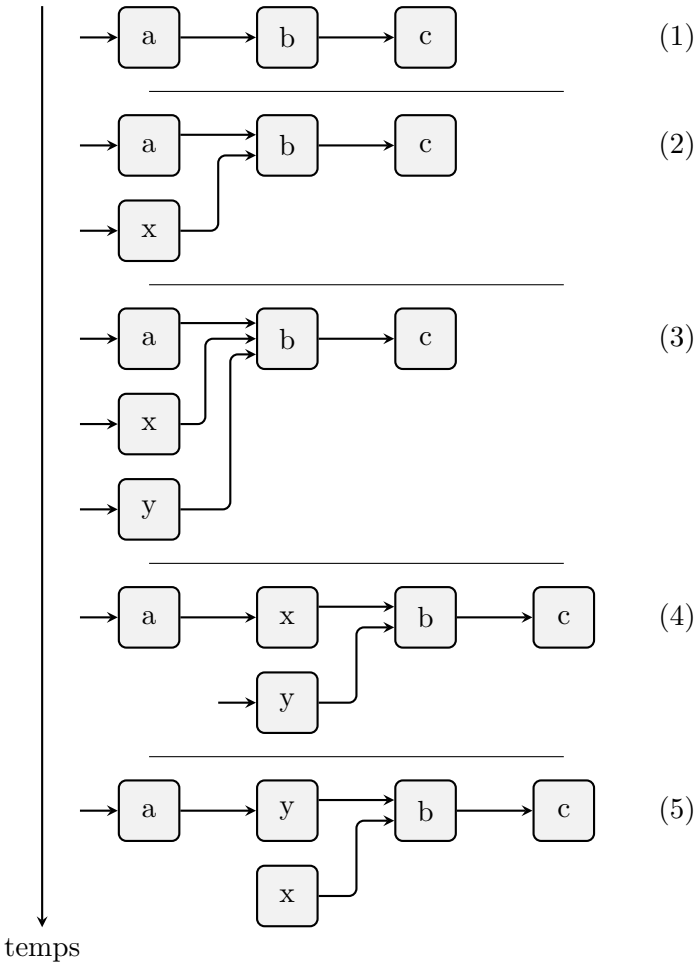


FIGURE 1.10 – Insertion incorrecte par deux *threads* dans une liste chaînée

Ce qui rend intéressante l'utilisation d'un sémaphore plutôt que d'une simple structure disposant d'un entier et d'une liste est l'atomicité garantie des opérations P et V. Une opération est atomique si le *thread* qui l'exécute ne peut pas être préempté par l'ordonnanceur avant de l'avoir terminée. L'opération V ne peut qu'augmenter le sémaphore de 1, en revanche l'opération P peut produire deux résultats. Si le sémaphore est non nul, il est simplement décrémenté et le *thread* peut continuer son exécution. Sinon, le *thread* est bloqué par l'attente d'une ressource : le sémaphore. Il sera automatiquement débloqué si le sémaphore est incrémenté, donc suite à l'exécution d'une opération V par un autre *thread*. Si plusieurs *threads* sont en attente d'un même sémaphore, ils sont placés dans une file d'attente de type PEPS (Premier Entré, Premier Sorti) (en anglais, FIFO (*First In, First Out*)).

Sur ce principe peut être construit l'outil de synchronisation par exclusion mutuelle [Courtois et al. 1971], en anglais *mutual exclusion* donnant le terme *mutex*. Un *mutex* permet de limiter à un seul *thread* l'accès à une donnée : lorsque celui-ci aura indiqué son entrée dans la section critique, tout autre *thread* voulant y entrer avant la sortie du premier sera bloqué. Ce cas particulier requiert un sémaphore initialisé à 1. La section critique doit alors être précédée de l'opération P, et suivie de l'opération V comme le montre l'algorithme 1.1. Les *threads* POSIX fournissent les structures (`pthread_mutex_t`) et les opérations (`pthread_mutex_lock` correspondant à P, `pthread_mutex_unlock` correspondant à V) nécessaires à la mise en place d'un *mutex*.

Algorithme 1.1 Utilisation d'un *mutex* pour protéger une section critique

```
1:                                          ▷ Le sémaphore m existe et a été initialisé à 1
2: procédure                                ▷ procédure exécutée par de multiples threads
3:     P(m)
4:     ...                                  ▷ section critique
5:     V(m)
```

En utilisant cette interface, le problème de l'insertion depuis de multiples *threads* dans une même liste chaînée (illustré par la figure 1.10) peut être résolu comme présenté dans l'extrait de code 1.1.

```c
#include <stdlib.h>
#include <pthread.h>

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void *task(void *arg) {
  int data = *(int*)arg;
  _Bool *success = malloc(sizeof *success);

  pthread_mutex_lock(&mutex);
  /* insertion non atomique dans une liste chaînée de la donnée data */
  /* la réussite de l'instruction est indiquée dans le booléen pointé par success */
  pthread_mutex_unlock(&mutex);

  return success;
}

int main() {
  int i;
  int values[] = {5, 7};
  pthread_t threads[2];

  for(i = 0; i < 2; ++i)
    pthread_create(threads+i, NULL, task, values+i);

  for(i = 0; i < 2; ++i) {
    _Bool *success;
    pthread_join(threads[i], (void**)&success);
    /* vérification de la réussite avec *success */
    free(success);
  }

  return 0;
}
```

(⩾ C99)

Extrait de code 1.1 – Utilisation des *threads* POSIX en C

Un autre patron d'utilisation des sémaphores est la barrière de synchronisation [J. Hill et Skillicorn 1998]. Son objectif, lorsqu'il s'agit de synchroniser N *threads*, est de bloquer les $N-1$ premiers qui arrivent au point de synchronisation, en utilisant le sémaphore b, jusqu'à ce que le dernier l'atteigne également. Ce cas général peut être implémenté comme le montre l'algorithme 1.2. Cet algorithme utilise aussi un sémaphore m comme un *mutex* pour protéger la variable entière qui compte le nombre de *threads* arrivés à la barrière.

Algorithme 1.2 Barrière de synchronisation pour N_t *threads*

```
1:  procédure BARRIÈRE                          ▷ procédure exécutée par les N_t threads
2:      P(m)                                    ▷ sémaphore utilisé comme mutex pour protéger N_b
3:      N_b ← N_b + 1                           ▷ N_b compte le nombre de thread arrivés
4:      si N_b = N_t alors                      ▷ Le dernier thread exécutera la première branche
5:          N_b ← N_b − 1
6:          répéter
7:              V(b)                            ▷ libération des autres threads
8:              N_b ← N_b − 1
9:          jusqu'à ce que N_b = 0
10:         V(m)
11:     sinon                                   ▷ pour les N_t − 1 premiers threads
12:         V(m)
13:         P(b)                                ▷ bloqué jusqu'à l'arrivée du dernier thread
```

Certaines implémentations des sémaphores fournissent une opération supplémentaire Z qui permet de bloquer l'exécution jusqu'à ce qu'un sémaphore donné soit nul. Avec une telle possibilité, la barrière de synchronisation pour un nombre N de *threads* peut simplement initialiser un sémaphore à N et, pour chacun des N *threads*, effectuer l'opération P suivie de l'opération Z.

L'algorithme 1.3 présente une version simplifiée pour le cas particulier d'une barrière synchronisant seulement deux *threads*. Il arrive fréquemment que ce cas particulier survienne pour synchroniser la fin de l'exécution d'un *thread* avec un moment précis de l'exécution d'un autre. Le standard pthread fournit pour cela `pthread_join` que peut appeler un *thread* pour attendre la terminaison d'un autre. Cette fonction est donc bloquante si le *thread* décrit par l'identifiant donné en argument n'est pas déjà terminé. Le *thread* attendu est ensuite automatiquement détaché par `pthread_join`.

Un autre mécanisme de synchronisation est l'attente qu'un certain état soit vrai. Par exemple, un *thread* peut devoir travailler sur des données présentes dans une file d'attente, auquel cas il doit attendre que celle-ci contienne au moins un élément. Cela peut être accompli avec un sémaphore par l'utilisation de l'opération P dans le *thread* travailleur, et ce avant de retirer un élément de la file d'attente. Un autre *thread*, fournissant des données et donc remplissant la file d'attente, devra alors utiliser l'opération V après avoir ajouté un élément pour notifier du changement d'état.

Si ce schéma peut relativement facilement être étendu à de multiples travailleurs lorsqu'il s'agit de faire par exemple de l'équilibrage de charge entre les *threads*, c'est-à-dire qu'un seul *thread* doit être notifié lorsque la file d'attente acquiert un nouvel élément, il devient complexe

Algorithme 1.3 Barrière de synchronisation pour 2 *threads*

```
1:  procédure THREAD1                           ▷ procédure exécutée par l'un des deux threads
2:      ...
3:      V(a)
4:      P(b)
5:  procédure THREAD2                           ▷ procédure exécutée par l'autre thread
6:      ...
7:      P(a)
8:      V(b)
```

d'étendre réellement l'utilisation pour inclure par exemple un mécanisme de notification de tous les travailleurs.

L'API des *threads* POSIX propose pour cela le mécanisme de *condition variable*. Leur utilisation repose principalement sur les fonctions `pthread_cond_wait`, `pthread_cond_signal` et `pthread_cond_broadcast`. La première permet d'attendre la réception d'un signal qui sera émis par l'une des deux autres. La fonction `pthread_cond_signal` aura pour effet de débloquer un des *threads* en attente avec un sémaphore, comme cela est expliqué ci-avant dans la présentation de ce mécanisme de synchronisation. La fonction `pthread_cond_broadcast` en revanche réveillera tous les *threads* en attente.

Tous les cas d'utilisations de sémaphores ne sont évidemment pas présentés et les pthreads offrent également davantage de possibilités. Cependant, les cas présentés en montrent assez pour rendre compte des difficultés qui peuvent survenir avec la programmation *multi-thread*. Il est très facile de mal employer les nombreux mécanismes de synchronisations, ce qui peut résulter en des résultats incorrects, des interblocages [Howard 1973], ... De nombreux problèmes très connus [Gingras 1990 ; Y. Zhang et al. 2009] existent pour montrer les difficultés soulevées par la nécessité de synchronisation.

1.4 Parallélisation automatique

La section précédente a permis de montrer que la programmation parallèle est un exercice difficile. Tout développeur n'est ainsi pas nécessairement capable de programmer avec ces contraintes, et le travail requis pour se former peut dépasser les avantages obtenus : même si quasiment tout programme bénéficierait d'être développé en pensant à sa parallélisation, les gains sont variables. Cette difficulté a induit la recherche de méthodes d'automatisation afin de produire des exécutables qui utilisent plus efficacement le matériel à leur disposition sans nécessiter de la part du développeur un effort de programmation supplémentaire.

La parallélisation automatique consiste en la production d'un programme qui s'exécute en utilisant plusieurs processeurs sans que le code source ait besoin d'être modifié par un humain. Il s'agit ainsi de parallélisme implicite. L'outil qui procède à la parallélisation automatique doit en conséquence être capable de déterminer de lui-même quelles parties du programme peuvent être parallélisées et comment elles doivent l'être. Un outil de parallélisation automatique est proposé dans le chapitre 5.

1.4.1 Parallélisme matériel

De nombreux processeurs ont été conçus sur mesure, construits sur la base de FPGA (un processeur matériellement programmable) ou de DSP (*Digital Signal Processor*) (un processeur dédié au traitement du signal) pour gérer des données ou des calculs très nombreux en parallèle [Batcher 1980 ; Batlle 2002 ; Perach et Weiss 2018]. Ces circuits sont dédiés à un traitement spécifique et doivent être conçus pour chaque besoin. Il existe en outre, sur les processeurs plus généraux, différentes techniques de parallélisation qui sont maintenant très courantes : le BLP (*Bit-Level Parallelism*) ou « parallélisme au niveau du bit » et l'ILP (*Instruction-Level Parallelism*) ou « parallélisme au niveau des instructions ».

1.4.1.1 Parallélisme au niveau du bit

Les processeurs disposent d'unités dédiées aux calculs (UAL (Unité Arithmétique et Logique), UVF (Unité de calcul en Virgule Flottante), ...) que l'on peut schématiser par la figure 1.11. Sur cette figure, les entrées sont à gauche et les sorties à droite. En simplifiant, une UAL attend 2 entrées numériques (E_0 et E_1 dans la figure) sur n bits ainsi qu'un ensemble d'autres informations (E_f), par exemple l'opération à effectuer. La sortie principale (S) est également sur n bits. Les sorties supplémentaires (S_f) permettent d'indiquer par exemple l'occurrence d'un dépassement d'entier.

Ces unités de calcul peuvent ainsi travailler avec des nombres dont l'amplitude de valeur dépend de la taille des bus de données utilisés pour les nombres en entrée et en sortie (et qui sont de taille identique). Une telle unité de calcul prend également en entrée et fournit en sortie d'autres informations qui ne seront pas particulièrement détaillées, mais qui vont permettre la détection de dépassement de capacité, par exemple.

Lorsqu'un compilateur produit les instructions assembleur pour un processeur, il prend en compte ces caractéristiques. Dans un langage de haut niveau, une addition entre deux nombres sera ainsi convertie en l'instruction correspondante.

Le parallélisme au niveau du bit est une technique consistant en l'utilisation de bus de données plus grands, permettant l'envoi, en une seule instruction, de davantage de données. Après être passé de 4 à 8, puis 16, 32, 64... bits, le parallélisme au niveau du bit s'est intéressé à la vectorisation grâce aux travaux de [Cray 1978]. Par exemple, un processeur peut fournir une instruction pour effectuer 4 additions parallèles de 8 entiers, chacun sur 16 bits. Cette instruction acceptera donc deux nombres sur 64 bits.

Un compilateur peut générer automatiquement des instructions profitant de ce mécanisme de vectorisation [2]. On retrouve ce que nous avions introduit dans la section 1.3, cette technique étant utilisée depuis longtemps pour accélérer des calculs au sein même des microprocesseurs [Jackson et al. 1992], tout comme Seymour Cray les réalisait avec des composants de l'époque pour constituer ses unités vectorielles.

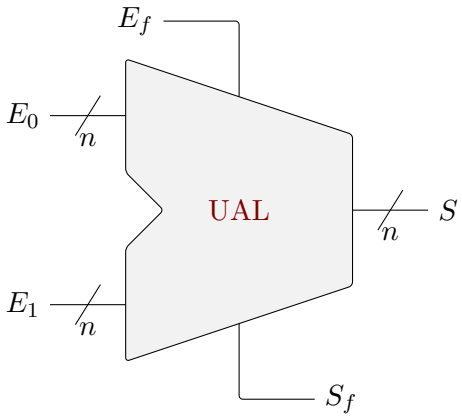


FIGURE 1.11 – Schéma de l'interface d'une unité arithmétique et logique

2. Dès lors que ces instructions sont comprises dans le jeu d'instructions qui lui est connu et permis.

1.4.1.2 Parallélisme au niveau des instructions

Les processeurs sont capables de procéder à du parallélisme au niveau des instructions. Plusieurs techniques existent, dont la chaîne de traitement d'instructions [Stallings 1988] (*pipeline*), l'exécution dynamique [Keller 1975] ou encore l'exécution spéculative [J. E. Smith 1998] qui peuvent être implémentées dans les processeurs.

L'exécution d'une instruction correspond à une séquence d'opérations telles que la récupération d'une instruction (IF (*Instruction Fetch*)), le décodage de celle-ci (ID (*Instruction Decode*)), son exécution (Ex), un éventuel accès à la mémoire (MA (*Memory Access*)) et enfin la sauvegarde en mémoire (WB (*WriteBack*)). Ces opérations sont effectuées par différentes parties du processeur et peuvent donc être effectuées simultanément pour différentes instructions (figure 1.12). Cette approche du parallélisme est appelée *pipeline*. Elle a été utilisée depuis longtemps dans tous types de chaînes de production manufacturières.

L'exécution dynamique, aussi appelée exécution dans le désordre (de l'anglais *out of order execution*) repose sur le fait que plusieurs instructions consécutives peuvent dépendre d'une même partie du processeur, auquel cas il n'est pas possible en l'état de les paralléliser par chaîne de traitement par exemple. Ce que [Keller 1975] a proposé est de regarder dans les instructions suivantes s'il en existe des compatibles avec la courante, et que l'on peut exécuter prématurément sans pour autant changer le comportement global des instructions. Dans ce cas, le processeur va exécuter les instructions dans un autre ordre que celui qui lui a été fourni, de manière à minimiser l'inoccupation des différentes sous-parties du processeur.

L'exécution spéculative permet d'exécuter des instructions avant de savoir si elles vont devoir l'être ou non. Ainsi, s'il est avéré par la suite que ces instructions devaient être exécutées, le travail est déjà accompli. En revanche, dans le cas contraire, du travail supplémentaire peut être requis pour annuler des opérations. Le cas classique utilise la prédiction de branchement : par une heuristique, le processeur suppose pour un branchement donné laquelle des deux branches va être exécutée, et ce avec de bons résultats [Patterson 1995]. Ces branchements sont causés par des structures de contrôle, et utilisant en assembleur des instructions de la famille des `jmp`.

Ces techniques sont habituellement directement implémentées par les processeurs bien qu'elles

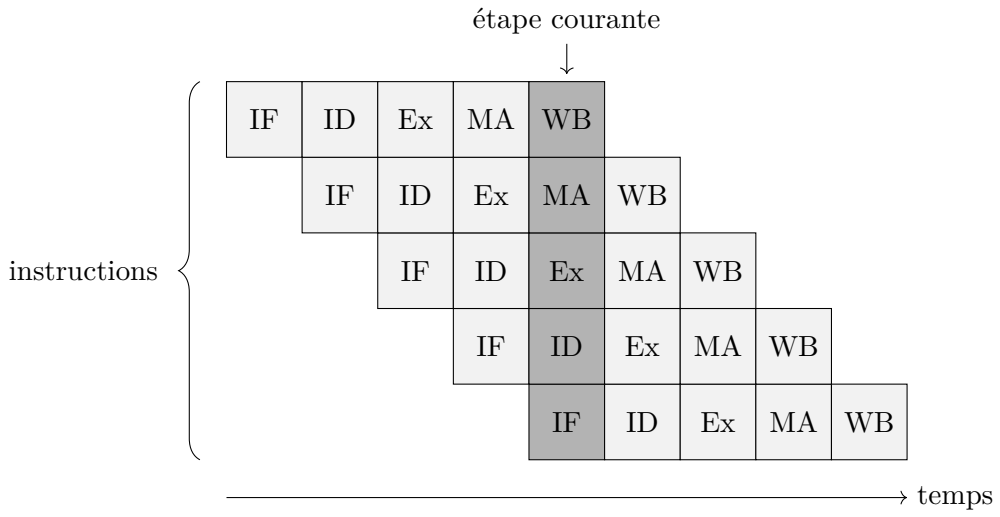


FIGURE 1.12 – Schéma d'une chaîne de traitement à 5 étapes, avec 5 instructions
inspiré de https://en.wikipedia.org/wiki/File:Fivestagespipeline.png

puissent également être conçues au niveau des compilateurs [Bringmann et al. 1993]. La plupart des processeurs faisant usage de ces techniques, tous les programme, y compris séquentiels, bénéficient automatiquement de cette parallélisation.

1.4.2 Parallélisme logiciel

La parallélisation peut être accomplie automatiquement à différents niveaux logiciels, que ce soit par un compilateur, un outil externe ou une bibliothèque [Mathews et Abraham 2016 ; Arabnejad et al. 2018]. Quel que soit l'outil, la figure 1.13 présente les étapes primaires nécessaires pour convertir l'entrée qu'est le programme séquentiel en programme parallèle. La première étape, l'analyse lexicale et syntaxique, comporte la reconnaissance du programme et de sa structure, qui va généralement être représenté en interne par un arbre syntaxique. Par exemple, l'algorithme 1.4 sera représenté par l'arbre syntaxique de la figure 1.14. Cet arbre décrit l'organisation des instructions d'un programme, chaque nœud représentant une instruction dont l'exécution est causée par l'exécution du nœud parent.

Algorithme 1.4 Algorithme calculant la somme des 10 premiers entiers

```
1: i = 10
2: r = 0
3: tant que i ≠ 0 faire
4:     r = r + i
5:     i = i − 1
6: retourner r
```

L'étape d'analyse lexicale et syntaxique est intrinsèque au fonctionnement d'un compilateur, et les mécanismes dont peut faire usage une bibliothèque sont traités dans le chapitre 4. Grâce à cette information, l'outil peut déterminer, par une analyse de dépendances, comment sont utilisées les variables, les dépendances entre les instructions, ... Enfin, à partir de ces informations, il est possible de générer le programme parallèle. À nouveau, c'est une étape naturelle au sein d'un compilateur, et les techniques employées par les bibliothèques seront traitées à part.

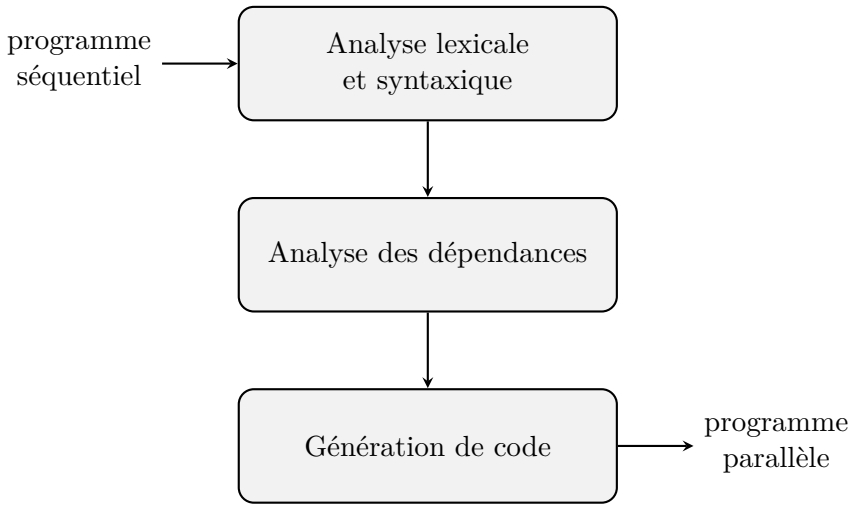


FIGURE 1.13 – Étapes de la parallélisation automatique d'un programme

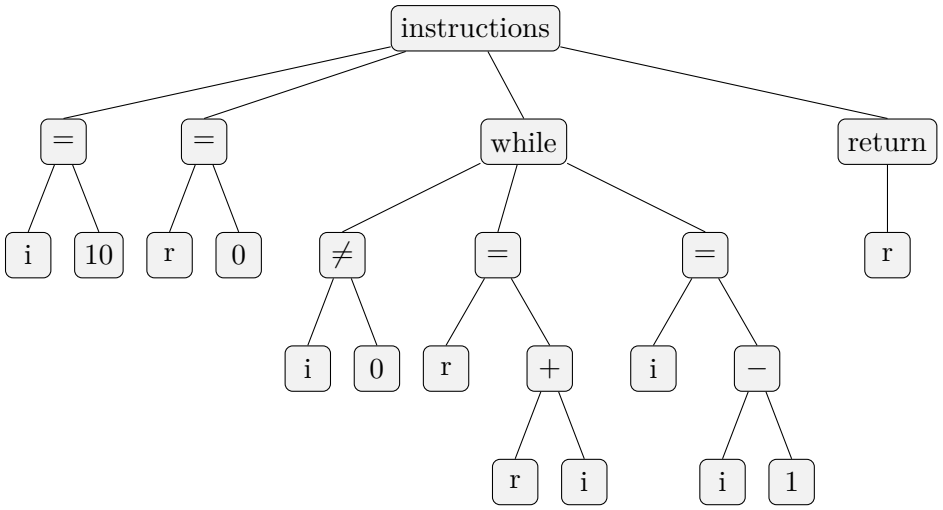


FIGURE 1.14 – Arbre syntaxique de l'algorithme 1.4

1.4.2.1 Analyse des dépendances

Un programme est une séquence d'instructions à effectuer qui ont été décrites dans un langage spécifique. Paralléliser un sous-ensemble de ces opérations ne peut être accompli que si ce qui est réalisé par un programme séquentiel est conservé lors de son exécution parallèle. Ainsi, certaines structures de programmes ne sont pas parallélisables car cela modifierait le comportement général du programme.

Il existe plusieurs types de dépendances :

— la dépendance de flux (RAW (*Read After Write*)) (algorithme 1.5) ;
— l'anti-dépendance (WAR (*Write After Read*)) (algorithme 1.6) ;
— la dépendance d'écriture (WAW (*Write After Write*)) (algorithme 1.7) ;
— la dépendance de contrôle (algorithme 1.8).

Une dépendance de flux apparaît lorsqu'une ressource est lue après avoir été modifiée : la valeur lue dépend de l'ordre d'exécution et est correcte si l'écriture est effectuée d'abord. Une anti-dépendance correspond à une ressource lue avant d'être modifiée. À nouveau, l'ordre

Algorithme 1.5 Dépendance RAW

1: $R_1 \leftarrow A$ ▷ *Une ressource utilisée en écriture...*
2: $R_2 \leftarrow R_1$ ▷ *... puis en lecture*

Algorithme 1.6 Dépendance WAR

1: $A \leftarrow R_1$ ▷ *Une ressource est utilisée en lecture...*
2: $R_1 \leftarrow B$ ▷ *... puis en écriture*

Algorithme 1.7 Dépendance WAW

1: $R_1 \leftarrow A$ ▷ *Une ressource utilisée en écriture...*
2: $R_1 \leftarrow B$ ▷ *... puis à nouveau en écriture*

Algorithme 1.8 Dépendance de contrôle

1: **si** $R_1 = A$ **alors**
2: $\quad R_2 \leftarrow B$ ▷ *L'exécution de cette instruction dépend de l'instruction* $R_1 = A$

d'exécution doit être conservé pour que la donnée lue soit correcte. La dépendance d'écriture survient lorsqu'une même variable est utilisée en écriture plusieurs fois de suite, l'ordre des écritures devant être conservé. Enfin, la dépendance de contrôle est liée aux structures de contrôle telles que le branchement conditionnel. L'exécution des instructions d'une branche du programme dépend alors d'une ressource utilisée pour déterminer quelle branche est choisie (ou pour une boucle, combien de fois elle est exécutée).

Ces dépendances affectent la parallélisation automatique à tous les niveaux, y compris, lorsqu'elles existent au sein des instructions pour le processeur, dans le cadre des optimisations telles que l'exécution dynamique [Hwu et Patt 1986]. Les dépendances WAR et WAW peuvent être contournées dans certains cas : si la ressource correspond à une variable, servant donc à retenir une information, il est possible d'introduire de nouvelles variables pour supprimer la dépendance [Nicolau et al. 1992]. L'algorithme 1.9 montre une solution pour rendre indépendantes les deux instructions présentes dans l'algorithme 1.6.

Algorithme 1.9 Résolution d'une dépendance WAR

1: $V \leftarrow R_1$ ▷ *Introduction d'une nouvelle variable*
2: $A \leftarrow V$ ▷ *Cette instruction ne dépend plus de la suivante*
3: $R_1 \leftarrow B$

En revanche, à une échelle d'instructions moins fine, une ressource peut correspondre, par exemple, à un fichier. Dans cet exemple, deux écritures consécutives ne peuvent jamais être désordonnées sans que le contenu final du fichier ne soit différent. Si l'ordre des écritures est important, alors la dépendance WAW est bloquante.

La difficulté d'analyse de ces dépendances est accrue lorsque l'on accède aux ressources de manière indirecte [Horwitz et al. 1989] comme c'est le cas dans de nombreux langages, par exemple ceux permettant l'utilisation de pointeurs (des variables contenant l'adresse en mémoire d'autres variables).

1.4.2.2 Dépendances entre les itérations d'une boucle

De très nombreuses études se sont concentrées sur la parallélisation des boucles [Collard 1995 ; Artigas et al. 2000 ; Ramon-Cortes et al. 2018 ; Neth et Strout 2019]. En effet, dans la plupart des programmes, la majorité du temps d'exécution est localisé dans des boucles, et il semble donc normal d'y consacrer beaucoup d'efforts si l'on souhaite améliorer les performances.

La parallélisation d'une boucle est soumise à des contraintes similaires à la parallélisation de deux segments d'un programme, mais dans ce cadre, la dépendance est portée par les différentes itérations et doit être vérifiée pour toutes ces itérations. Les boucles peuvent principalement être catégorisées selon leur condition d'arrêt. Celles dont on connaît à l'avance le nombre d'itérations (de nombreux langages utilisant classiquement pour celles-ci le mot-clé « `for` », ce nom est quelquefois utilisé pour les qualifier), et plus précisément le n-uplet des indices de l'itérateur utilisé, sont traitées dans le chapitre 4. Ce chapitre aborde en détail différentes optimisations applicables aux boucles, ainsi que les contraintes sur leur parallélisation. Les techniques de détection de dépendances et de génération de code y sont également expliquées.

Les autres boucles, dont le nombre d'itérations est *a priori* inconnu, sont couramment qualifiées de boucle « `while` ». Malgré l'usage, il ne faut cependant pas croire qu'une boucle « `for` » corresponde nécessairement à une boucle ayant un nombre d'itérations bien connu :

il est possible de les utiliser pour remplacer une boucle « `while` », ou encore de la quitter prématurément dans certaines conditions. À l'inverse, une boucle « `while` » peut être construite de telle sorte que le nombre d'itérations soit connu. La parallélisation automatique des boucles « `while` » a été étudiée, en particulier dans le cadre de boucles « `while` » imbriquées dans des boucles « `for` » [Martin Griebl et Collard 1995 ; Lengauer et M. Griebl 1995 ; Geuns et al. 2011].

Les données sur lesquelles agissent les boucles sont souvent des tableaux. Dans ce cas, il est possible de vectoriser l'exécution. Le principe est d'utiliser les instructions SIMD du CPU (MMX, SSE, AVX... Voir la section 1.3.2.1) pour exécuter la boucle par partie. Par exemple, une boucle de 16 itérations peut être transformée en boucle de 4 itérations, chacune exécutant de manière vectorisée 4 instructions.

La parallélisation peut également être accomplie automatiquement sur des structures de programmes plus complexes. Les structures de données de type arbre sont fréquentes en recherche scientifique, les algorithmes qui s'appliquent sur celles-ci sont donc souvent utilisés. Il existe ainsi de nombreux travaux dont l'objectif est la vectorisation, ou la parallélisation, automatique de différents algorithmes appliqués aux arbres [Jo et al. 2013 ; Matsuzaki et al. 2006].

1.5 Parallélisation assistée

Si la parallélisation automatique permet à un développeur sans connaissance dans le domaine du parallélisme de profiter davantage du matériel multicœur, cela reste limité à cause de l'analyse complexe du programme qui est nécessaire. Une analyse qui ne prend pas le risque de générer un programme dont le comportement dévie de l'original (ce qui est normalement attendu d'un outil de parallélisation automatique) est souvent amené à échouer à paralléliser des portions qui auraient pu l'être [Kennedy 1994].

Une approche intermédiaire, entre manuel et automatique, consiste à assister autant que possible le développeur, mais en lui laissant une part de la charge de décision. Les décisions concernées peuvent être de l'ordre du mécanisme bas niveau de parallélisation (*threads*, processus, ...), de quelles parties du programme doivent, ou ne doivent pas, être parallélisées, ou d'informations de plus haut niveau. Le chapitre 5 propose un outil entrant dans cette catégorie, la parallélisation assistée.

1.5.1 Abstractions

La section 1.3 (plus précisément, la section 1.3.3) a introduit les mécanismes du parallélisme basé sur les *threads*. Une première étape pour assister l'utilisateur est de fournir une API standard permettant d'utiliser les *threads* facilement et de manière portable : c'est ce qu'a permis l'API POSIX. Cependant, mal employer les différentes primitives et structures de données est facile et le langage C par exemple ne prévient pas certaines erreurs qui pourraient être évitées. Parmi celles-ci : oublier l'initialisation d'un *mutex* (qui est une structure complexe).

L'extrait de code 1.2 montre l'aspect d'un programme construit en utilisant uniquement les primitives de ce standard. Cet algorithme démarre un *thread* qui partage un *mutex* avec le *thread* principal. Chacun des deux *threads* travaille sur une donnée partagée (pour justifier l'utilisation d'un *mutex* dans cet exemple). Malgré la simplicité apparente (aucune problématique propre au parallélisme, sinon la protection d'une section critique, n'est abordée), cet algorithme présente déjà le risque de commettre plusieurs erreurs. Parmi celles-ci : oublier d'initialiser le *mutex*

(avec `PTHREAD_MUTEX_INITIALIZER` car le type associé aux *mutex* est en fait une structure) ; mal associer la prise et la libération d'un *mutex* ; ou encore oublier l'appel `pthread_join`.

Ces erreurs peuvent être évitées, par exemple, dans les langages orientés objets. Ces langages fournissent la possibilité d'exécuter une fonction à la création et à la destruction d'une instance d'une structure. Ainsi, l'extrait de code 1.3 représente le même programme que l'extrait de code 1.2, mais en utilisant l'interface d'une implémentation possible dans un tel langage. À l'instar de ce qui est accompli pour le *mutex*, il est possible d'utiliser le destructeur de `std::thread`

```c
#include <pthread.h>

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void *task(void *arg) {
  (void)arg; /* argument inutilisé */

  pthread_mutex_lock(&mutex);
  /* accès en écriture à une variable partagée */
  pthread_mutex_unlock(&mutex);

  return NULL;
}

int main() {
  pthread_t thread;

  pthread_create(&thread, NULL, task, NULL);
  task(NULL); /* appel exécuté dans le thread principal */

  pthread_join(thread, NULL);

  return 0;
}
```

(≥ C89)

Extrait de code 1.2 – Exemple simple utilisant des *threads* POSIX en C

```cpp
#include <thread>
#include <mutex>

std::mutex m; // initialisation faite par le constructeur

void task() {
  // Le constructeur prend le mutex m
  std::lock_guard<std::mutex> lock{m};
  // accès en écriture à une variable partagée
} // le destructeur de lock libère le mutex m

int main() {
  std::thread threadTask{task}; // création du thread faite par le constructeur
  task(); // appel exécuté dans le thread principal

  threadTask.join();
}
```

(≥ C++11)

Extrait de code 1.3 – Utilisation des *threads* POSIX en C++

pour automatiquement appeler la fonction membre `join`. Ceci a été proposé par [Voutilainen 2016], et peut facilement être implémenté. Cela ne fait pour le moment pas partie du standard, et il existe des discussions sur les risques que cette fonctionnalité peut présenter en cas de mauvais usage [Vollmann 2016].

Pour aller plus loin, les concepts de « futures » [Halstead 1985] et de « promesses » [Liskov et Shrira 1988] qui introduisent un cadre au sein de la programmation asynchrone ont été proposés. Ces outils permettent d'exécuter une tâche dans un autre *thread* (donc de manière asynchrone) et de disposer d'un mécanisme de synchronisation automatique lorsqu'une valeur que doit produire cette tâche est effectivement utilisée. L'extrait de code 1.4 présente un cas simplifié d'utilisation. La tâche produisant la valeur l'écrit dans la « promesse » tandis que le *thread* principal peut travailler pendant ce temps jusqu'à ce qu'il ait besoin de cette valeur. Il utilise alors la « future » pour l'obtenir. Si la valeur est déjà disponible, elle est immédiatement acquise et l'exécution se poursuit. Sinon, le *thread* est bloqué, en attente que la valeur soit renseignée dans la « promesse ». Ce mécanisme peut être détourné pour construire une barrière de synchronisation entre deux *threads* comme le montre l'extrait de code 1.5.

```
void task(std::promise<double> promise) {
  double result = 1.61803; // production d'un résultat
  promise.set_value(result);
}

int main() {
  std::promise<double> promise;
  std::future<double> future = promise.get_future();
  std::thread threadTask{task, std::move(promise)};

  double phi = future.get();

  threadTask.join();
}
```

(⩾ C++11)

Extrait de code 1.4 – Synchronisation par « future » et « promesse »

```
void task(std::promise<void> promise) {
  // travail à effectuer avant la barrière de synchronisation
  promise.set_value();
  // travail à effectuer après la barrière de synchronisation
}

int main() {
  std::promise<void> promise;
  std::future<void> future = promise.get_future();
  std::thread threadTask{task, std::move(promise)};

  future.get(); // synchronisation avec promise.set_value()

  threadTask.join();
}
```

(⩾ C++11)

Extrait de code 1.5 – Barrière de synchronisation par « future » et « promesse »

1.5.2 Annotations

Grâce aux outils de la section précédente, un développeur sachant écrire des programmes parallèles risque moins de commettre certaines erreurs et dispose d'outils de plus haut niveau que les primitives fournies par le standard POSIX pour synchroniser des *threads*. Cependant, l'utilisation des *threads* reste explicite.

L'objectif de la parallélisation assistée est de libérer le développeur de la charge de programmer explicitement ses *threads* (ou autres unités de parallélisme) sans pour autant laisser le système décider entièrement. Des annotations, ajoutées au code séquentiel, permettent alors au système d'obtenir des informations fiables, fournies directement par le développeur. L'API d'OpenMP [Dagum et Menon 1998] utilise des annotations, traitées par le compilateur, pour indiquer par exemple que deux séquences d'un programme sont exécutables en parallèle ou encore qu'une boucle peut voir ses itérations être parallélisées. Lorsqu'un utilisateur précède une boucle de l'annotation OpenMP `#pragma omp parallel for`, il indique en fait que la boucle ne possède aucune dépendance entre les différentes itérations. Si le développeur commet une erreur d'analyse et ajoute une annotation pour paralléliser un code ayant des dépendances, le compilateur produira alors tout de même une implémentation parallèle et le résultat de l'exécution du programme sera erroné.

Certains langages de programmation subissent plusieurs transformations entre le code source et l'exécution effective. Le dernier état d'un tel programme avant son exécution est nommé *bytecode* et est interprété par une machine virtuelle. Le *bytecode* est ainsi comparable à un langage assembleur destiné à un programme, un interpréteur, plutôt qu'à un processeur réel. Dans ces langages, les annotations peuvent être encodées dans le *bytecode* et traitées *a posteriori* de la compilation [Dittamo et al. 2007].

Le langage Cilk (et ses variantes Cilk Plus et Cilk++) [Blumofe et al. 1996], construit sur les langage C et C++, propose des annotations au moyen de directives du préprocesseur (à l'instar d'OpenMP) mais également par des mots-clés supplémentaires permettant par exemple d'indiquer au compilateur qu'une fonction peut être exécutée en parallèle.

Depuis la norme C++17, le langage C++ dispose dans sa bibliothèque standard d'un système apparenté à des annotations. Les polices d'exécution [Hoberock et al. 2013] permettent d'indiquer aux algorithmes de la bibliothèque standard (et à toute autre bibliothèque se rendant compatible) leur manière de s'exécuter. Sans annotation, l'exécution est par défaut séquentielle, mais le développeur peut ainsi demander une exécution parallèle. Cependant, bien que présent dans la norme, ce mécanisme n'est pas encore implémenté par la majorité des compilateurs.

1.5.3 Patrons parallèles

Il existe pour la programmation un ensemble de patrons classiques [Gamma et al. 1995] qui apparaissent dans de très nombreux programmes. Ceci existe de manière similaire spécifiquement pour la programmation parallèle [Mattson et al. 2005 ; McCool et al. 2012].

Le patron *fork-join* est un des plus classiques et permet l'exécution parallèle de plusieurs tâches indépendantes dont la terminaison est synchronisée. Une application récursive de ce patron permet l'implémentation parallèle d'un algorithme de type « diviser pour régner ». Ce patron est directement lié aux primitives `pthread_create` (*fork*) et `pthread_join`.

Le patron *map* permet d'exécuter une même tâche sur un ensemble de données (modèle SIMD). Le terme *farm* désigne un patron *map* appliqué pour un flux plutôt qu'un ensemble de

données. Ce patron requiert l'indépendance des différentes exécutions de la tâche.

Le patron *pipeline* correspond à un ensemble de tâches exécutées en parallèle et dont les entrées et sorties sont chaînées. La première tâche accepte ainsi en entrée un flux de données fourni de manière externe. Chaque autre tâche obtient alors son entrée à partir de la sortie de la tâche qui la précède.

D'autres patrons existent, dont des patrons permettant une exécution séquentielle, cependant l'objectif de cette section n'est pas d'en faire une liste exhaustive mais d'en présenter le principe. Des bibliothèques comme TBB (*Threading Building Blocks*) [Willhalm et Popovici 2008] proposent des structures correspondant à ces différents patrons pour être utilisées et combinées par le développeur.

L'utilisation de patrons pour masquer au développeur la complexité réelle de l'implémentation parallèle d'un programme a été introduite par [Cole 1989] au moyen de squelettes algorithmiques. Ceux-ci sont définis comme des fonctions d'ordre supérieur fournissant un patron (le squelette) dans lequel la tâche effective est encore non définie (la fonction en paramètre). Ainsi, un développeur capable de choisir un patron d'exécution adapté à son programme peut le paralléliser en utilisant le squelette correspondant.

Pour permettre l'utilisation des squelettes algorithmiques sur la plupart des algorithmes, il faut qu'il soit possible de les composer [Benoit et Cole 2005]. Leur nature de fonction d'ordre supérieur, selon l'implémentation, le permet puisqu'un squelette algorithmique est lui-même une fonction : il peut donc être utilisé comme argument d'un autre squelette.

Le chapitre 5 présente une nouvelle proposition sur les squelettes algorithmiques par rapport aux nombreuses implémentations qui ont déjà été proposées [Kuchen 2002 ; J. Falcou et al. 2006 ; Marques et al. 2013 ; Ernstsson et al. 2018]. Celle-ci s'oriente sur l'utilisation de la métaprogrammation afin de générer les codes représentés par les squelettes et propose une description intégrale de ceux-ci (plutôt que de supposer à l'avance certaines informations comme le font la plupart des implémentations de bibliothèques basées sur les squelettes algorithmiques). Grâce à cela, notre proposition permet l'intégration d'outils avancés pour, par exemple, simplifier l'utilisation de nombres pseudo-aléatoires dans un contexte parallèle.

1.6 Conclusion

Ce chapitre a présenté la notion d'exécution parallèle et de parallélisme en introduisant les notions fondamentales de ce domaine. Parmi celles-ci, on trouve notamment le concept de *thread* qui permet de créer un flux d'exécution indépendant, mais partageant une partie de la mémoire.

Ce chapitre a également traité des problèmes de synchronisation des accès aux données communes et a donc expliqué différents mécanismes tels que les sémaphores, le *mutex* ou encore les « futures ». En général, ceux-ci doivent être évités lorsque l'objectif est de maximiser les performances du programme. Ainsi ils ne seront utilisés que très ponctuellement dans les travaux présentés.

Il existe différentes techniques de parallélisation, allant de celles qui procèdent entièrement automatiquement à la conversion d'un programme séquentiel en un programme parallèle à celles qui opèrent en suivant des instructions spécifiques pour la parallélisation. Chacune possède des avantages : une parallélisation automatique demande *a priori* moins d'efforts de la part du développeur ; à l'inverse une parallélisation assistée permet un plus grand contrôle.

Que ce soit pour procéder à une parallélisation automatique ou assistée, les deux étant traités

dans cette thèse, il est nécessaire d'obtenir des informations à propos du programme. Dans le premier cas, ces informations peuvent être acquises depuis le code source. Dans le second cas, le développeur fournira une forme d'annotation permettant d'assister la parallélisation. Les chapitres suivants donnent les notions élémentaires pour ce faire, en C++ et durant la compilation.

Chapitre 2

Généricité en C++

2.1 Introduction

Un programme consiste en une suite d'instructions devant être exécutées par une machine (généralement un texte, quelque fois un diagramme, ...). L'objectif d'un programme est le traitement d'informations, et pour ce faire il manipule des données et produit un résultat (une sortie numérique, un affichage, l'envoi d'une trame sur le réseau, ...).

Généralement, pour des applications scientifiques, il s'agit de résoudre un problème spécifique. Cependant, il est souvent intéressant de généraliser le code écrit afin de résoudre une classe plus large de problèmes et ainsi fournir un code réutilisable. Pour prendre un exemple simple : s'il faut disposer d'une fonction $f(x) = x + 7$, fournir la fonction $f(x, y) = x + y$ implémente le comportement désiré pour peu que $y = 7$, mais permet par ailleurs de répondre à d'autres besoins très similaires. Ce principe est à la base de la manière de concevoir une bibliothèque, puisque l'objectif est de la rendre utilisable au sein de différents projets, mais plus généralement, il est appliqué autant qu'il peut l'être [1] car il évite une forme de répétition du code. En faisant cela, on applique une abstraction sur les données manipulées par un algorithme en les acceptant comme des paramètres qui seront remplacés par des arguments fournis à l'exécution de cet algorithme.

La généricité est un paradigme de programmation consistant en une abstraction de certaines informations dans la définition d'un algorithme ou d'une structure de donnée et permettant l'écriture de codes plus réutilisables [Musser et Stepanov 1989]. Ce concept a été implémenté de différentes manières par de nombreux langages de programmation. La généricité permet l'écriture d'un code paramétré pour générer des éléments du langage tels que des fonctions ou des classes (Java, C#, Eiffel, ...). Les abstractions possibles varient selon le langage et peuvent inclure en particulier celles de types ou de valeurs.

Le préfixe « méta » (au-delà de) est ajouté devant le nom de l'élément généré pour désigner le générateur qui décrit comment seront organisés et produits les éléments souhaités. Dans le contexte de la métaprogrammation, le mot « méta » peut être compris avec un sens de « description » et de « conformité » à une référence. Une métaclasse paramétrée décrit comment produire des classes, une métafonction décrit comment produire des fonctions.

La généricité existe en C++ au moyen des patrons. Nous utiliserons le terme anglais *template* à partir de maintenant dans ce document. Le terme patron pourra être retenu pour les patrons de conception [Gamma et al. 1995]. Comme nous l'avons signalé, une des particularités de l'implémentation proposée par le C++ est d'être entièrement résolue durant la compilation du code source. D'autres langages utilisent des techniques comme l'effacement de type (Java) ou la réification (C#) qui sont effectués à l'exécution du programme, et seulement une vérification de la validité de la paramétrisation est effectuée lors de la compilation. Que tout soit statique (évalué durant la compilation) impose des contraintes : la paramétrisation ne pourra être faite qu'avec des données connues par le compilateur durant la compilation. Mais cela apporte aussi des avantages, en particulier la possibilité d'utiliser la généricité sans surcoût en temps d'exécution par rapport à un programme n'en faisant pas l'usage.

Ce chapitre présente, sans être exhaustif, la généricité en C++. Des ressources présentant plus complètement ces aspects existent, par exemple l'ouvrage « *C++ Templates: The Complete Guide* », pour les normes de C++ avant C++11 [Vandevoorde et Josuttis 2010] et sa seconde édition pour aller jusqu'à la norme C++17 [Vandevoorde et al. 2017]. Le chapitre suivant traite de la métaprogrammation qui, en C++, repose sur des constructions permises par les templates.

1. Tant que cela ne nuit pas aux performances.

2.2 Templates

Les templates du C++ sont des fonctions, exécutées par le compilateur durant la compilation, dont le résultat (jusqu'en C++20) peut être :

— une structure ou une classe ou une union ;
— une fonction ;
— une variable (depuis C++14).

Les templates permettent ainsi d'exprimer des métaclasses, des métafonctions et des métavariables. L'exécution d'un template (c'est-à-dire la génération de l'élément du langage) est appelée instanciation. À l'instar des fonctions (classiques), qui sont évaluées à l'exécution, les templates peuvent accepter des arguments. La syntaxe générale est présentée dans l'extrait de code 2.1. Après cette ligne, il est possible d'écrire la déclaration ou la définition des différents résultats possibles d'un template.

```
template<parameter-list>
```

(⩾ C++98)

Extrait de code 2.1 – Syntaxe générale pour démarrer la déclaration ou la définition d'un template

Les paramètres de la liste sont séparés par une virgule, et indiqués entre chevrons. L'utilisation des chevrons permet de différencier les paramètres statiques, utilisés à la compilation des paramètres dynamiques, utilisés à l'exécution, qui, eux, sont entre parenthèses. Chacun des paramètres peut être :

— une valeur ;
— un type ;
— un template.

2.3 Comparaison avec le CPP

Les exemples présentés dans cette section vont dans un premier temps être suffisamment simples pour qu'il soit possible de les implémenter en utilisant le CPP (*C PreProcessor*). Or, s'il est possible dans certains cas d'obtenir un résultat similaire avec ces macros, il convient de rendre compte des nombreuses limites que celles-ci ont et qui sont levées avec les templates.

Le premier exemple qui sera présenté permettra le calcul de la somme de deux entiers. Une implémentation correcte d'une macro CPP `SUM` est présentée dans l'extrait de code 2.2.

```
#define SUM(a, b) ((a) + (b))
```

(⩾ C89)

Extrait de code 2.2 – Macro CPP `SUM` qui est remplacée par la somme de deux valeurs

Un second exemple, `MIN` (extrait de code 2.3), trouvant le minimum de deux valeurs sera également utilisé pour présenter certaines limites.

```
#define MIN(a, b) ((a) < (b)? (a) : (b))
```

(⩾ C89)

Extrait de code 2.3 – Macro CPP `MIN` qui est remplacée par la plus petite de deux valeurs

2.3.1 Faiblement typé

Les types des paramètres ou le type du résultat produit ne sont pas précisés. Ceci est dû à la manière de fonctionner du CPP : il ne traite que de chaînes de caractères et effectue des remplacements de texte. En conséquence, il est possible d'exécuter la macro `SUM` en fournissant des valeurs dont les types sont incompatibles (par inexistence d'un opérateur `+`), l'erreur n'étant relevée que lors de l'utilisation de la macro et ce dans le contexte d'utilisation.

Pour la même raison, rien ne prévient l'injection d'un code qui modifiera entièrement le comportement de la macro. L'extrait de code 2.4 montre comment changer l'opérateur `+` en opérateur `*`.

```
#define A   * ((0
#define B   0),
#define C   )

// SUM(2, 3)        : ((2) + (3))           = 2 + 3 = 5
// SUM(2 A, B 3 C)  : (( 2 * ((0+0), 3) ))  = 2 * 3 = 6
```

(⩾ C89)

Extrait de code 2.4 – Injection de code dans une macro

2.3.2 Priorité des opérations

La rigueur nécessaire lors de l'écriture est plus grande : les parenthèses semblent être une superfétation. Cependant, en les retirant, il devient assez facile d'obtenir des comportements surprenants et erronés. Si l'on retire les parenthèses intérieures, l'utilisation d'un opérateur moins prioritaire que celui employé en interne par la macro causera des résultats erronés. Dans notre exemple, les opérateurs de décalage de bit vont causer le problème. L'appel à `SUM(32 >> 1, 3)` devrait donner le nombre $\frac{32}{2} + 3 = 19$, or l'expansion de la macro donnerait `32 >> 1 + 3` avec l'opérateur `+` plus prioritaire que l'opérateur `>>`. Autrement dit, c'est équivalent à `32 >> (1 + 3)`, donnant comme résultat $\frac{32}{2^4} = 2$. L'omission des parenthèses extérieures peut produire des problèmes similaires avec cette fois-ci le contexte d'appel de la macro plutôt que la valeur de ses arguments.

2.3.3 Évaluation multiple

L'évaluation multiple des arguments est un autre problème bien connu de ces macros. En effet, lors de l'expansion de la macro, si un paramètre apparaît plusieurs fois, l'argument qui lui correspond est écrit, tel quel, plusieurs fois. Ainsi, si un argument est une expression produisant un effet de bord, l'exemple classique étant l'appel à un opérateur d'incrément `++`, celui-ci peut être exécuté plus d'une fois.

En utilisant la macro `MIN` avec l'expression `a += 2` où `a` est une variable de type `int` initialisée à 0 et l'expression `3`, c'est-à-dire en écrivant `MIN(a += 2, 3)`, le résultat retourné peut être 4 [2]. De plus, dans tous les cas, la variable « `a` » voit sa valeur augmenter de 4 au lieu de 2.

2. Le standard du C++ ne spécifie pas l'ordre d'évaluation des arguments d'une fonction. Il est donc possible de voir s'exécuter d'abord les deux incréments de la variable avant d'exécuter le test, auquel cas le résultat sera 3.

2.3.4 Utilisation

Les macros du CPP sont traitées très tôt durant le processus de compilation, phase durant laquelle toute occurrence d'une macro est remplacée par un autre texte supposé être du C++ valide. La compilation du C++ intervient ensuite et ne peut donc interagir avec les macros.

À l'inverse, le produit d'un template (étant une variable, une fonction ou un type) est utilisable au même titre que s'il avait été écrit directement. Si `SUM` et `MIN`, plutôt que des macros, sont implémentées par des fonctions (que l'on nomme respectivement `sum` et `min`), elles peuvent être utilisées comme argument d'une autre fonction. L'extrait de code 2.5 montre cela en calculant la somme des valeurs contenues dans `v` en utilisant `std::accumulate`.

```
std::accumulate(std::begin(v), std::end(v), 0, sum);
```

(⩾ C++11)

Extrait de code 2.5 – Appel d'une fonction avec comme 4ème argument une fonction

2.4 Déclaration et définition

Pour utiliser une valeur comme argument d'un template, celle-ci doit être connue au moment de la compilation. Il est par exemple possible d'écrire un template `sum` (extrait de code 2.6) qui accepte deux paramètres entiers (exemple d'instanciation : `sum<5, 7>`) et qui génère une fonction retournant la somme des valeurs données en argument.

```
template<int a, int b>
int sum() {
  return a + b;
}
```

(⩾ C++98)

Extrait de code 2.6 – Définition d'un template de fonction

L'instanciation générant une fonction, il est possible de la nommer comme dans l'extrait de code 2.7 puis d'utiliser cette fonction comme n'importe quelle autre.

```
int (*sum_5_7)() = sum<5, 7>;
std::cout << sum_5_7();
```

(⩾ C++98)

Extrait de code 2.7 – Instanciation d'un template

Dans la pratique, des appels directs (`sum<5, 7>()`) sont plus courants : si le template a déjà été instancié, le compilateur se contente de retourner le même résultat qu'à la première instanciation, cela n'implique donc aucun temps supplémentaire durant la compilation.

Il est de la même manière possible d'écrire un template de classe (extrait de code 2.8) ou de variable (extrait de code 2.9). Ces templates seront ensuite utilisables de manière similaire, leur instanciation générant respectivement une classe ou une variable. Pour le cas de la variable, il est possible d'en faire davantage une valeur plutôt qu'une variable à proprement parler comme c'est le cas de l'extrait de code 2.9 : lorsqu'une instance de `isOdd` est utilisée, elle est remplacée par une valeur directe. Cela peut être comparé à la manière de fonctionner du préprocesseur, à l'exception du typage fort qu'apporte la généricité.

```cpp
template<int capacity>
struct ArrayInt {
  int data[capacity];
};
```

(⩾ C++98)

Extrait de code 2.8 – Définition d'un template de classe

```cpp
template<int n>
constexpr bool isOdd = n&1;
```

(⩾ C++14)

Extrait de code 2.9 – Définition d'un template de variable

L'écriture d'un code générique implique très souvent l'abstraction du type de données sur lequel celui-ci s'applique. Cette abstraction est possible avec les templates puisqu'ils peuvent accepter des types comme arguments. Un exemple d'algorithme très simple qui peut fonctionner pour différents types est celui de la recherche du minimum de deux valeurs (extrait de code 2.10).

```cpp
template<typename T>
T min(T a, T b) {
  return a < b? a : b;
}
```

(⩾ C++98)

Extrait de code 2.10 – Définition d'un template de fonction `min`

Lorsque ce template est instancié (par exemple : `min<int>`), le compilateur produit une fonction, de la même manière que cela avait été fait avec `sum`. Une différence cependant : la fonction obtenue attend deux paramètres de type `int`. Pour que la fonction générée compile et donne le bon résultat, il est nécessaire d'avoir un opérateur `<` valide pour le type `T`.

2.5 Déduction d'arguments template

La déduction d'arguments template (de l'anglais TAD (*Template Argument Deduction*), à ne pas confondre avec « type abstrait de données ») permet l'instanciation d'un template de fonction en déduisant ses arguments template à partir des types des arguments passés à la fonction (qui est l'instanciation du template). Pour que cela soit possible, il faut qu'il existe un lien entre les paramètres template et les types des paramètres non template. Dans le template `min` de l'extrait de code 2.10, les arguments `a` et `b` sont indiqués de type `T`, le seul paramètre template de `min`. En écrivant `min(0, 1)` (sans chevrons), le type des arguments est `int` et la valeur de `T` est donc déduite à `int`. Dans ce cas-là, c'est donc équivalent à écrire `min<int>(0, 1)`.

Les contraintes de ce mécanisme se devinent facilement mais doivent être prises en compte. Comme dit plus haut, il faut un lien entre les paramètres template et les types des paramètres de la fonction. Ainsi, le template de l'extrait de code 2.11 ne permet pas le TAD, et le seul moyen de l'instancier est de le faire explicitement en utilisant les chevrons (par exemple : `f<char>`).

```cpp
template<typename T>
T f();
```

(⩾ C++98)

Extrait de code 2.11 – Template de fonction pour lequel le TAD n'est pas possible

Ce mécanisme requiert de plus une concordance de déduction dans le cas où un paramètre template est utilisé pour plusieurs paramètres non template. En écrivant `min(0, 1.)`, `T` sera déduit de valeur `int` à cause du premier argument, et de type `double` à cause du second. Le compilateur ne fera pas le choix et abandonnera la déduction de `T` plutôt que de forcer une conversion implicite, même si elle est possible. S'il existe un autre candidat (moins prioritaire), celui-ci valide, il sera alors utilisé, sinon la compilation échoue et le compilateur affiche tous les candidats. Dans un tel cas, pour utiliser la version template, il est possible de l'instancier explicitement et donc ne pas recourir au TAD : le compilateur peut alors utiliser les conversions implicites. Les conversions implicites sont néanmoins à éviter autant que possible et un bon compilateur lèvera une alerte en cas de perte d'information due à une telle conversion (par exemple de `double` vers `int` lorsque la partie décimale est non nulle).

Il est également possible de déduire autre chose qu'un type, que ce soit une valeur ou un template. Le principe reste fondamentalement le même, cependant il y a des limites supplémentaires lorsqu'il s'agit de déduire des valeurs. L'extrait de code 2.12 présente un cas pour lequel la déduction fonctionne.

```cpp
template<std::size_t n>
void f(std::array<char, n>);
```

(⩾ C++11)

Extrait de code 2.12 – TAD de valeur

Cependant, cet appariement est bien plus limité puisqu'il faut que le lien soit immédiat. Par exemple, un calcul comme dans l'extrait de code 2.13 empêche la déduction. Ceci reste valable même si le calcul semble immédiat (`n+1`) voire trivial (`n+0`).

```cpp
template<std::size_t n>
void f(std::array<char, 2*n>);
```

(ne compile pas)

Extrait de code 2.13 – TAD de valeur - déduction impossible

Si la déduction d'arguments template peut déterminer la totalité des arguments, elle peut aussi se limiter à un sous-ensemble de ceux-ci. Dans ce cas-là, les arguments qui n'ont pas été déduits doivent être spécifiés explicitement. L'extrait de code 2.14 présente un cas de spécification partielle (qui aurait pu être laissé entièrement déduit).

```cpp
template<typename T, typename U>
void f(T, U);

f<int>(0, 5.f); // T is specified to be int, U is deduced to be float
```

(⩾ C++98)

Extrait de code 2.14 – TAD partielle

Si un argument déduit précède un argument devant être explicité, il n'est pas possible de bénéficier de la déduction, puisque pour donner sa valeur au second argument, une valeur doit aussi être donnée au premier. L'extrait de code 2.15 montre un template avec 3 paramètres dont deux peuvent être déduits par TAD. Puisque `U` ne peut pas être déduit, il est obligatoire d'en spécifier la valeur. Pour cela, il faut dans cet exemple également spécifier `T` qui précède `U`.

```cpp
template<typename T, typename U, typename V>
void f(T, V);

// f(0, 0); // U cannot be deduced

f<int, float>(0, 0);
```

(⩾ C++98)

Extrait de code 2.15 – TAD partielle

Il est possible d'imaginer une syntaxe, utilisant par exemple `auto`, pour maintenir la déduction d'arguments template, mais cela n'existe pas à ce jour. Généralement, lorsqu'un template qui peut bénéficier du TAD requiert également des paramètres qui ne peuvent pas être déduits, afin de conserver le bénéfice du TAD, ceux-ci sont placés en premier.

Depuis C++17, il existe le CTAD (*Class Template Argument Deduction*) : le TAD peut aussi s'appliquer aux templates de classe en se basant sur des guides de déduction, explicites ou implicitement déduits des constructeurs. En simplifiant, le CTAD implicitement déduit consiste en l'utilisation de fonctions fictives (une par constructeur) ayant comme paramètres template les mêmes que ceux de la classe, suivis des éventuels paramètres template du constructeur s'il en a ; les paramètres non template quant à eux sont ceux du constructeur. Puis le TAD est utilisé sur ces fonctions fictives pour tenter de déduire les arguments template de la classe.

L'extrait de code 2.16 présente un template de classe dont le seul paramètre est un type. Un constructeur prenant un paramètre de ce type est déclaré et cela cause la création d'un guide de déduction implicite : une fonction template (dont le paramètre est nommé dans l'exemple `U` pour lever l'ambiguïté avec `T`, mais dont le nom n'importe pas) dont le type de retour spécifié permet de connaître la valeur qui doit être donnée à `T` lors d'un appel à ce constructeur.

Le second constructeur est lui-même template et prend 3 paramètres, le guide de déduction

```cpp
template<typename T>
struct A {
  A(T const&);

  template<typename E>
  A(T const&, E const&, int);
};

/* implicit deduction guides:
 *
 * template<typename U>
 * A(U) -> A<U>;
 *
 * template<typename U, typename E>
 * A(U, E, int) -> A<U>;
 */

 A(char const*) -> A<std::string>;

 A a{0};          // a: A<int>
 A b{0, "", 1};   // b: A<int>
 A c{""};         // c: A<std::string>
```

(⩾ C++17)

Extrait de code 2.16 – Guides de déduction implicites et explicites

implicite construit à partir de celui-ci prend donc lui aussi 3 paramètres et sera un template paramétré non seulement sur un type `U` (comme précédemment) mais également sur un type `E` à l'instar du constructeur. De la même manière, le type retourné permet au compilateur de déduire que c'est `U` qui doit être utilisé comme valeur pour le template de classe.

Le guide de déduction explicite indique que pour le cas particulier où le premier constructeur serait appelé avec un paramètre de type `char const*`, le type à déduire n'est pas celui-ci, mais `std::string`.

2.6 Spécialisation

La spécialisation de template est un des mécanismes les plus importants de la généricité en C++. Il permet de donner un comportement particulier pour certaines valeurs des arguments qui sont donnés au template.

Il existe deux cas principaux de spécialisation en C++, la spécialisation complète et la spécialisation partielle.

La spécialisation complète consiste en la définition d'une classe, d'une fonction ou d'une variable correspondant à l'instanciation d'un template qui fixe la totalité des arguments qui lui seront passés. Considérant l'exemple de la fonction template `min` présentée ci-avant (extrait de code 2.10), il est souhaitable d'avoir un comportement spécifique pour le cas des chaînes de caractères définies comme dans le langage C, de type `char*` : plutôt que comparer les adresses, il faut comparer ce qui est pointé. Pour ce faire, il est possible de spécialiser le template comme montré dans l'extrait de code 2.17.

```
template<>
char* min<char*>(char* a, char* b) {
  return strcmp(a, b) < 0? a : b;
}
```

(⩾ C++98)

Extrait de code 2.17 – Spécialisation totale d'un template

Dans la première ligne, `template<>` permet d'indiquer que ce qui suit est une spécialisation complète (aucune partie n'est inconnue) d'un template. Toutes les occurrences de `T` sont remplacées par la valeur spécifique correspondant à la spécialisation, et cette valeur est indiquée au niveau du nom de la fonction (`min<char*>`).

La spécialisation peut aussi être partielle (c'est-à-dire qu'il reste au moins une inconnue dans les paramètres du template), mais uniquement pour les classes template. Le template `Array` (extrait de code 2.18) permet de généraliser l'exemple `ArrayInt` (extrait de code 2.8).

```
template<typename T, int capacity>
struct Array {
  T data[capacity];
};
```

(⩾ C++98)

Extrait de code 2.18 – Template de classe `Array`

Il est possible de spécialiser pour le cas d'un tableau à 1 case, quel que soit le type. Cela se fait en utilisant une spécialisation comme le montre l'extrait de code 2.19.

```
template<typename T>
struct Array<T, 1> {
  T data;
};
```

(⩾ C++98)

Extrait de code 2.19 – Spécialisation partielle d'un template

À l'instar de la spécialisation totale, la spécialisation partielle se fait dans un contexte de définition de template, mais cette fois-ci, les chevrons ne sont pas vides puisqu'il reste au moins une inconnue : ici, le type `T`. Les paramètres indiqués à la ligne suivante indiquent au compilateur lesquels sont déterminés et lesquels sont encore inconnus, dépendants de paramètres templates.

Les spécialisations partielles peuvent être plus complexes que les spécialisations totales. Pour montrer un exemple simple, prenons le cas d'une classe semblable à `std::unique_ptr`. Le but de cette classe est de posséder un pointeur sur une zone mémoire allouée dynamiquement pour la libérer automatiquement à sa destruction (principe du RAII (*Resource Acquisition Is Initialisation*) [Stroustrup 1997, §14.4.1], quelquefois renommé RRID (*Resource Release Is Destruction*)). La partie qui va nous intéresser est le destructeur d'une telle classe. En effet, s'il est écrit comme dans l'extrait de code 2.20, il fonctionnera correctement uniquement si la mémoire a été allouée en utilisant l'opérateur `new`. Or, il existe aussi l'opérateur `new[]` pour allouer plusieurs éléments contigus. Si ce deuxième opérateur est utilisé, il est nécessaire d'utiliser l'opérateur correspondant `delete[]` (qui s'occupe de l'appel du destructeur sur chacun des éléments construits, et qui libérera correctement la mémoire : une implémentation possible de `new[]` retourne un pointeur au-delà de la zone mémoire effectivement allouée, pour conserver la quantité d'éléments comme dans la figure 2.1).

```
template<typename T>
class UniquePtr {
  T *_ptr;

  // rest of implementation
public:
  ~UniquePtr() { delete _ptr; }
};
```

(⩾ C++98)

Extrait de code 2.20 – Extrait d'une implémentation d'un template de classe `UniquePtr`

Il faut écrire une spécialisation, mais celle-ci doit être utilisée lorsque l'adresse stockée pointe sur un tableau. La spécialisation partielle dans l'extrait de code 2.21 répond au besoin. Il faut une spécialisation partielle car une part d'inconnu subsiste : le tableau peut contenir des données

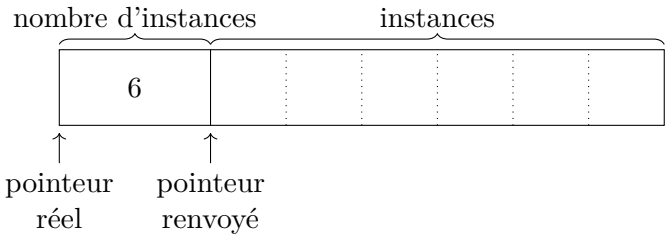


FIGURE 2.1 – Implémentation possible d'une allocation par `new[]`

```cpp
template<typename T>
class UniquePtr<T[]> {
  T *_ptr;

  // rest of implementation
public:
  ~UniquePtr() { delete[] _ptr; }
};
```

(⩾ C++98)

Extrait de code 2.21 – Spécialisation partielle d'un template

de n'importe quel type.

Lorsqu'il y a une définition primaire et au moins une définition spécialisée d'un template, le compilateur doit en choisir une lorsqu'une instanciation est demandée. Dans l'exemple avec le template de classe `UniquePtr`, il faut que le compilateur sélectionne la spécialisation dans le cas de `UniquePtr<int[]>`. C'est effectivement ce qui arrive grâce au mécanisme de sélection de la version la plus spécialisée.

La sélection du meilleur template n'est appliquée que s'il n'existe pas une version non template qui corresponde [3]. Dans ce cas, cette sélection se déroule en deux phases. Dans un premier temps, seuls les templates primaires sont considérés. Celui qui correspond, c'est-à-dire celui pour lequel les arguments sont valides, et qui est le plus restrictif sur ses arguments est choisi. Ensuite, ses spécialisations sont considérées et la version la plus spécialisée est choisie.

Cette procédure cause un comportement qui peut paraître non naturel si sa logique n'est pas connue. Dans l'extrait de code 2.22, la première occurrence de `foo` correspond à un template primaire, la deuxième à une spécialisation de ce template pour le cas `T = int*`. La dernière est un autre template primaire (une surcharge), contrainte sur des types qui sont des pointeurs. La deuxième semble correspondre parfaitement (c'est le cas), mais elle ne sera pas choisie car la sélection va d'abord chercher parmi les templates primaires lequel correspond le mieux, puis parmi ses spécialisations. Or, dans cet exemple, le template primaire (3) correspond mieux que le (1). Pour déterminer lequel correspond mieux, en simplifiant, le compilateur cherche parmi les candidats celui dont l'ensemble des arguments possibles est le plus petit [4].

```cpp
template<typename T> void foo(T);          // 1
template<>           void foo<int*>(int*); // 2
template<typename T> void foo(T*);         // 3

int main() {
  int bar;
  foo(&bar);
}
```

(⩾ C++98)

Extrait de code 2.22 – Cas de sélection de template moins intuitif

3. À l'exception de fonctions variadiques utilisant la syntaxe du langage C, à savoir de la forme `void f(...);`, `void f(int, ...);`, etc. Attention cependant, ce document traitera d'une syntaxe C++ qui y ressemble (l'utilisation des `...`) mais ne doit pas être confondue.

4. Pour une explications plus correcte, il est possible de se référer à https://en.cppreference.com/w/cpp/language/function_template.

2.7 Contraintes

La spécialisation seule permet de définir un template qui sera utilisé pour un sous-ensemble spécifié d'arguments, mais il est souvent intéressant de contraindre de manière plus spécifique les arguments que peut accepter un template.

Pour cela, le principe de SFINAE (*Substitution Failure Is Not An Error*) est une solution possible. Cet acronyme correspond à une règle du langage qui stipule que, lorsque le compilateur cherche un candidat valide parmi les templates disponibles par rapport aux arguments donnés, si la substitution des arguments aux paramètres échoue, cela n'est pas en soi une erreur. Il y aura une erreur si cela se produit pour tous les candidats. Une telle substitution peut échouer si, par exemple, il est attendu qu'un des paramètres possède un membre `B` (c'est-à-dire, étant donné `A` un paramètre, qu'il y a une occurrence de `A::B`) et que l'argument passé n'en possède pas, par exemple le type `int`.

Le terme SFINAE est maintenant utilisé pour nommer les techniques permettant de spécifier des contraintes sur les arguments d'un template et utilisant la règle du même nom [Vandevoorde et Josuttis 2010].

Pour observer ce qu'implique la SFINAE, l'extrait de code 2.23 présente un exemple.

```
template<typename T, typename = typename T::X>
void f(T);

void f(...);

struct A { using X = void; };

f(A{}); // first
f(0);   // second
```

(⩾ C++11)

Extrait de code 2.23 – Utilisation du principe de SFINAE

Deux fonctions `f` sont définies, la première accepte un argument de type *a priori* quelconque `T`. La seconde accepte n'importe quoi et en un nombre quelconque de paramètres [5]. S'il n'y avait pas la partie `, typename = typename T::X` des paramètres template de la première fonction `f`, les deux appels utiliseraient cette définition de `f`. Cette partie du code indique qu'un paramètre (un type : `typename`) est attendu, qu'il dispose d'une valeur par défaut (`= typename T::X`). Le `typename` supplémentaire sert à lever l'ambiguïté sur ce qu'est `X`, c'est-à-dire un type, car ce qu'est ce membre dépend d'une inconnue (`T`). Ce paramètre n'a, par ailleurs, pas besoin d'être nommé. L'objectif de ce code est seulement de faire apparaître `T::X` : ceci déclenchera un échec de substitution de `T` si l'argument correspond ne possède pas de type membre nommé `X`. Cet échec de substitution, grâce à la SFINAE, n'interrompt pas la compilation, mais force le compilateur à rejeter le candidat. Tout ce qui entoure `T::X` est nécessaire uniquement pour correspondre à la grammaire du langage et garantir que le compilateur ne va pas ignorer cette contrainte. Ainsi, si le premier appel à `f` dans le code ci-dessus va pouvoir utiliser la première définition de cette fonction, le second quant à lui va causer un échec de substitution (`int::X` est invalide puisque le type `int` n'a pas de type membre `X`) : le compilateur va éliminer ce candidat et voir s'il y a

5. Une syntaxe C++ existe et sera abordée plus tard dans le document. Pour cet exemple, la syntaxe héritée du C suffit.

d'autres fonctions `f` possibles, ici c'est le cas, donc aucune erreur de compilation n'est émise, et la deuxième définition de la fonction est utilisée.

Cette syntaxe atteint vite ses limites. En définissant deux fonctions `f` ayant les mêmes paramètres, l'une pour tout `T` ayant un type membre `X` et l'autre un type membre `Y` de la même manière que dans l'exemple précédent, le code ressemblera à l'extrait de code 2.24. Or, ce code ne peut pas compiler car le compilateur ne peut distinguer les deux définitions de `f`.

```
template<typename T, typename = typename T::X>
void f(T);

template<typename T, typename = typename T::Y>
void f(T);
```

(ne compile pas)

Extrait de code 2.24 – Tentative d'utilisation de SFINAE ne compilant pas

Pour parvenir à notre objectif, il est possible de rendre la liste de paramètres template différente pour chaque template. L'extrait de code 2.25 présente une des manières de procéder. Ce code permet de sélectionner `f` selon que le type `T` possède un type membre `X` ou `Y`, et d'échouer à compiler s'il n'en possède aucun des deux. La compilation échouera aussi si le type possède les deux membres à cause d'une ambiguïté entre les deux versions de la fonction qui sont alors aussi valables l'une que l'autre. Si ce comportement n'est pas souhaité, il peut être évité, mais ce n'est pas le propos pour le moment. Plutôt que d'utiliser un paramètre ayant une valeur par défaut dépendant de `T`, il faut qu'il dépende directement de cet inconnu `T`. Pour cela, il suffit de créer un paramètre attendant une valeur et dont le type dépend de `T`. Même si après évaluation ce type se révèle être systématiquement le même, le compilateur ne cherche pas à vérifier cela et considère les différentes versions de `f`. Le fait de donner une valeur par défaut à ce paramètre permet d'en cacher l'existence à l'utilisateur. L'utilisation d'un pointeur permet par ailleurs à cette astuce de fonctionner indépendamment du type de `X` ou `Y`.

```
template<typename T, typename T::X* = nullptr>
void f(T);

template<typename T, typename T::Y* = nullptr>
void f(T);
```

(⩾ C++11)

Extrait de code 2.25 – Contraintes différentes sur une même fonction

D'autres techniques, utilisant le type de retour ou des paramètres non template de la fonction (avec une valeur par défaut) existent, mais elles ont l'inconvénient de mélanger la fonction et le template, ce que la technique présentée ci-dessus évite.

Nous venons de voir une méthode pour tester la présence d'un type membre, mais s'il s'agit de tester, par exemple, la parité d'un argument template, cela ne suffit pas. Pour ce cas, il est possible d'écrire, grâce à la spécialisation de template, un outil pour transformer un test en contrainte par SFINAE. L'idée consiste à fournir un template de classe primaire ayant un booléen en paramètre et aucun membre, et le spécialiser pour le cas où le booléen est vrai pour fournir un membre. Ce concept a été implémenté pour la première fois par une bibliothèque de Boost, au nom de `enable_if`. Une implémentation minimale peut ressembler à l'extrait de code 2.26 et s'utiliser comme montré dans l'extrait de code 2.27.

```cpp
template<bool>
struct EnableIf {};

template<>
struct EnableIf<true> {
  using type = void;
};
```

(⩾ C++11)

Extrait de code 2.26 – Implémentation minimale d'un « enable if »

```cpp
template<int i, typename EnableIf<i%2 == 0>::type* = nullptr>
void f();

template<int i, typename EnableIf<not (i%2 == 0)>::type* = nullptr>
void f();
```

(⩾ C++11)

Extrait de code 2.27 – SFINAE sur la parité d'un argument

Si le test `i%2 == 0` est évalué à faux, ce sera `EnableIf` sans type membre qui sera utilisé, faisant échouer la substitution. Au contraire, s'il est évalué à vrai, le type membre existera et la fonction sera instanciée. Dans cet exemple, les conditions d'activation des templates sont exclusives et couvrent tous les cas, donc pour tout argument utilisé, l'une des deux `f` sera valide. Depuis C++11, la bibliothèque standard inclut une implémentation de `EnableIf` : `std::enable_if`.

2.8 Concepts

Même s'il est beaucoup utilisé pour cela, le mécanisme de SFINAE n'a pas été créé avec pour objectif la spécification de contraintes. La syntaxe exposée par ce mécanisme n'est donc pas vraiment adaptée.

En concevant un mécanisme pensé pour cette tâche, il est facile d'écrire des contraintes similaires avec une meilleure lisibilité, et éventuellement une plus grande expressivité. La programmation générique définit pour cela les *concepts* [Siek et al. 2005 ; Stroustrup 2017], correspondant à un ensemble d'exigences devant être respectées, que ce soit syntaxiques, sémantiques, ou sur d'autres critères tels que la complexité temporelle ou spatiale. Plusieurs implémentations, plus ou moins complètes, ont été proposées. Boost fournit une bibliothèque [Siek et Lumsdaine 2000] permettant la vérification de concepts, c'est-à-dire la partie validation syntaxique du respect de contraintes spécifiées. Une autre implémentation partielle [Bachelet et Yon 2017] s'est concentrée sur l'aspect raffinement et sélection du template du meilleur concept, c'est-à-dire le plus raffiné de ceux qui sont respectés. Cette implémentation est aussi une bibliothèque écrite en C++ et utilise un système de tag pour la sélection du template. À l'inverse de la bibliothèque de Boost, le respect d'un concept est cette fois-ci déclaratif et déclaré après le type. Outre les bibliothèques, il est possible de fournir cette fonctionnalité directement au niveau du langage par le biais du compilateur comme cela a été fait pour Clang [Voufo et al. 2011] et GCC [6] depuis sa version 6, et comme cela sera le cas pour tout compilateur respectant le standard C++20 puisque les concepts y ont été intégrés [ISO et C++ committee 2020, §13.7.8 et §18]. Pour être exact, les concepts

6. https://gcc.gnu.org/projects/cxx-status.html

intégrés sont une sous partie de ce qu'ils auraient pu être, se limitant aux aspects statiques, et excluant donc la vérification sémantique par exemple.

Les concepts du C++20 permettent ainsi la spécification de contraintes devant être respectées par un argument donné à un template afin que le candidat soit considéré. Si on considère cet exemple présenté dans l'extrait de code 2.28, il est possible de remplacer l'utilisation de la SFINAE par un `requires` comme dans la nouvelle définition de `f` de l'extrait de code 2.29.

```cpp
#include <type_traits>

struct A {};
struct B: A {};

template<typename T, std::enable_if_t<std::is_base_of_v<A, T>>* = nullptr>
void f(T const&);

int main() {
    f(A{}); // ok
    f(B{}); // ok
    f(0); // error
}
```

(⩾ C++17)

Extrait de code 2.28 – Utilisation de la SFINAE - contrainte d'héritage

```cpp
template<typename T> requires(std::is_base_of_v<A, T>)
void f(T const&);
```

(⩾ C++20)

Extrait de code 2.29 – Utilisation de `requires` - contrainte d'héritage

Plusieurs avantages sont à observer. Le premier est qu'il n'y a plus de recours à un outil fourni par une bibliothèque puisque le nouveau mécanisme existe directement au niveau du langage. Ensuite, les rôles étant mieux distingués, le contournement de la contrainte est rendu impossible. Enfin, cela permet aussi de disposer d'informations bien plus précises en cas d'erreur. En utilisant le même compilateur, GCC 8.3, la différence entre les diagnostics que l'on obtient, en utilisant soit la SFINAE (extrait de code 2.30) soit les contraintes (extrait de code 2.31), est intéressante. Dans le premier cas, bien que l'information utile soit présente, elle n'est pas mise en valeur. Dans le second cas, au contraire, le problème est mieux indiqué et son origine est explicitée.

Par ailleurs, il est possible de nommer un ensemble de contraintes sous la forme d'un concept et ainsi l'utiliser plutôt que les contraintes qu'il représente. L'extrait de code 2.32 montre comment

```
main.cpp: In function ‘int main()’:
main.cpp:12:8: error: no matching function for call to ‘f(int)’
     f(0); // error
        ^
main.cpp:7:6: note: candidate: ‘template<class T, std::enable_if_t<is_base_of_v<A,
↪  T> >* <anonymous> > void f(const T&)’
 void f(T const&);
      ^
main.cpp:7:6: note:   template argument deduction/substitution failed:
```

(g++ 8.3)

Extrait de code 2.30 – Diagnostic de GCC 8.3 en l'absence de candidat valide par SFINAE

```
main.cpp: In function ‘int main()’:
main.cpp:12:8: error: cannot call function ‘void f(const T&) [with T = int]’
     f(0); // error
        ^
main.cpp:7:6: note:   constraints not satisfied
 void f(T const&);
      ^
main.cpp:7:6: note: ‘is_base_of_v<A, T>’ evaluated to false
```

(g++ 8.3)

Extrait de code 2.31 – Diagnostic de GCC 8.3 en cas de contrainte non satisfaite

il est possible de nommer la contrainte « hérite de A » (ce qui n'est pas nécessairement pertinent à faire, mais sert ici d'exemple).

```
template<typename T>
concept InheritsFromA = std::is_base_of_v<A, T>;
```

(⩾ C++20)

Extrait de code 2.32 – Définition d'un concept

Un concept est défini comme un prédicat booléen à vérifier, lequel peut utiliser des conjonctions et des disjonctions (et logique et ou logique) pour construire une expression plus complexe. Le raffinement de concept se fait alors en définissant un nouveau concept comme étant la conjonction ou disjonction d'un ou plusieurs concepts existants et éventuellement d'autres contraintes spécifiques. Enfin, le mot-clé **requires** permet, lors de la définition d'un concept, d'établir des contraintes syntaxiques.

Ainsi, si le type doit avoir un constructeur par défaut, supporter les comparaisons d'égalité et hériter de `A`, le concept peut être défini comme présenté dans l'extrait de code 2.33. Requérir un concept peut être fait de la même manière qu'une contrainte non nommée, en utilisant le mot-clé **requires** (extrait de code 2.34). Il est également possible d'utiliser une syntaxe plus légère où le mot-clé **typename** est remplacé par le nom du concept (extrait de code 2.35).

```
template<typename T>
concept EqComparableA = InheritsFromA<T> and requires(T a, T b) {
  T();
  { a == b } -> bool;
  { a != b } -> bool;
};
```

(⩾ C++20)

Extrait de code 2.33 – Implémentation du « concept » `EqComparableA`

```
template<typename T> requires(EqComparableA<T>)
void f(T const&);
```

(⩾ C++20)

Extrait de code 2.34 – Utilisation d'un concept avec **requires**

```
template<EqComparableA T>
void f(T const&);
```

(⩾ C++20)

Extrait de code 2.35 – Exemple d'utilisation d'un concept

L'exemple de l'extrait de code 2.33 utilise le raffinement de concept : il faut respecter le concept `InheritsFromA` pour respecter `EqComparableA`. Le raffinement peut être comparé à l'héritage de la POO (Programmation Orientée Objet), avec davantage de libertés (par exemple, cela n'est pas limité à la conjonction logique). Émuler cela avec la SFINAE est possible, cependant, le raffinement est plus puissant lorsqu'il s'agit de la résolution d'une surcharge de fonction contrainte par exemple. Si l'on a une surcharge contrainte par `InheritsFromA` et une autre par `EqComparableA`, un argument satisfaisant le second va faire sélectionner la seconde surcharge. Si cela paraît évident, obtenir ce comportement avec la SFINAE est difficile : en procédant de la même manière que présenté dans ce document, les deux surcharges seront acceptables et la compilation échouera pour cause d'ambiguïté.

2.9 Instanciation

Le mécanisme d'instanciation des templates est une étape importante du fonctionnement de la généricité telle qu'elle est implémentée en C++. En particulier dans le développement de métaprogrammes pour lesquels certaines spécificités du mécanisme d'instanciation sont nécessaires. Le principe de base ressemble beaucoup au fonctionnement du pré-processeur du langage C : chaque occurrence d'une variable template est remplacée par sa valeur. Cependant, l'instanciation d'un template diffère sur certains aspects qui sont particulièrement importants. Un de ces aspects est la validation syntaxique du code, vérifiée au moment de l'instanciation et non de l'utilisation. Le chapitre suivant détaillera davantage l'instanciation des templates.

2.10 Mécanisme de résolution de surcharges

Outre les templates, le C++ permet la généricité par la surcharge de fonctions. La surcharge de fonctions consiste en l'écriture de plusieurs fonctions portant le même nom mais n'acceptant pas strictement les mêmes paramètres. Lors de l'appel, plusieurs candidats sont possibles et le compilateur doit donc déterminer lequel utiliser, s'il en existe un unique qui corresponde. Pour cela, il considère les arguments utilisés (leur nombre et type) et vérifie si une fonction peut les accepter. Différentes règles s'appliquent selon que le candidat est un template ou non : dans le premier cas, une conversion implicite ne sera pas permise pour adapter un argument dont le type déduit entre en conflit avec un type déjà déduit (section 2.5). De plus, tout candidat template sera moins prioritaire qu'un candidat non template. Si plusieurs candidats de même priorité correspondent, il y a ambiguïté et une erreur survient.

2.11 Inférence de type

Pour expliquer l'intérêt de l'inférence de type, implémentons une fonction d'addition générique. L'extrait de code 2.36 propose une première version, acceptant deux paramètres de type `T` et retournant donc une valeur de ce type[7]. À l'appel, pour que le type `T` soit déduit, il faut que les paramètres `lhs` et `rhs` soient de même type, sans possibilité de conversion implicite. Si l'on veut pouvoir additionner des instances de types distincts, il faut changer la manière de définir cette fonction.

7. En réalité, ce type de retour peut déjà être un problème si l'on souhaite gérer le dépassement de valeur des types numériques.

```
template<typename T>
T addition(T lhs, T rhs) {
  return lhs + rhs;
}
```

(⩾ C++98)

Extrait de code 2.36 – Addition template (première version)

Il convient pour cela de prendre deux paramètres de deux types, `LHS` et `RHS`. Un problème est alors soulevé : le type de retour n'est plus aussi simple que dans l'extrait de code 2.36 car les opérandes de l'addition peuvent avoir des types différents. Il existe des techniques sans inférence de type par le langage, mais elles requièrent davantage de code et ne sont pas l'objet de cette section. Pour résoudre ce problème, il est possible de laisser le compilateur décider du type qu'il convient de retourner (extrait de code 2.37).

```
template<typename LHS, typename RHS>
auto addition(LHS lhs, RHS rhs) {
  return lhs + rhs;
}
```

(⩾ C++14)

Extrait de code 2.37 – Addition template (deuxième version)

L'usage du mot-clé `auto`, introduit en C++11, permet l'inférence statique d'un type à partir de la valeur utilisée pour initialiser la variable (dans le cas de l'exemple : le retour de la fonction). Les règles d'inférence pour `auto` font que le type déduit sera celui de la valeur qui va être stockée, sans qualificatif (`const` et `volatile`) ni référence[8]. Cela permet de choisir d'ajouter ou non ces modificateurs de type selon le besoin.

Si, au contraire, le type exact de déclaration de l'expression est souhaité, il faut utiliser `decltype` avec l'expression pour argument. On pourra donc par exemple écrire pour une expression `a` : `decltype(a)`. Ce dernier possède cependant une particularité syntaxique : `decltype((a))` ne correspond pas au type de `a` mais à une référence sur ce type[9].

Il est possible de combiner l'utilisation de `decltype` et de `auto`, le premier pour déterminer le type exact, et le second comme remplaçant de l'expression qui servira à initialiser la variable. Il est ainsi possible de déclarer une variable de type `decltype(auto)`, ou d'utiliser cela comme type de retour pour une fonction. À noter que, dans le cas du retour de fonction, la règle de `decltype` sur les parenthèses s'applique également, ainsi `return var;` et `return (var);` ne sont pas équivalents si le type de retour est `decltype(auto)` : le second retournera systématiquement une référence sur la variable tandis que le premier retournera une variable dont le type sera exactement celui spécifié.

Ce mécanisme d'inférence de type est utile en général, mais il devient pratiquement indispensable à la métaprogrammation template : les types manipulés peuvent rapidement devenir difficiles à écrire, et devoir le faire rendrait considérablement plus lourd l'utilisation de n'importe quel métaprogramme. À vrai dire, certains usages de la métaprogrammation template reposent sur la génération de types que l'utilisateur ne peut donc pas écrire lui-même sans annuler l'intérêt du métaprogramme. Des exemples de tels cas sont présents dans le chapitre suivant.

8. Si le type est `int const*`, ce qualificatif est conservé : sont retirés ceux qui s'appliquent après toute indirection.

9. Si `a` est déjà une référence, `decltype((a))` correspond alors à son type exact.

2.12 Transmission parfaite

Il n'est pas rare, à l'écriture de fonctions en général, d'avoir à traiter de multiples cas selon les modificateurs appliqués aux types, inconnus, que ce soit des qualificatifs, des références ou des références sur des r-values (des références sur une donnée expirant et dont le contenu peut être modifié, permettant d'éviter des copies profondes en faveur de copies superficielles). Lorsque cela arrive sur une fonction unaire, cela fait jusqu'à 8 variantes à écrire, et la croissance est exponentielle en le nombre de paramètres pour lesquels les différentes possibilités sont gérées (8^n, où n est le nombre de paramètres à gérer).

En passant par un paramètre générique, il est possible de réduire ces différentes variantes en une seule qui s'adaptera selon le type réel. La transmission parfaite du type (en anglais, *perfect forwarding*) utilise des règles dites de *reference collapsing* [Dimov et al. 2002] qui se comportent comme dans l'extrait de code 2.38 et résumées par le tableau 2.1.

```
using LRef = int&;
using RRef = int&&;
int n;

LRef&  r1 = n; // decltype(r1): int&
LRef&& r2 = n; // decltype(r2): int&
RRef&  r3 = n; // decltype(r3): int&
RRef&& r4 = 0; // decltype(r4): int&&
```

(⩾ C++11)

Extrait de code 2.38 – Règles de reference collapsing [cppreference 2011]

Pour transmettre un paramètre en conservant son type, et donc appeler la bonne fonction en cas de surcharge, la bibliothèque standard fournit un outil qui s'occupe de la conversion : `std::forward`. L'extrait de code 2.39 est un exemple d'utilisation de la référence de transmission (en anglais, *forwarding reference*) avec `std::forward`. Dans cet exemple, comme `T` est un paramètre template, `T&&` ne représente pas une référence sur une r-value mais une référence de transmission. Il est important de noter que si l'on écrit quelque chose comme `std::vector<T>&&`, ce qui est à gauche des esperluettes n'étant pas directement un paramètre template (même s'il en dépend), cela correspond à une référence sur une r-value et non plus à une référence de transmission.

```
template<typename T>
void callF(T&& arg) {
  f(std::forward<T>(arg));
}
```

(⩾ C++11)

Extrait de code 2.39 – Exemple de transmission parfaite

type	référence ajoutée	résultat
&	&	&
&	&&	&
&&	&	&
&&	&&	&&

Table 2.1 – Table des règles de *reference collapsing*

2.13 Packs de paramètres

Depuis la norme C++11, il est possible de prendre un pack de paramètres de même nature, c'est-à-dire exclusivement des types, ou exclusivement des valeurs ou exclusivement des templates attendant les mêmes paramètres. Le but de cette section n'est pas de présenter tout ce qu'il est possible de faire avec cet outil, mais d'en expliquer succinctement le fonctionnement. Il sera employé par la suite, et de manière plus complète que présenté ici.

Les packs de paramètres permettent entre autres l'implémentation de fonctions variadiques typées. La différence la plus importante avec les fonctions variadiques, comme il est possible de les implémenter en C, est que le type de chaque paramètre est bien connu au moment de la compilation. Cela signifie qu'il n'est plus nécessaire d'avoir recours à une astuce pour retrouver le type comme le fait par exemple `printf`, et donc, dès la compilation, la bonne surcharge d'une fonction appelée sur un paramètre peut être sélectionnée.

L'exemple présenté dans l'extrait de code 2.40 montre l'utilisation d'un pack de paramètres. La fonction `add` calcule la somme de tous ses paramètres. Cette fonction est récursive car il faut isoler un paramètre du pack afin de pouvoir l'utiliser spécifiquement. Des techniques permettent d'éviter cette récursivité, elles seront présentées ultérieurement. Un pack ne peut être utilisé tel quel : il doit être étendu, et c'est à cela que servent les 3 points « `...` ».

L'expansion du pack de paramètres template `Ts` sert à faire correspondre à chaque paramètre template un argument de la fonction `add` lors de son appel. Ces arguments sont eux-mêmes regroupés dans le pack `values`. Dans l'exemple, l'expansion du pack `values` permet d'appeler la fonction `add` avec tous les paramètres sauf le premier. Un pack de paramètres peut être vide, et lorsque cela surviendra, la surcharge de `add` n'acceptant qu'un paramètre sera appelée et terminera la récursion.

Si cette fonction est appelée avec une série d'entiers comme arguments, leur somme sera retournée, tandis qu'avec une série de chaînes de caractères, c'est la concaténation de celles-ci qui sera construite, parce que les types des paramètres sont connus.

```
template<typename T, typename... Ts>
auto add(T value, Ts... values) {
  return value + add(values...);
}

template<typename T>
auto add(T value) {
  return value;
}
```

(≥ C++14)

Extrait de code 2.40 – Calcul de la somme d'un nombre quelconque de valeurs

La suite de cette section va s'attarder sur quelques détails de fonctionnement de l'expansion de pack. En C++17, il est devenu possible de choisir quel opérateur binaire sépare les éléments d'un pack lors de son expansion à la place de la virgule avec les notations `(... OP pack)` et `(pack OP ...)` (la position du pack par rapport à l'opérateur change l'associativité). Cela inclut l'opérateur virgule, mais ce n'est pas équivalent à la syntaxe `pack...` qui n'utilise pas l'opérateur virgule mais une virgule servant à séparer des arguments d'une fonction. L'exemple présenté dans l'extrait de code 2.41 montre une utilisation de ce mécanisme. L'expansion du pack se fera ainsi : `(values0+(values1+(values2+...)))`. L'opérateur `sizeof...` retourne le nombre d'éléments

```
template<typename... Ts>
auto avg(Ts... values) {
  return (values + ...) / sizeof...(Ts);
}
```

(⩾ C++14)

Extrait de code 2.41 – Calcul de la moyenne d'un nombre quelconque de valeurs

contenu dans un pack.

Le pack peut faire parti d'un motif (voir [ISO et C++ committee 2011, §14.5.3.4]), auquel cas l'expansion applique ce motif sur chacun des éléments du pack. Par exemple `f(pack)...` sera étendu `f(p0), f(p1), f(p2), ...`. Ceci est très utilisé pour les paramètres d'une fonction. Ainsi, pour la fonction suivante : `void f(Ts const&... args)`, les éléments du pack `args` sont tous une référence (`&`) sur un type T_i quelconque constant (`const`).

2.14 if constexpr

Il arrive fréquemment, lors de l'écriture d'un programme générique, d'avoir à effectuer un traitement qui dépende des données en entrée comme pour le cas d'un programme classique. Cependant, il y a une différence importante : dans un programme classique, il est possible d'utiliser un simple branchement (`if`) pour sélectionner le comportement désiré. Par exemple, les entiers ne permettant pas de diviser par 0, un programme divisant par une variable pourra tester sa valeur et ne pas exécuter la division le cas échéant. Dans le cas du programme générique, il est possible de dépendre d'un type et selon la valeur de ce dernier, certaines opérations peuvent être invalides du point de vue du compilateur (par exemple l'opérateur d'indirection ne peut être utilisé sur une instance de type `int`), or, toutes les branches d'un `if` sont effectivement compilées.

Si l'on prend un exemple de fonction dont l'objectif est de retourner la valeur pointée dans le cas où l'argument est une adresse, sinon la valeur elle-même, il est possible de l'écrire en se reposant sur la surcharge de fonction et le fait que la version la plus contrainte va être utilisée. Deux versions sont à écrire : le cas général, et une surcharge acceptant un pointeur (extrait de code 2.42).

```
template<typename T>
T getValue(T t) {
  return t;
}

template<typename T>
T getValue(T* t) {
  return *t;
}
```

(⩾ C++98)

Extrait de code 2.42 – Indirection conditionnelle avant C++17

Depuis C++17, il existe `if constexpr`, grâce auquel il est possible de n'écrire qu'une fonction, laquelle se scinde en deux branches selon que l'argument est vrai ou faux. La différence avec le `if` classique est que la branche que le programme n'exécute pas est entièrement supprimée du programme : cela ne pose aucun problème qu'elle soit erronée. Dans l'extrait de code 2.43, si `T`

n'est pas un pointeur, en utilisant un `if`, l'appel à l'indirection cause une erreur de compilation [10].

```
template<typename T>
auto getValue(T t) {
  if constexpr(std::is_pointer_v<T>)
    return *t;
  else
    return t;
}
```

(⩾ C++17)

Extrait de code 2.43 – Indirection conditionnelle avec `if constexpr`

2.15 Conclusion

La généricité est un concept omniprésent en informatique et permet l'écriture de bibliothèques dont les fonctions peuvent être utilisées dans des projets variés. Au-delà de ce principe de base, cela s'étend également à la généralisation de types dans certains langages ayant le paradigme « orienté objet ».

En C++, le mécanisme des « templates » répond à ce besoin en permettant de paramétrer un type, une fonction ou encore une variable. Ceux-ci peuvent alors dépendre de valeurs ou de types. Une particularité des templates est d'être entièrement résolu au moment de la compilation. Ce faisant, ils permettent une forme d'abstraction sans nécessairement causer d'augmentation du temps d'exécution.

Ce chapitre a présenté différents concepts de la généricité en C++. La métaprogrammation dite « template », qui est le sujet du chapitre suivant, est essentiellement construite à partir des templates et de leurs spécificités. Pour cette raison, ce chapitre a principalement introduit les éléments qui sont nécessaires à cette métaprogrammation, notamment la spécialisation des templates, mais aussi des fonctionnalités qui, bien que dispensables, sont incontournables, par exemple les packs.

10. Sauf si `T` représente un type qui implémente l'opérateur d'indirection sans pour autant avoir la sémantique de pointeur, mais ce n'est pas le propos ici.

Chapitre 3

Métaprogrammation template

3.1 Introduction

Dans le cadre de cette thèse, la métaprogrammation est utilisée pour répondre au besoin de traiter de manière la plus générique possible un ensemble défini de problèmes sans que les conséquences sur les performances obtenues ne soient trop pénalisantes. Afin de faciliter la compréhension par le lecteur des travaux présentés, ce chapitre présente la métaprogrammation et en particulier celle utilisée dans ce cadre.

On parle de métaprogramme lorsqu'un programme traite ou produit des données représentant un programme. Le premier langage à proposer un mécanisme permettant la métaprogrammation était un langage de la famille Lisp. [Sheard 2001] distingue deux catégories de métaprogrammes : d'une part les générateurs de programmes, et d'autre part les analyseurs de programmes qui traitent de données représentant un programme et fournissent une sortie qui n'en est pas un. Un exemple typique d'analyseur est un programme de preuve [C. A. R. Hoare 1971], tandis que le compilateur correspond à un générateur de programmes [Aho et al. 1986], que ce soit en transformant un code source d'un langage en code source d'un autre langage (voire du même langage après transformation) ou encore en fichier de pseudo-code ou binaire, ce qui est techniquement juste un autre langage. C'est cette dernière catégorie, la génération de programme, qui nous intéressera davantage.

Le langage dans lequel le métaprogramme est écrit se nomme un métalangage, et celui qui est traité comme une donnée est appelé le langage « objet » (en anglais, *object language*). Si les deux langages sont identiques, il s'agit alors de métaprogrammation homogène, sinon de métaprogrammation hétérogène.

Dans un premier temps, les différents types de métaprogrammation sont présentés. Ensuite, le concept de généricité est détaillé afin de préparer le lecteur au prochain chapitre qui en fait un usage extensif. Enfin, la métaprogrammation template, qui est un cas spécifique de métaprogrammation, est présentée et expliquée en étudiant différents cas.

3.2 Types de métaprogrammation

Il est possible de classer les différents types de métaprogrammation selon de nombreux axes distincts. Selon [Sheard 2001], il est possible de distinguer principalement les analyseurs et les générateurs.

Les analyseurs traitent des données représentant d'autres programmes sans nécessairement produire de telles données. Dans cette catégorie se trouvent de nombreux outils, du debugger aux programmes de preuve [Binkley 2007] qui produisent souvent des graphes très utiles aux vérifications et aux optimisations. Si certaines techniques qui sont employées par les analyseurs sont intéressantes, elles sont également utilisées avec les générateurs, et c'est plutôt cette dernière catégorie qui nous concerne.

Les générateurs produisent des données représentant un programme. Ainsi, un métaprogramme qui écrit dans un fichier le code source d'un programme affichant une chaîne de caractères spécifique qui aura été donnée en paramètre du métaprogramme est un exemple très simple de générateur. L'extrait de code 3.1 est une implémentation possible en C d'un tel métaprogramme.

Il s'agit ici de génération de texte : le métalangage utilisé (le langage C) ne fournit pas d'outil permettant de générer du langage « objet » (ici, également du C) de manière plus élaborée. Cette manière d'opérer possède des inconvénients : il n'est pas possible de vérifier la validité (ne

```c
#include <stdio.h>

int main(int argc, char **argv) {
  FILE *output = fopen(argv[1], "w");
  if(!output) return 1;
  (void)argc;

  fputs("#include <stdio.h>\n\n", output);
  fputs("int main() {\n", output);
  fprintf(output, "\tputs(\"%s\");\n", argv[2]);
  fputs("}", output);

  fclose(output);

  return 0;
}
```

(⩾ C89)

Extrait de code 3.1 – Métaprogramme en C générant un code C

serait-ce que syntaxique) du programme généré au niveau du métaprogramme. Cela signifie qu'il est nécessaire de compiler le fichier source généré pour le vérifier.

Cette section est inspirée des travaux de [Lilis et Savidis 2019] et présente différents types de métaprogrammation.

3.2.1 Macros

Le langage C permet la métaprogrammation au moyen de macros [Kernighan et Ritchie 1988]. Celles-ci sont en réalité traitées par le préprocesseur du langage C, CPP (*C PreProcessor*). L'exécution de ces macros est faite avant la compilation du code C. Il s'agit principalement de remplacement simple de texte, où `#define A B` instruit le CPP qu'il faut ensuite remplacer toute occurrence de `A` par `B`, `B` pouvant être lui-même une autre définition, aussi longtemps qu'aucun cycle n'est induit par la séquence de remplacements. Ces remplacements de texte peuvent prendre la forme de « fonctions » en acceptant des paramètres (traités comme du texte eux aussi) qui peuvent être utilisés, par exemple `#define PRINT_TWICE(T) puts(T); puts(T)` qui affiche deux fois son argument. Il existe, depuis 2011, un moyen de sélectionner le texte à produire en fonction du type d'un argument d'une macro, permettant un certain niveau de généricité qui étend la capacité du langage à l'écriture de code générique grâce au mot-clé `_Generic`. L'extrait de code 3.2 montre une utilisation de `_Generic`. Dans cet exemple, l'utilisation de la macro `display` va causer un appel à différentes fonctions selon le type de l'argument donné pour le paramètre `X`. Cette fonctionnalité ne permet, durant la compilation, qu'une simple association entre le type d'une expression (ici `X`) et une expression à utiliser : il n'est donc pas possible d'utiliser à la place une valeur par exemple. Par ailleurs, étant implémentée par le CPP, la substitution est faite en amont de la compilation et ne permet pas l'utilisation de la macro `display` au même titre qu'une fonction qui peut être utilisée comme argument d'une autre.

Dans les langages Lisp il existe également un système de macros, mais celui-ci est différent de ce que CPP propose puisque celles-ci permettent de traiter les données non comme du texte mais comme du code. À l'instar des macros CPP, ces macros sont exécutées avant l'évaluation de l'AST (*Abstract Syntax Tree*).

Le Lisp est homoiconique, les listes de données permettant de représenter une expression

```c
#include <stdio.h>

void displayi(int i)    { printf("%d\n", i); }
void displayf(float f)  { printf("%f\n", f); }
void displays(char *s)  { puts(s); }

#define display(X) \
  _Generic((X), int: displayi, float: displayf, char*: displays) \
  (X)

int main() {
  display("Hello World!");
  return 0;
}
```

(⩾ C11)

Extrait de code 3.2 – Utilisation de `_Generic` en C

fonctionnelle si le premier élément est une fonction (les éléments suivants devenant les opérandes de celle-ci). Cette particularité facilite la manipulation des fonctions en Lisp au travers des macros, et permet de faire de la métaprogrammation homogène, puisque les macros Lisp peuvent résulter en un AST comprenant des macros à évaluer.

Un dialecte de Lisp, le Scheme, complète les fonctionnalités des macros Lisp en les rendant « hygiéniques » : c'est-à-dire que leur résolution/expansion ne peut pas causer de confusion entre un identifiant interne à la macro et un identifiant externe à celle-ci.

3.2.2 Réflexion

Dans le contexte de la programmation informatique, la réflexion représente la possibilité d'un programme d'agir sur lui-même. Il s'agit d'introspection lorsque le programme est capable de s'examiner, en lecture seule. À l'inverse, lorsqu'il est capable de se modifier, il s'agit d'intercession. Deux types de réflexion sont également distingués, indépendamment de l'introspection et l'intercession. La réflexion structurelle permet l'accès à la structure du programme : les types, les fonctions, ... La réflexion comportementale permet l'accès aux éléments dynamiques du programme : l'instanciation ou l'accès aux variables, les appels de fonctions...

Le Java est un exemple de langage capable d'introspection structurelle ainsi que d'introspection et intercession comportementale, tous deux durant l'exécution du programme. Le support de la réflexion repose sur les métaclasses. Couplé à l'utilisation de la compilation à la volée qui permet une métaprogrammation par génération de texte, il est possible de faire de la métaprogrammation dynamique.

Le support complet de la réflexion n'est généralement pas sans coût observable sur le temps d'exécution du programme [Asai 2014]. La réflexion statique étant appliquée au moment de la compilation du programme, le surcoût qu'elle peut engendrer y est restreint, et possède d'autres avantages tels que la sûreté du typage et l'assurance de produire un code bien formé.

Le C++ permet partiellement la réflexion statique au moyen des *type traits*, une technique qui permet d'associer différentes caractéristiques à des types (et de manière plus étendue à tout élément pouvant être utilisé en argument d'un patron). De plus, une proposition d'évolution du langage [Sutter 2019] permettrait la définition de métaclasses [1]. Une métaclasse telle que présentée

1. Le terme « métaclasse » est ici à différencier de l'usage qui en est fait par exemple en Java.

par la proposition est un nouveau genre de fonction évaluée par le compilateur et qui pourra, entre autres, itérer sur les différents membres d'une classe (introspection structurelle) et produire une nouvelle classe (intercession structurelle). Il s'agit ainsi d'une extension au fonctionnement actuel des classes qui font déjà ce travail au niveau du compilateur (par exemple, une classe définie avec le mot-clé `class` aura des membres par défaut privés, avec le mot-clé `struct` par défaut publics ; par héritage, des fonctions membres peuvent se voir ajouter la propriété `virtual` automatiquement). Le programme décidant de ce qui est produit à partir de la définition faite est alors écrit en C++.

Cet outil rend possible l'implémentation de concepts dont les contraintes seront garanties par le compilateur, par exemple celui d'interface (qui serait alors implémentée par une classe ne contenant ni variable membre ni fonction membre définie) ou celui de valeur (une classe fournissant automatiquement, entre autres, les opérateurs de comparaison).

3.2.3 Patrons

Cette méthode de métaprogrammation fait partie de la programmation générative, qui consiste à développer un programme qui produit d'autres programmes en fonction de données d'entrée spécifiques. Les langages C++ et D sont connus pour leur mécanisme de patrons. Nous utiliserons le terme anglais *template* à partir de maintenant dans ce document. Le terme patron pourra être retenu pour les patrons de conception [Gamma et al. 1995]. Ces templates, à l'origine conçus pour permettre la généricité au sein du C++, peuvent être utilisés pour de la métaprogrammation supposée [2] complète au sens de Turing [3] [Unruh 1994 ; Veldhuizen 2003]. Cela est possible grâce aux mécanismes d'instanciation et de spécialisation des templates implémentés par ces langages.

L'évaluation des templates est effectuée par le compilateur, elle est donc purement statique. Cela empêche une forme de métaprogrammation durant l'exécution du programme mais permet en contrepartie la génération de programmes qui ne subissent pas de surcoût en temps d'exécution comparés à une écriture manuelle équivalente.

D'autres langages ont un support de la généricité que l'on distingue des templates du C++, parce qu'ils n'offrent pas les mêmes avantages, en les nommant « génériques ». Les génériques du langage C# ne permettant, entre autres, pas la spécialisation, ils ne permettent donc pas l'implémentation d'algorithmes de métaprogrammation comme le font les templates. D'autre part, leur résolution (la substitution des types) est effectuée durant l'exécution du programme, ce qui induit un surcoût. En Java, les génériques sont implémentés par effacement de type, limitant la sûreté du typage à une vérification à la compilation.

Un avantage des génériques par rapport aux templates est qu'ils ne causent la génération que d'une seule instance de l'élément générique. En revanche, dans le cadre de l'écriture d'un métaprogramme en C++, le nombre d'instances de templates qui peuvent être générées par le compilateur peut être grand. Une limite, configurable, de nombre d'instances générées en cascade existe pour empêcher une génération infinie en cas d'erreur de programmation.

Pour que la métaprogrammation template puisse être efficace (étant donné la création potentielle de nombreuses instances d'un template, les appels à des fonctions intermédiaires, ...), certaines optimisations fournies par les compilateurs sont indispensables (notamment l'*inlining*

2. Les parties du langage C++ sur lesquelles repose la preuve n'étant pas définies formellement, la preuve ne peut pas être formelle elle-même.

3. La Turing-complétude est limitée par la profondeur d'instanciation autorisée par le compilateur, réglable arbitrairement.

qui permet d'annuler le coût d'un appel de fonction) ou au moins particulièrement bénéfiques (par exemple l'EBO (*Empty Base Optimization*) qui permet de réduire la taille en mémoire d'une classe en cas d'héritage avec une classe de taille nulle).

3.2.4 Programmation multi-étapes

La programmation multi-étapes permet de générer du code qui sera ensuite évalué dans une étape ultérieure de l'exécution du programme. Des annotations peuvent être utilisées pour distinguer le code censé être évalué durant une étape de celui qui doit être transmis tel quel à l'étape suivante. Un avantage de cette méthode réside dans la validité par rapport au langage, attestée par le compilateur, à la fois du code évalué durant une étape ainsi que de celui transmis.

Certains langages Lisp permettent cela. Il existe la fonction `quote` qui permet de ne pas évaluer une expression. Le mécanisme de *backquote* (macro caractère `) est similaire mais permet avec la function *unquote* (macro caractère ,) que des portions de l'expression soient évaluées au sein d'une expression non évaluée (donc transmise à l'étape suivante, mais avec une partie évaluée).

Les templates peuvent être assimilés à une forme de programmation multi-étapes dans la mesure où l'évaluation d'un template peut causer l'évaluation d'au moins un autre template, de manière similaire à ce qu'il est possible de faire en Lisp.

3.2.5 Conclusion

La métaprogrammation est un moyen de générer un programme spécialisé pour résoudre efficacement un problème donné en bénéficiant d'un niveau plus élevé d'abstraction. Cela permet ainsi d'écrire un programme générique pour résoudre un ensemble de problèmes sans sacrifier les performances. Lorsque la métaprogrammation intervient durant l'exécution du programme, il est difficile, sinon impossible, de ne pas causer de surcoût. Pour cette raison, les méthodes s'appliquant avant ou pendant la compilation ont été préférées. La métaprogrammation template, qui intervient durant la compilation, possède l'avantage d'être exprimée dans le même langage que le langage « objet » et ainsi d'assurer que le code généré est valide. Enfin, non seulement le C++ est plus couramment utilisé et *a fortiori* plus éprouvé que le D, mais il évolue beaucoup (depuis la norme validée en 2011) dans une direction favorisant la métaprogrammation (`constexpr` et `consteval`, structures conditionnelles statiques, métaclasses, ...).

Les travaux présentés dans cette thèse utilisant tous la métaprogrammation template en C++, nous donnons dans ce chapitre les éléments permettant d'expliquer son fonctionnement.

3.3 Métaprogrammation template en C++

Telle qu'est conçue la généricité en C++, celle-ci permet la métaprogrammation. Cela a été montré la première fois par Erwin Unruh [Unruh 1994] au moyen d'un programme affichant dans sa sortie d'erreur durant la compilation la suite des nombres premiers. De par la nature à l'origine de la mise en œuvre de la métaprogrammation en C++, utilisant les templates, cette manière de programmer a pris le nom de TMP (*template metaprogramming*), en français métaprogrammation par les patrons ou plus communément métaprogrammation template. Suite à cela, de nombreux usages de la métaprogrammation template ont été faits, introduisant le concept de « bibliothèque active » [Veldhuizen et Gannon 1998], c'est-à-dire une bibliothèque dont une portion du travail

est effectué durant la phrase de compilation, par le compilateur, et qui peuvent donc agir de manière analogue à une extension de compilateur. Cette fonctionnalité de métaprogrammation du langage est améliorée au fur et à mesure de ses évolutions. Cette section présente certaines techniques élémentaires de métaprogrammation et, grâce à l'utilisation de ces dernières, les outils utilisés durant la thèse.

3.3.1 Métafonction

Une métafonction, au sens de la généricité, correspond au modèle générique capable de produire une fonction. Cependant, comme présenté dans la section précédente, le terme fonction template est préféré. Au sens de la métaprogrammation, à l'inverse, le terme métafonction est utilisé, et c'est à celui-ci que correspondront les utilisations futures de ce mot.

Une métafonction est une fonction exécutée durant la compilation et dont le résultat est également utilisable durant cette phase. Il est possible de retourner une valeur (une instance d'un type donné) ou un type.

Cette section présente diverses métafonctions, en suivant un ordre de difficulté croissant, avec pour objectif la familiarisation avec les techniques utilisées en métaprogrammation. L'intérêt des métafonctions présentées peut paraître limité, et il est vrai que, sans les accompagner d'applications, il peut être difficile de se rendre compte des usages qui peuvent être faits. Des applications ne seront cependant pas traitées ici, mais dans des parties ultérieures pour lesquelles les concepts explorés, ainsi que certaines métafonctions, seront indispensables.

3.3.1.1 Métafonctions pures

Une métafonction est pure si elle est entièrement résolue durant la phase de compilation. Un exemple de telle métafonction est le calcul, au moyen de la métaprogrammation template, d'une factorielle. Cet exemple va nous permettre de définir certains points utiles par la suite, en particulier certains usages. S'il existe plusieurs manières d'écrire un même programme, c'est également le cas pour les métaprogrammes et le C++ ne fait pas exception. Les évolutions récentes du langage ont apporté de nouveaux outils pour la métaprogrammation, et ceci va se poursuivre avec les futurs standards [Sutter 2019 ; Meneide 2020]. Plusieurs versions vont être présentées pour le cas de la factorielle.

Avant d'écrire un métaprogramme, l'équation (3.1) donne une définition possible de ce qu'est la factorielle.

$$n! = \Pi_{i=1}^{n} i \tag{3.1}$$

Si en programmation plus classique, l'écriture récursive d'une fonction est souvent évitée, potentiellement pour des raisons d'optimisation, ce ne sera pas le cas en métaprogrammation template. En effet, tout comme en programmation fonctionnelle, il n'y a pas d'effet de bord en métaprogrammation template, ce qui rend nécessaire d'exprimer les fonctions de manière récursive. Ainsi, pour produire le métaprogramme, l'équation (3.2) donnant une autre définition de la factorielle, récursive cette fois, est plus adaptée.

$$\begin{cases} 0! = 1 \\ n! = n \times (n-1)! \end{cases} \tag{3.2}$$

Puisque le métaprogramme est évalué durant la compilation, sa complexité et son temps d'exécution peuvent être ignorés au regard de l'exécutable produit et de son temps d'exécution. Malgré cela, le temps d'évaluation d'un métaprogramme ayant un impact direct sur le temps de compilation, il ne sera pas inutile de chercher à écrire des implémentations de complexité minimale.

À partir de cette définition par récurrence de la factorielle, il est possible d'écrire le métaprogramme de l'extrait de code 3.3.

```
template<unsigned int n>
struct Factorial {
  static constexpr unsigned int value = n * Factorial<n-1>::value;
};

template<>
struct Factorial<0> {
  static constexpr unsigned int value = 1;
};
```

(⩾ C++11)

Extrait de code 3.3 – Factorielle en C++11

On retrouve dans ce code les deux égalités du système l'équation (3.1) affectées à la valeur membre `value`, la terminaison étant implémentée par une spécialisation du template pour le cas particulier de la valeur 0. Ce premier exemple montre comment il est possible d'employer les templates pour faire travailler le compilateur : lorsque l'instanciation de `Factorial` est demandée, le compilateur doit créer une classe possédant un membre dont la valeur, pour être déterminée, requiert l'instanciation d'un autre template (excepté pour 0). Ainsi, si `Factorial<3>::value` est demandé, le compilateur devra également instancier `Factorial<2>`, `Factorial<1>` et `Factorial<0>`, et ce membre `value` prendra la valeur $3 \times 2 \times 1 \times 1$, ce qui sera évalué par le compilateur à la valeur 6.

On peut s'assurer qu'il n'y a aucun calcul effectué durant l'exécution du programme en observant l'appel à la métafonction et l'assembleur que cela produit dans l'extrait de code 3.4.

```
int i = Factorial<10>::value;
```

(⩾ C++98)

```
mov DWORD PTR [rsp-0x4], 3628800
```

(x86 Intel)

Extrait de code 3.4 – Utilisation de la métafonction `Factorial` et assembleur produit

On voit dans l'instruction assembleur, où 3 628 800 correspond à 10!, qu'il n'y a pas d'appel à une fonction, contrairement à ce qui est obtenu avec une version non basée sur la métaprogrammation (extrait de code 3.5).

L'assembleur présenté dans l'extrait de code 3.5, commenté et simplifié pour ne conserver que les parties qui nous intéressent, est ce qui est produit à partir d'une fonction factorielle écrite classiquement. La présence d'une instruction `call` et d'une boucle rend assez évident le possible gain de performance qui peut être atteint au moyen de la métaprogrammation. À noter toutefois que pour des exemples particulièrement simples – c'est le cas de la factorielle – un compilateur est habituellement en mesure de produire un code pour lequel l'appel est remplacé par un `mov` identique à celui obtenu par métaprogrammation. Avec GCC 8.1.0, il faut cependant, pour cela, activer les optimisations incluses dans `-O2` plus `-ftree-loop-vectorize` (incluse dans `-O3`), la

```
factorial(int):          ; argument stored in edi
  mov eax, 1             ; eax is the return value
  cmp edi, 1
  jle .L4                ; edi <= 1
  mov edx, 1
.L3:                     ; begin loop, edx from 1 to edi
  imul eax, edx
  add edx, 1
  cmp edi, edx
  jne .L3                ; loop while edx < edi
  ret
.L4:
  ret
main:
  mov edi, 10            ; prepare argument
  call factorial(int)    ; call factorial
  ret
```

(x86 Intel)

Extrait de code 3.5 – Assembleur produit par GCC pour une implémentation usuelle d'une factorielle

résolution en un simple nombre n'est alors pas toujours garantie. Par contre, en employant la métaprogrammation, il est certain que cette optimisation sera faite, et ce quel que soit le niveau d'optimisation utilisé.

L'utilisation du mot-clé `constexpr` permet de dire au compilateur que la valeur lui est connue (la compilation échoue dans le cas contraire) et donc de l'utiliser dans les contextes où cette connaissance est nécessaire pour opérer. Le mot-clé `static` permet de rendre la variable propre à la classe et non à ses instances.

Ainsi, `Factorial<3>::value` est équivalent au nombre 6 et peut être utilisé en particulier comme argument d'un autre template. On peut alors écrire `Factorial<Factorial<3>::value>::value`, équivalent à 3!!.

Une écriture encore plus moderne est possible en C++14, permettant une utilisation de la factorielle très similaire à un appel de fonction (les parenthèses étant remplacées par des chevrons) grâce à l'utilisation d'une variable template.

Le C++14 apporte une fonctionnalité au langage qui permet une écriture plus légère en utilisant les variables template (extrait de code 3.6). Cette solution rend au mieux le terme de métafonction au regard de l'appel à celle-ci : `factorial<10>`. Outre les chevrons, remplaçant les parenthèses, la syntaxe est identique entre l'appel d'une fonction et celui d'une métafonction conçue de la sorte.

```
template<unsigned int n>
constexpr unsigned int factorial = n * factorial<n-1>;

template<>
constexpr unsigned int factorial<0> = 1;
```

(⩾ C++14)

Extrait de code 3.6 – Factorielle en C++14

On observe au travers de ces exemples l'aspect programmation fonctionnelle de la métaprogrammation en C++. Pour s'en convaincre, l'extrait de code 3.7 présente une implémentation

possible de la factorielle en Haskell, un langage de programmation fonctionnel.

```haskell
factorial 0 = 1
factorial n = n * factorial (n-1)
```

(Haskell)

Extrait de code 3.7 – Factorielle en Haskell

Si l'écriture d'une métafonction au moyen d'un template de classe est courante et encore usitée en C++ moderne, il est cependant possible dans certains cas d'arriver à une écriture moins fonctionnelle et plus impérative avec l'utilisation du mot-clé `constexpr` appliqué aux fonctions, rendant possible leur évaluation durant la compilation, et d'une variante de `if`, à savoir `if constexpr`. L'extrait de code 3.8 implémente la factorielle en utilisant `if constexpr`.

```cpp
template<unsigned int n>
unsigned int factorial() {
  if constexpr(n > 1) return n * factorial<n-1>();
  return 1;
}
```

(⩾ C++17)

Extrait de code 3.8 – Factorielle en C++17

L'instanciation d'un template comportant un branchement à évaluation statique, c'est-à-dire avec `if constexpr`, ignore les branches dont la condition est fausse, ce qui permet de terminer la récursion lorsque $n \leqslant 1$.

Le C++20 introduit un nouveau mot-clé, `consteval` [R. Smith et al. 2018a], qui permet la définition de fonctions évaluées immédiatement par le compilateur. À la différence de `constexpr` qui permet l'utilisation de la fonction même si au moins un de ses arguments n'est pas connu durant la compilation (elle s'effectue alors durant l'exécution), `consteval` fait échouer la compilation si l'appel ne peut être effectué durant celle-ci. L'extrait de code 3.9 est une implémentation de la factorielle utilisant cette fonctionnalité. Dans ce cas, `constexpr` est à éviter : si la fonction est utilisée avec un argument dont la valeur est inconnue durant la compilation, cette implémentation récursive de la factorielle sera inefficacement employée durant l'exécution du programme. Or, une erreur d'utilisation permet facilement de donner un argument que le compilateur considérera comme inconnu (par exemple, en utilisant une variable non déclarée `constexpr` pour contenir la valeur donnée en argument).

```cpp
consteval unsigned int factorial(int n) {
  if(n > 1) return n * factorial(n-1);
  return 1;
}
```

(⩾ C++20)

Extrait de code 3.9 – Factorielle en C++20

Le mécanisme de `consteval` peut être utilisé pour remplacer un template pour implémenter une métafonction lorsque celle-ci manipule des valeurs (par opposition à des types) et est pure (n'opérant que durant la compilation). Quant aux fonctions `constexpr`, leur évaluation est effectuée entièrement durant la compilation ou, exclusivement, entièrement durant l'exécution, en plus de ne pas non plus permettre le traitement de types. Pour ces raisons, elles ne forment pas une base utilisable pour la métaprogrammation dont fait usage ce document.

3.3.1.2 Métafonctions mixtes

Si l'on se limite aux métafonctions pures uniquement, la métaprogrammation ne permet alors que le déplacement d'un traitement durant l'exécution à un traitement durant la compilation. Elles sont alors intéressantes par exemple s'il s'agit de produire un résultat à partir de données connues durant la compilation et qui seront utilisées par le programme durant son exécution. Cette section présente, au travers de l'exemple du calcul d'une puissance (x^n où x est un entier inconnu durant la compilation et n un entier naturel connu), les capacités des métafonctions mixtes. Une métafonction mixte est effectivement exécutée durant la compilation, mais produit un résultat qui sera ensuite traité durant l'exécution du programme plutôt qu'une donnée constante.

Dans le cas où l'exposant est un entier naturel, l'équation (3.3) est une définition de la puissance.

$$x^n = \Pi_{i=1}^{n} x \tag{3.3}$$

Une première manière d'implémenter ce calcul consiste à écrire directement la boucle comme montré dans l'extrait de code 3.10.

```
int pow(int x, int n) {
  int r = 1;
  while(n-- > 0) r *= x;
  return r;
}
```

(⩾ C++98)

Extrait de code 3.10 – Implémentation possible de la fonction `pow`

Cependant, ce code est mal optimisé : malgré la connaissance de n, au pire une boucle est effectuée, et selon le compilateur utilisé, environ 50 lignes d'assembleur pour $n = 24$. Il est possible de faire bien mieux. Si l'on écrit directement dans le programme :

`x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x`

(il y a 24 occurrences de x), le compilateur parvient à produire un code assembleur ne faisant que 5 multiplications. Cela est possible par exponentiation rapide [4] [Gordon 1998]. Il est évident qu'écrire un code similaire est impensable : s'assurer d'avoir le bon nombre de x ou modifier la valeur de l'exposant sont des tâches particulièrement difficiles à accomplir.

Grâce à la métaprogrammation, il est possible d'avoir le meilleur des deux solutions. À l'instar du cas de la factorielle, une définition par récurrence comme dans l'équation (3.4) est plus simple à transcrire en programmation fonctionnelle, et donc en métaprogrammation template (extrait de code 3.11).

$$\begin{cases} x^0 = 1 \\ x^n = x \times x^{n-1} \end{cases} \tag{3.4}$$

4. À noter : en utilisant des flottants (norme IEEE754), cette optimisation se heurte à la non associativité de leur multiplication et un compilateur ne va l'appliquer que s'il lui est autorisé de ne pas tenir compte de cette propriété. C'est donc au prix d'une éventuelle perte de précision des résultats que cette optimisation peut être forcée pour les flottants.

```cpp
template<unsigned int n>
constexpr int pow(int x) {
  return x * pow<n-1>(x);
}

template<>
constexpr int pow<0>(int) {
  return 1;
}
```

(⩾ C++11)

Extrait de code 3.11 – Métafonction puissance

Cette fonction s'utilise comme montré dans l'extrait de code 3.12, à gauche. L'assembleur présenté à droite correspond à ce qui est généré par le compilateur. Seules 5 multiplications sont nécessaires à l'évaluation de cette nouvelle fonction, exactement comme si les multiplications avaient été écrites directement.

```cpp
// volatile : évite la suppression
// des instructions inutilisées
int main() {
  int volatile x = 2;
  int volatile r = pow<24>(x);
}
```

(⩾ C++98)

```asm
; extrait de l'assembleur produit
imul ecx, ecx
imul ecx, eax
imul ecx, ecx
imul ecx, ecx
imul ecx, ecx
```

(x86 Intel)

Extrait de code 3.12 – Utilisation de la métafonction `pow` et assembleur produit

Ce qui peut être retenu, c'est qu'en écrivant une métafonction, il est possible de générer un code C++ qui pourra ensuite être plus efficacement optimisé par le compilateur.

3.3.1.3 Mécanisme d'instanciation

Dans cette section, nous allons présenter l'implémentation d'une métafonction pure retournant un terme, dont l'indice sera en paramètre, de la suite de Fibonacci. Les équations (3.5) à (3.7) définissent cette suite.

$$F_0 = 0 \tag{3.5}$$

$$F_1 = 1 \tag{3.6}$$

$$\forall n \in [\![2, +\infty[\![, F_n = F_{n-1} + F_{n-2} \tag{3.7}$$

En C++, la métafonction `Fibonacci` peut être implémentée comme dans l'extrait de code 3.13. Lors de l'appel, par exemple `Fibonacci<7>`, le compilateur doit résoudre la métafonction pour les valeurs 6 et 5. La complexité associée au nombre d'instances qui doivent être créées est exponentielle en n, l'argument donné à la fonction. Évidemment, si cette fonction devait être implémentée dans un langage de programmation impérative, par exemple, elle ne le serait pas ainsi puisqu'il est possible de la résoudre en complexité linéaire.

```cpp
template<unsigned int n>
constexpr unsigned int Fibonacci = Fibonacci<n-1> + Fibonacci<n-2>;

template<>
constexpr unsigned int Fibonacci<0> = 0;

template<>
constexpr unsigned int Fibonacci<1> = 1;
```

(⩾ C++14)

Extrait de code 3.13 – Métafonction `Fibonacci`

La métaprogrammation template hérite de la programmation fonctionnelle certaines caractéristiques, et comme l'ont montré les précédents extraits de code, certaines manières de penser dont la récursivité. Cette métaprogrammation repose sur le mécanisme des templates, nécessaires à la généricité du langage. En conséquence, un autre aspect fonctionnel dont elle hérite est l'absence d'effet de bord de ses fonctions. En effet, un template ne peut que produire un résultat à partir des arguments qui lui sont fournis, mais il ne peut en aucun cas modifier d'état.

Grâce à cela, il est garanti que deux instanciations d'un même template avec le même n-uplet d'arguments produiront exactement le même résultat. Un compilateur peut donc réutiliser le résultat d'une instanciation si celle-ci est à nouveau demandée, et c'est ce qui est fait en pratique.

Transitivement, une métafonction ne possède pas non plus d'effet de bord et ne sera exécutée qu'une fois par le compilateur pour un même n-uplet d'arguments. Ainsi, si le calcul de la factorielle d'un entier n nécessite bien de générer n instances d'un template, demander à nouveau ce calcul ne causera aucune instanciation supplémentaire, et par extension, demander le calcul de la factorielle d'un entier supérieur ne demandera que la différence entre ces deux nombres en instances additionnelles. Enfin, pour cet exemple du calcul de Fibonacci, on peut se rendre compte que la complexité qui semble exponentielle sera effectivement linéaire.

3.3.1.4 Traitements sur des types

La métaprogrammation en C++ ne permet pas seulement de travailler avec des valeurs mais aussi avec des types. Un exemple des plus simples (sans aller jusqu'à la fonction identité) est la manipulation d'un type pour le modifier. Il est par exemple possible de lui retirer ses qualificatifs (`const` et `volatile`) grâce à la métafonction RemoveCV (extrait de code 3.14). Le principe à implémenter est le suivant : par défaut, c'est le type lui-même qui est retourné ; si le type est qualifié d'un `const`, c'est le type sans le qualificatif ; de même pour `volatile` ; enfin, le dernier cas est celui où les deux qualificatifs sont employés, et encore une fois, ce que la métafonction retourne est le type nu.

```cpp
template<typename T> struct RemoveCV
{ using type = T; };
template<typename T> struct RemoveCV<T const>
{ using type = T; };
template<typename T> struct RemoveCV<T volatile>
{ using type = T; };
template<typename T> struct RemoveCV<T const volatile>
{ using type = T; };
```

(⩾ C++11)

Extrait de code 3.14 – Métafonction RemoveCV

La manipulation de types à la manière de simples données permet d'imaginer l'application de traitements classiques en programmation, mais transposés à la métaprogrammation. Pour cela, une bonne première étape consiste à créer une structure de donnée basique pour contenir les types. La liste de types [Alexandrescu 2001] (*typelist*) est un des premiers conteneurs de type a avoir été créé et une implémentation possible (extrait de code 3.15) consiste en une liste chaînée de types, avec un type particulier servant à reconnaître la fin. Des métafonctions spécifiques accompagnent cette structure de donnée pour la manipuler, mais celles-ci ne seront pas abordées pour cette implémentation spécifique des listes de types. Pour cela, l'ouvrage « *Modern C++ Design* » [Alexandrescu 2001] est une bonne référence.

```
struct EndOfList {};

template<typename Head_, typename Tail_ = EndOfList>
struct TypeList {
  using Head = Head_;
  using Tail = Tail_;
};
```

(⩾ C++11)

Extrait de code 3.15 – Liste de types

Cette manière d'implémenter les listes de types n'est cependant plus aussi utile qu'elle a pu l'être avant C++11 : l'arrivée des templates variadiques a permis une très grande simplification de l'écriture et de la manipulation des structures de données en métaprogrammation. Une implémentation minimale d'une liste de types est présentée dans l'extrait de code 3.16.

```
template<typename...> struct TypeList {};
```

(⩾ C++11)

Extrait de code 3.16 – Liste de types moderne minimale

Cette définition permet ensuite l'écriture de métafonctions pour manipuler la structure de données. La suite de cette partie va présenter certaines de ces métafonctions, à commencer par l'ajout d'un élément en fin de liste. Avant de détailler le fonctionnement de la métafonction `TypeListPushBack`, l'extrait de code 3.17 présente un exemple d'utilisation.

```
using Types  = TypeList<char, int, float, char, bool>;
using Result = TypeListPushBack<Types, double>::type;
// Result is: TypeList<char, int, float, char, bool, double>
```

(⩾ C++11)

Extrait de code 3.17 – Utilisation de la métafonction `TypeListPushBack`

L'implémentation (extrait de code 3.18) de cette métafonction est composée de deux parties : la déclaration et la définition. La déclaration n'est pas systématiquement séparée de la définition, mais cela est nécessaire dans cet exemple : la définition permet d'avoir une vue sur les paramètres templates plus fine qu'un simple type. La partie définition est en réalité une spécialisation, et par ce biais, le pack `Ts` est indiqué comme correspondant aux types contenu dans la liste : l'avoir nommé permet de l'utiliser par la suite.

Ajouter un élément en fin de liste devient alors simple : il suffit de créer une nouvelle liste dont les éléments sont ceux de la liste originale suivis du type à insérer. Il est important de noter que la liste donnée en argument n'a pas été modifiée par l'opération. D'aucune manière il n'est

```cpp
template<typename, typename> struct TypeListPushBack;

template<typename... Ts, typename T>
struct TypeListPushBack<TypeList<Ts...>, T> {
  using type = TypeList<Ts..., T>;
};
```

(⩾ C++11)

Extrait de code 3.18 – Ajout d'un élément à une liste de types

possible de la modifier, et ce sera le cas pour toutes les données qui seront manipulées par les métafonctions : elle sont immuables. Plutôt que d'ajouter un élément à une liste, il s'agit en fait ici de créer une nouvelle liste comportant le nouvel élément, et c'est un abus de langage par analogie à ce qui est fait de similaire en programmation. En cela, la métaprogrammation template possède un attribut de la programmation fonctionnelle pure que nous avons déjà mentionné : l'absence d'effet de bord.

Une autre opération intéressante est l'accès à un élément de la liste connaissant son indice. Cette métafonction, `TypeListGet`, peut s'utiliser comme dans l'extrait de code 3.19.

```cpp
using Second = TypeListGet<Types, 1>::type;
// Second is: int
```

(⩾ C++11)

Extrait de code 3.19 – Utilisation de la métafonction `TypeListGet`

L'implémentation (extrait de code 3.20) est composée de trois parties : la déclaration, le cas général et le cas particulier qui permet de terminer la récursion. Comme cela a été fait pour `TypeListPushBack`, il est possible d'accéder aux éléments de la liste par un pack. Par ailleurs, il est aussi possible d'isoler le premier (`H`) des autres (pack `Ts`). Accéder au $I^{\text{ème}}$ élément (I non nul) d'une liste équivaut à accéder au $(I-1)^{\text{ème}}$ élément d'une liste privée de son premier élément : en séparant le premier type des autres dans la liste de types, il est possible d'appeler `TypeListGet` en lui donnant comme argument une liste de types ne comportant pas le premier élément `H` mais seulement les suivants `Ts...`. L'appel à `TypeListGet` est précédé du mot-clé `typename` ici. Cet usage de ce mot-clé sert à indiquer au compilateur que le membre `type` sera un type (plutôt qu'une variable, par exemple). Le compilateur ne peut pas le vérifier lui-même dans ce contexte car le template `TypeListGet` pourrait être tout à fait différent (le membre `type` pourrait être d'une autre nature, voire ne pas exister) selon les arguments qui lui sont

```cpp
template<typename, unsigned> struct TypeListGet;

template<typename H, typename... Ts, unsigned i>
struct TypeListGet<TypeList<H, Ts...>, i> {
  using type = typename TypeListGet<TypeList<Ts...>, i-1>::type;
};

template<typename H, typename... Ts>
struct TypeListGet<TypeList<H, Ts...>, 0> {
  using type = H;
};
```

(⩾ C++11)

Extrait de code 3.20 – Accès à un élément d'une liste de types

donnés, or à ce niveau, ces arguments sont inconnus. À partir de C++20, cet usage ne sera plus obligatoire [Ranns et Vandevoorde 2018], le compilateur étant capable de supposer qu'un type est attendu puisque l'instruction sert à définir un nouveau type. Le cas terminal arrive lorsque l'indice est nul, auquel cas l'élément qui doit être retourné est le premier élément de la liste, lequel est facilement accessible.

Les métafonctions peuvent ensuite être utilisées dans l'implémentation d'autres métafonctions comme c'est le cas de `TypeListPushBack` pour `TypeListRev` par exemple, qui permet d'inverser l'ordre des éléments d'une liste donnée en argument, et qui peut être appelé comme présenté dans l'extrait de code 3.21.

```
using Result = TypeListRev<Types>::type;
// Result is: TypeList<bool, char, float, int, char>
```

(⩾ C++11)

Extrait de code 3.21 – Exemple d'appel de la métafonction `TypeListRev`

L'implémentation (voir l'extrait de code 3.22) est composée des trois mêmes parties que l'est `TypeListGet`. Le principe est par ailleurs assez semblable, le cas général consistant à isoler le premier élément de la liste pour l'ajouter en fin d'une sous liste dont l'ordre des éléments aura aussi été inversé. Le cas particulier permettant de terminer la récursion est simplement l'inversion des éléments d'une liste vide, qui revient à retourner une liste vide.

```
template<typename> struct TypeListRev;

template<typename H, typename... Ts>
struct TypeListRev<TypeList<H, Ts...>> {
  using Rev  = typename TypeListRev<TypeList<Ts...>>::type;
  using type = typename TypeListPushBack<Rev, H>::type;
};

template<>
struct TypeListRev<TypeList<>> {
  using type = TypeList<>;
};
```

(⩾ C++11)

Extrait de code 3.22 – Inversion de l'ordre des éléments d'une liste de types

3.3.1.5 Liste de types : exemple

Pour terminer sur ce sujet des listes de types avec un exemple plus complexe, nécessitant l'implémentation de plusieurs métafonctions, le prochain cas étudié est la construction d'une liste dans laquelle les valeurs (qui sont des types) sont toutes uniques, à partir d'une liste quelconque. Cette métafonction se nomme `TypeListUniq` et l'objectif est de pouvoir l'utiliser à la manière présentée dans l'extrait de code 3.23.

```
using Result = TypeListUniq<
    TypeList<int, int, char, int, bool, char, int>
  >::type;
// Result is: TypeList<int, char, bool>
```

(⩾ C++11)

Extrait de code 3.23 – Appel de la métafonction `TypeListUniq`

Pour résoudre le problème, il est possible de raisonner de la même manière que pour les exemples précédents : isoler la queue de la liste (tous les éléments excepté le premier) pour faire l'appel récursif et construire la solution en utilisant la valeur retournée par cet appel ainsi que la tête de la liste, tout en prévoyant un cas terminal pour lequel le retour ne nécessite pas d'appel récursif. Dans le présent cas, il est possible de construire le résultat en y insérant le type courant uniquement s'il n'est pas déjà présent. Cette simple description implique deux métafonctions : une permettant de savoir si un type est présent (ou contenu) dans une liste, et une autre un peu particulière permettant de sélectionner un résultat selon une condition (une forme de structure de contrôle).

La première métafonction nécessaire, `TypeListContains` (extrait de code 3.24) permettant de déterminer si un type est contenu dans une liste est semblable aux métafonctions présentées jusqu'ici. Si le premier type de la liste n'est pas le type recherché, la métafonction s'appelle elle-même sur une liste privée de son premier élément. Si au contraire le premier type de la liste est le type recherché, elle retourne vrai. Un second cas terminal existe pour le cas où le type n'existe pas dans la liste. Si cela arrive, la récursion doit être terminée lorsque la liste est vide, auquel cas la métafonction retourne faux.

```
template<typename, typename> struct TypeListContains;

template<typename H, typename... Ts, typename T>
struct TypeListContains<TypeList<H, Ts...>, T> {
  static constexpr bool value = TypeListContains<TypeList<Ts...>, T>::value;
};

template<typename T, typename... Ts>
struct TypeListContains<TypeList<T, Ts...>, T> {
  static constexpr bool value = true;
};

template<typename T>
struct TypeListContains<TypeList<>, T> {
  static constexpr bool value = false;
};
```

(⩾ C++11)

Extrait de code 3.24 – Test de la présence d'un type dans une liste

La deuxième métafonction, `If` (extrait de code 3.25), permet de sélectionner un type parmi deux selon une valeur booléenne. Celle-ci est fréquemment utilisée en métaprogrammation et est implémentée de manière assez immédiate : si le booléen est vrai, le premier type est retourné, sinon c'est le second qui l'est.

```cpp
template<bool, typename, typename> struct If;

template<typename T, typename F>
struct If<true, T, F> {
  using type = T;
};

template<typename T, typename F>
struct If<false, T, F> {
  using type = F;
};
```

(⩾ C++11)

Extrait de code 3.25 – Sélection d'un type selon une valeur booléenne

À l'aide de ces deux métafonctions, l'implémentation de `TypeListUniq` (extrait de code 3.26) est possible. Comme expliqué partiellement, deux cas sont à distinguer. Si plus aucun élément n'est à traiter, la liste qui a été construite peut être retournée. Autrement, il faut ajouter à la liste en cours de construction l'élément courant s'il n'est pas déjà présent. Pour cela, dans un premier temps, `Result` est déterminé en fonction de la présence de `H` dans la liste construite courante `Rs...` : la liste retournée est `Rs...` à laquelle on ajoute en fin `H` seulement s'il n'existe pas déjà dans `Rs...` (ce que l'on détermine en utilisant la métafonction `TypeListContains`) grâce à la métafonction `If`. `Result` deviendra ensuite la liste courante lors de l'appel récursif.

```cpp
template<typename, typename = TypeList<>> struct TypeListUniq {};

template<typename H, typename... Ts, typename... Rs>
struct TypeListUniq<TypeList<H, Ts...>, TypeList<Rs...>> {
  using Result = typename If<
    TypeListContains<TypeList<Rs...>, H>::value,
    TypeList<Rs...>,
    TypeList<Rs..., H>
  >::type;
  using type = typename TypeListUniq<TypeList<Ts...>, Result>::type;
};

template<typename Result>
struct TypeListUniq<TypeList<>, Result> {
  using type = Result;
};
```

(⩾ C++11)

Extrait de code 3.26 – Implémentation de TypeListUniq

3.3.2 *Helper type* et *helper variable*

Avant d'entrer dans un cas concret de métaprogrammation, cette section présente quelques pratiques utilisées dans cette thèse. Dans la section précédente, plusieurs métafonctions ont été présentées. Pour les appeler, il était nécessaire d'accéder à un membre (type ou valeur), et ce explicitement, en suffixant d'un `::type` ou d'un `::value`. Ceci peut être allégé en fournissant des *helpers*. Pour la bibliothèque standard, le choix a été fait d'ajouter au nom de la métafonction le suffixe `_t` pour celles retournant un type, et `_v` pour celles retournant une valeur. Elle fournit ainsi pour la métafonction `std::decay` le *helper type* `std::decay_t` et pour la métafonction `std::is_same` la *helper variable* `std::is_same_v`.

Puisqu'ici aucune contrainte de compatibilité n'a besoin d'être respectée, il est possible de plutôt renommer l'implémentation effective en la suffixant de `Impl` ou en l'encapsulant dans un espace de noms `impl`. L'identifiant d'origine peut alors être utilisé comme alias.

L'implémentation de tels outils est très simple, l'extrait de code 3.27 définit un type template qui appelle la métafonction et accède au membre `type`, et de manière similaires, l'extrait de code 3.28 définit une variable template qui appelle la métafonction et accède au membre `value`.

```
template<typename TL>
using TypeListUniq = typename TypeListUniqImpl<TL>::type;
```

(⩾ C++11)

Extrait de code 3.27 – *Helper type* `TypeListUniq`

```
template<typename TL, typename T>
constexpr bool typeListContains = TypeListContains<TL, T>::value;
```

(⩾ C++14)

Extrait de code 3.28 – *Helper variable* `typeListContains`

De plus, cette technique permet de proposer une interface plus naturelle lors de l'utilisation de la métafonction. En effet, le second paramètre template de `TypeListUniqImpl` possède une valeur par défaut et une valeur ne doit pas lui être attribué par l'utilisateur lors du premier appel. S'il le fait, le résultat peut devenir faux ou pire, la fonction peut ne plus compiler. Avec l'extrait de code 3.27, la métafonction `TypeListUniq` n'a bien qu'un paramètre, le risque est donc évité.

Ces *helpers* sont possibles seulement pour les métafonctions qui ne retournent qu'une information et dont le nom est prédictible (`type` ou `value`). Si une métafonction doit enfreindre une de ces contraintes, elle devra être appelée de manière spécifique.

3.3.3 Patrons d'expression

Cette section traite des ET (*Expression Templates*) [Veldhuizen 1995], patrons d'expression en français. Les ET vont servir de prétexte à l'exploration de diverses techniques de métaprogrammation template, mais ils ont aussi été utilisés dans l'implémentation d'une partie des travaux réalisés durant la thèse. Les ET permettent une représentation, sous la forme d'un type, d'un AST issu d'une expression C++ et à laquelle il est ensuite possible d'appliquer un traitement durant la compilation afin par exemple de générer un nouveau code C++.

Elles sont utilisées par exemple pour optimiser l'évaluation d'une expression arithmétique, particulièrement dans le cadre de vecteurs et de matrices. Sans ET, un calcul comme $A = B + C + D + E$, où B, C, D et E sont des vecteurs de n éléments, va produire une séquence de trois calculs avec ces vecteurs : calcul du temporaire $T_1 = B + C$, puis du temporaire $T_2 = T_1 + D$ et enfin du temporaire $T_3 = T_2 + E$, enfin déplacé dans A. Autrement dit, cela correspond à la séquence des trois boucles qui permettent d'implémenter les formules des équations (3.8) à (3.10) :

$$\forall i \in [\![1, n]\!], \quad T_1[i] = B[i] + C[i] \tag{3.8}$$

$$\forall i \in [\![1, n]\!], \quad T_2[i] = T_1[i] + D[i] \tag{3.9}$$

$$\forall i \in [\![1, n]\!], \quad A[i] = T_2[i] + E[i]. \tag{3.10}$$

Grâce aux ET, il est possible de ne produire qu'une boucle, implémentant la formule de l'équation (3.11) :

$$\forall i \in [\![1, n]\!], \quad A[i] = B[i] + C[i] + D[i] + E[i]. \tag{3.11}$$

En effet, l'expression C++ écrite ne procède pas au calcul immédiatement mais permet de générer à la place un autre code. Ce type de techniques est utilisé par Blitz++ [Veldhuizen 2000] ou LLANO [Kirby 2003].

Afin d'expliquer le fonctionnement des ET, la suite de cette section présente des implémentations partielles possibles, en augmentant la complexité au fur et à mesure [5]. Les extraits de code peuvent être un peu simplifiés afin de ne pas devoir se perdre dans des détails de mise en œuvre qui ne sont pas directement liés au fonctionnement même des ET.

3.3.3.1 Premier jet

Le premier exemple est particulièrement simple, mais permet de montrer les principes. L'objectif dans un premier temps est d'être capable d'exprimer une expression (arithmétique) et de l'évaluer indépendamment de sa construction : faire de l'évaluation dite paresseuse [Iglberger et al. 2012]. Pour cela, il faut représenter l'expression qui va se limiter, dans un premier temps, à des opérations binaires sur des opérandes connues. Acceptons deux opérateurs : `Add` et `Mul`. Limitons une opérande à soit une valeur, soit une expression, où une expression est une opération appliquée à deux opérandes.

$$operator ::= Add \mid Mul \tag{3.12}$$

$$operand ::= value \mid expression \tag{3.13}$$

$$expression ::= operator(operand, operand) \tag{3.14}$$

L'extrait de code 3.29 définit les deux opérateurs. Afin de fonctionner sans tenir compte du type effectif des opérandes, leur fonction d'évaluation est générique et retourne respectivement la somme et la multiplication des arguments qui leur sont donnés.

Ces opérateurs définis, il est ensuite possible d'implémenter ce qui permettra de contenir une

```
struct Add {
  template<typename T>
  static auto eval(T lhs, T rhs) {
    return lhs + rhs;
  }
};

struct Mul {
  template<typename T>
  static auto eval(T lhs, T rhs) {
    return lhs * rhs;
  }
};
```

(≥ C++14)

Extrait de code 3.29 – Définition des opérateurs `Add` et `Mul`

5. Des exemples complets sont disponibles à l'adresse https://phd.pereda.fr/compl/mp.

valeur. Pour que cela s'intègre dans une expression, une valeur doit aussi pouvoir être évaluée. L'évaluation d'une valeur consiste simplement à retourner la valeur elle-même comme le montre l'extrait de code 3.30. Il faut par ailleurs un constructeur acceptant un argument du type de la valeur.

```
template<typename T>
class Value {
  T _value;

public:
  Value(T value): _value{value} {}

  auto eval() const { return _value; }
};
```

(⩾ C++14)

Extrait de code 3.30 – Définition de `Value`

Enfin, l'extrait de code 3.31 définit ce qu'est une expression (binaire) : un opérateur et deux opérandes. L'évaluation de l'expression correspond à l'application de son opérateur aux arguments fournis. Le type `Value` et le type `BinExpr` définissent une fonction membre `eval` sans paramètre et qui retourne un résultat. Cette propriété commune, générique, permet d'utiliser l'un ou l'autre des deux types de la même manière. Grâce à cela, il est possible de composer une expression à partir d'une autre expression pour former un arbre dont les feuilles seront nécessairement de type `Value`.

```
template<typename Op, typename LHS, typename RHS>
struct BinExpr {
  LHS lhs;
  RHS rhs;

  auto eval() const {
    return Op::eval(lhs.eval(), rhs.eval());
  }
};
```

(⩾ C++14)

Extrait de code 3.31 – Définition d'une expression binaire

L'expression arithmétique $v_0 + v_1 \times v_2$ peut être représentée par l'arbre de la figure 3.1 et retranscrit en un type C++ comme montré dans l'extrait de code 3.32 : `SubExpr` représente le sous arbre de racine $\times$ dans la figure 3.1, c'est-à-dire l'application de l'opérateur `Mul` sur les valeurs `v1` et `v2` ; puis `Expr` représente l'arbre entier par l'application de l'opérateur `Add` sur la valeur `v0` et le sous arbre `SubExpr`.

```
using SubExpr = BinExpr<Mul, Value<int>, Value<int>>;
using MyExpr  = BinExpr<Add, Value<int>, SubExpr>;
```

(⩾ C++11)

Extrait de code 3.32 – Création du type représentant l'expression $v_0 + v_1 \times v_2$

Pour terminer, sur cet exemple, l'assembleur généré montre que le code produit par l'évaluation de l'expression (extrait de code 3.33) est identique à une écriture directe sans la couche d'ET.

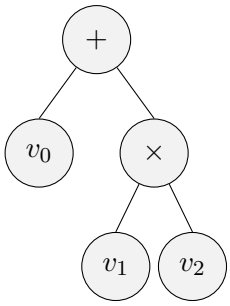


FIGURE 3.1 – Arbre de l'expression $v_0 + v_1 \times v_2$

```
MyExpr expr{{v0}, {{v1}, {v2}}};
int result = expr.eval();
```

(⩾ C++11)

```
imul eax, edx
add eax, ecx
```

(x86 Intel)

Extrait de code 3.33 – Instanciation, évaluation et assembleur généré de l'expression $v_0 + v_1 \times v_2$

L'instanciation de l'expression est pour le moment assez peu intuitive et ne correspond pas à ce qui est attendu d'un patron d'expression. Ce problème est réglé par la suite, lorsque la surcharge des opérateurs sera introduite afin d'obtenir une écriture semblable à l'expression mathématique correspondante. La première ligne (partie gauche) de l'extrait de code 3.33 correspondra alors à un code tel que `v0 + v1*v2`. Il s'agit ici d'évaluation paresseuse puisque cette expression C++ ne procède pas encore au calcul demandé sur les opérandes mais construit à la place un type (et son instance) représentant ce calcul. L'évaluation peut ensuite être déclenchée (éventuellement par un opérateur, ici par un appel de fonction), celle-ci faisant appel à un code généré par l'ET.

3.3.3.2 Arité générique

Telles que sont implémentées les différentes parties nécessaires au fonctionnement des ET jusqu'à présent, seules les opérations binaires sont possibles. Nous proposons d'implémenter la grammaire suivante (où ε dénote l'absence d'élément) :

$$operator ::= Add \mid Mul \mid Minus \tag{3.15}$$

$$operand ::= value \mid expression \tag{3.16}$$

$$operands ::= \varepsilon \mid operand, operands \tag{3.17}$$

$$expression ::= operator(operands) \tag{3.18}$$

Pour cela, plusieurs solutions sont réalisables. Cette section propose une implémentation, construite à partir de ce qui a déjà été proposé, et qui sera adaptée aux modifications ultérieures. Seule la classe template `BinExpr` est remplacée par une nouvelle classe template `Expr` (extrait de code 3.34) capable de représenter une opération dont l'arité est librement fixée à l'utilisation. Le template est donc variadique sur les types des opérandes.

La classe template `std::tuple` implémente un n-uplet qui permet à la classe template `Expr` de contenir les instances des opérandes. La fonction d'évaluation doit être implémentée pour appeler la fonction d'évaluation de l'opérateur (`Op::eval`) en passant comme arguments le résultat obtenu par l'évaluation de chacun des opérandes, c'est-à-dire le résultat de l'appel de la fonction membre `eval()` sur chacun des éléments contenu dans le n-uplet.

```
template<typename Op, typename... Operands>
class Expr {
  std::tuple<Operands...> operands;

public:
  auto eval() const {
    return eval(std::make_index_sequence<sizeof...(Operands)>{});
  }

private:
  template<std::size_t... is>
  auto eval(std::index_sequence<is...>) const {
    return Op::eval(std::get<is>(operands).eval()...);
  }
};
```

(⩾ C++14)

Extrait de code 3.34 – Définition d'une expression d'arité générique

Une manière classique de faire cela est d'utiliser un pack d'indices des éléments auxquels on souhaite accéder au sein du tuple (dans ce cas, tous, dans l'ordre). Que ce soit sous la forme d'un pack permet son utilisation durant la compilation en appliquant une expansion sur celui-ci, de la forme `accessTupleElement<is>(tuple)...` où `accessTupleElement` est un template permettant d'accéder à un élément d'un tuple à partir de son indice, et `is` un pack d'indices.

La fonction membre `eval()` ne dispose pas d'un tel pack d'indice. Une indirection supplémentaire est donc nécessaire afin de créer un pack *ad hoc* contenant les indices d'accès aux éléments du n-uplet, c'est-à-dire tous les nombres de 0 à $N-1$ si N est le nombre d'opérandes. La bibliothèque standard fournit pour cela une classe template : `std::index_sequence`. Son rôle est de contenir un pack d'indices, en écrivant par exemple `std::index_sequence<0, 1, 2, 3>`. En complément, `std::make_index_sequence` retourne une instance de `std::index_sequence` contenant des indices allant de 0 jusqu'à une borne supérieure, exclue, donnée en argument.

Le langage fournit depuis C++11 un opérateur, `sizeof...`, qui retourne le nombre d'éléments contenus dans un pack. Il faut donc utiliser ce nombre comme argument à `std::make_index_sequence` afin d'obtenir le pack nécessaire (contenu dans le type d'une variable) que l'on peut ainsi transmettre à la fonction qui va véritablement évaluer l'expression.

Cette fonction accepte une instance de `std::index_sequence` afin de pouvoir en extraire le pack `is` que son type contient. L'expansion d'un pack pouvant se faire sur un motif complexe (voir la section 2.13), il est possible d'appliquer sur le n-uplet d'opérandes l'accès à un opérande puis l'appel sur celui-ci de sa fonction d'évaluation. La fonction template `std::get` permet d'accéder à un élément d'un n-uplet par son indice. Par exemple, l'appel à la fonction membre `eval` d'une classe `Expr` dont le pack d'opérandes est de taille 3 va se traduire par la création d'un pack d'indices $\{0, 1, 2\}$ qui produit par expansion l'extrait de code 3.35.

```
Op::eval(
  std::get<0>(operands).eval(),
  std::get<1>(operands).eval(),
  std::get<2>(operands).eval()
)
```

(⩾ C++11)

Extrait de code 3.35 – Appel à la fonction d'évaluation d'un opérateur avec pack étendu

3.3.3.3 *Embedded Domain Specific Language*

Un DSL (*Domain Specific Language*) est un langage conçu pour être adapté à répondre aux besoins d'un domaine précis, s'opposant aux langages de programmation plus polyvalents comme le C++ ou le Python par exemple. Lorsqu'un DSL est construit au sein d'un autre langage de programmation, il s'agit d'un EDSL (*Embedded Domain Specific Language*). Un EDSL permet de fournir aux utilisateurs du langage hôte, par le moyen d'une bibliothèque, des outils dont la grammaire et/ou la syntaxe sont optimisées pour retranscrire une problématique d'un domaine en code source valide pour le langage hôte et qui implémente le comportement désiré.

Les ET permettent l'implémentation de langages embarqués en utilisant des fonctions pour construire le type complexe d'une expression. Le C++ supportant la surcharge de la plupart des opérateurs, il est classique d'utiliser ces fonctions particulières pour remplir ce rôle. Dans le cas d'un langage dédié à la construction d'expressions arithmétiques, l'utilisation des opérateurs pour construire une expression devient même une évidence.

Pour cela, il suffit de surcharger un opérateur, par exemple `operator*` pour `Mul` (extrait de code 3.36). Cet opérateur particulier accepte deux paramètres qui sont des valeurs, utilisées comme opérandes d'une expression représentant l'application de l'opérateur `Mul`.

```
template<typename T>
auto operator*(Value<T> lhs, Value<T> rhs) {
  return BinExpr<Mul, Value<T>, Value<T>>{lhs, rhs};
}
```

(⩾ C++14)

Extrait de code 3.36 – Opérateur spécifique produisant une expression de multiplication de deux valeurs

Afin de pouvoir construire le type correspondant à l'expression spécifique utilisée jusqu'ici $v_0 + v_1 \times v_2$, la définition de l'opérateur `operator+` de l'extrait de code 3.37 est nécessaire. Celle-ci accepte une valeur et une expression binaire de multiplication en paramètre et retourne la nouvelle expression.

```
template<typename T, typename Op>
auto operator+(Value<T> lhs, BinExpr<Op, Value<T>, Value<T>> rhs) {
  return BinExpr<Add, Value<T>, BinExpr<Op, Value<T>, Value<T>>>{lhs, rhs};
}
```

(⩾ C++14)

Extrait de code 3.37 – Opérateur spécifique produisant une expression d'addition d'une valeur avec une expression

La limite de cette manière de procéder est vite atteinte : si cela permet d'écrire quelques expressions, supporter davantage de formes requiert l'écriture de nombreuses surcharges d'opérateurs. Une alternative plus générique consiste à être capable de combiner toute expression avec un opérateur valide, il suffit donc de reconnaître ce qu'est une expression et ne pas accepter ce qui n'en est pas une. S'il est possible de supposer qu'un type correspond à une expression en vérifiant la présence de la fonction membre `eval` qui doit être présente, cela peut mener à de faux positifs. Le mécanisme qui sera employé repose sur les *type traits* [Maddock 2002]. Le principe d'un *type trait* est de fournir des informations à propos d'un type au moyen d'un template l'acceptant en seul argument. Par exemple, il existe dans la bibliothèque standard le

type trait `std::is_pointer` qui retourne, sous la forme d'un booléen nommé `value`, vrai si le type correspond à un pointeur. D'autres *type traits* possèdent plusieurs caractéristiques comme c'est le cas de `std::numeric_limits` qui peut informer, entre autres, sur les valeurs pouvant être représentées par le type (minimale, maximale, précision, ...).

Ce mécanisme peut s'appliquer de deux façons. En étant intrusifs dans l'implémentation des types qui sont acceptés dans une expression, il est possible de leur ajouter un membre qui sera spécifique et détecté automatiquement. L'avantage réside dans la simplicité de l'implémentation de la classe de *traits*. Cependant, en plus du fait que ce mécanisme est intrusif – et donc inadapté aux types qui ne peuvent pas avoir de membre, les types primitifs, ainsi qu'aux types ne pouvant être modifiés, provenant d'une bibliothèque tierce par exemple –, pour véritablement éliminer le risque de faux positif, il faut que le membre testé ne puisse pas être confondu avec un membre qui pourrait exister sans avoir cette signification. Cela est possible en utilisant un type membre dont la valeur est un type spécifique créé pour ce besoin.

La seconde façon de procéder n'est pas intrusive mais requiert une classe de *traits* un peu plus complexe. Aucun des types acceptés n'a à être altéré, mais une spécialisation de la classe de *traits* doit être fournie pour chacun d'eux (extrait de code 3.38). De plus, puisqu'il s'agit de spécialisations d'une classe template, il est dans certains cas possible d'exprimer algorithmiquement la caractéristique pour déduire par une métafonction sa valeur. Le *type trait* `std::is_pointer` permet d'illustrer ce principe, bien qu'en en étant un cas trivial, puisqu'il suffit de spécialiser le template pour le cas des pointeurs en retournant alors « vrai », et de retourner « faux » dans le cas général. À noter également qu'il serait impossible d'appliquer la première méthode pour ce cas là puisque les pointeurs sont des types directement fournis par le langage et sont ainsi non modifiables. C'est donc cette seconde manière qui est habituellement utilisée, y compris dans la bibliothèque standard.

```
template<typename T> struct IsExpr: std::false_type {};

template<typename Op, typename LHS, typename RHS>
struct IsExpr<BinExpr<Op, LHS, RHS>>: std::true_type {};

template<typename T>
struct IsExpr<Value<T>>: std::true_type {};
```

(⩾ C++11)

Extrait de code 3.38 – Classe de *traits* déterminant si un type peut être utilisé au sein d'une expression

En utilisant cette information, il est alors possible de simplifier la création des opérateurs en les rendant génériques sur les types qu'ils acceptent (extrait de code 3.39). La SFINAE

```
template<
  typename LHS, typename RHS,
  std::enable_if_t<IsExpr<LHS> && IsExpr<RHS>>* = nullptr
>
auto operator*(LHS lhs, RHS rhs) {
  return BinExpr<Mul, LHS, RHS>{lhs, rhs};
}
```

(⩾ C++14)

Extrait de code 3.39 – Opérateur générique produisant une expression de multiplication

(*Substitution Failure Is Not An Error*) (voir la section 2.7) permet ici de s'assurer que les deux types sont valides au sein d'une expression, en utilisant la classe de *traits* définie juste avant.

L'écriture d'un autre opérateur binaire, par exemple `operator+`, est très similaire : il suffit de changer le nom de la fonction et le type d'opération à appliquer (`Mul` devient `Add` par exemple). Il est généralement préférable d'éviter d'avoir recours au préprocesseur en C++ dès lors que le langage propose une alternative, car le CPP, n'étant qu'un système de remplacement de texte, permet l'écriture et la génération de code de manière quelquefois trop libre. Cependant, pour le moment, le C++ ne permet pas l'écriture de code génératif d'opérateurs. Lorsqu'il s'agit de créer un DSL, il est souvent nécessaire de surcharger, comme nous l'avons présenté, plusieurs opérateurs, et pour réduire la répétition du code, le préprocesseur est donc la seule solution standard qui existe.

3.3.3.4 Séparation données/traitement

Afin de rendre le système d'ET plus polymorphe et extensible, l'implémentation peut être adaptée pour séparer la manière de représenter l'expression et le comportement des opérateurs. Cela permet, à partir d'une même expression, d'appliquer des traitements différents en utilisant une évaluation différente. Par exemple, l'expression C++ `v0 * -(v1 + v2)` peut être évaluée à la manière de l'arithmétique élémentaire, auquel cas l'exécution de cette expression correspond au calcul $v_0 \times -(v_1 + v_2)$; mais il est également possible de faire correspondre aux opérateurs de l'expression des opérateurs correspondants de l'algèbre booléenne, le calcul devenant $v_0 \wedge \neg(v_1 \vee v_2)$.

Une possibilité est le *tag dispatching* [Estérie et al. 2014], une technique qui permet d'utiliser un type pour sélectionner une fonction à appeler et qui est basée sur les mécanismes liés à la surcharge de fonctions du C++. Les structures qui avaient pour membre les fonctions à exécuter (extrait de code 3.29) sont remplacées par des *tags*, de simples structures creuses dont seul le type, différent pour chacune, est utilisé (extrait de code 3.40).

```
namespace tag {

struct Add {};
struct Mul {};
struct Minus {};

}
```

(⩾ C++98)

Extrait de code 3.40 – *Tags* correspondant aux opérations

La fonction d'évaluation d'une expression doit être adaptée à ce mécanisme. Pour cela, l'opérateur duquel elle obtenait jusqu'alors l'implémentation à exécuter et qui est remplacé par un simple *tag* ne va plus servir qu'à sélectionner une fonction fournie par un évaluateur qui reste à spécifier (extrait de code 3.41).

```
template<typename Evaluator, std::size_t... is>
auto eval(std::index_sequence<is...>) const {
  return Evaluator::eval(Op{}, std::get<is>(operands).template
  ↪  eval<Evaluator>()...);
}
```

(⩾ C++14)

Extrait de code 3.41 – Fonction d'évaluation d'une expression basée sur le *tag dispatching*

Cette fonction template `eval`, membre de `Expr` (extrait de code 3.34), accepte en paramètre template l'évaluateur : une classe fournissant une fonction d'évaluation surchargée basée sur le *tag dispatching* pour sélectionner la version qui implémente l'opération souhaitée (le *tag* étant l'instance créée du type `Op` (paramètre de `Expr`)).

L'évaluation de l'opérateur est appliquée sur les opérandes eux-mêmes évalués par leur fonction membre `eval`, faisant indirectement appel à cette surcharge (extrait de code 3.41). Cela cause donc en cascade l'évaluation de l'entièreté de l'arbre.

Grâce à ce système, choisir la manière dont sera évaluée l'expression est devenu très simple : il suffit d'écrire une classe disposant de fonctions membres `eval` pour chaque opération qui doit être supportée, en utilisant le *tag* correspondant. L'extrait de code 3.42 présente ainsi l'implémentation des évaluateurs sus-cités calculant selon l'arithmétique élémentaire (`EvaluatorElem`) et selon l'arithmétique booléenne (`EvaluatorBool`).

```
struct EvaluatorElem {
  template<typename T>
  static auto eval(tag::Add, T lhs, T rhs) { return lhs + rhs; }

  template<typename T>
  static auto eval(tag::Mul, T lhs, T rhs) { return lhs * rhs; }

  template<typename T>
  static auto eval(tag::Minus, T rhs) { return -rhs; }
};

struct EvaluatorBool {
  static auto eval(tag::Add, bool lhs, bool rhs) { return lhs or rhs; }
  static auto eval(tag::Mul, bool lhs, bool rhs) { return lhs and rhs; }
  static auto eval(tag::Minus, bool rhs) { return not rhs; }
};
```

(⩾ C++14)

Extrait de code 3.42 – Évaluateurs par arithmétique élémentaire et booléenne basés sur le *tag dispatching*

Cependant, l'extensibilité de ce système n'est pas totale. En effet, si, *a posteriori*, un utilisateur souhaite ajouter le support d'une nouvelle opération à un évaluateur existant, il doit le modifier. Une technique ressemblant au *tag dispatching*, mais utilisant la spécialisation d'un template selon un *tag* plutôt que la résolution de surcharge de fonction, permet de supprimer cette limite. L'implémentation des *tags* ne change pas, en revanche, comme la manière de définir l'évaluateur est différente, l'évaluateur lui-même devenant un template plutôt qu'un type (extrait de code 3.44), la fonction d'évaluation d'une expression doit être adaptée (extrait de code 3.43).

```
template<template<typename> class Evaluator, std::size_t... is>
auto eval(std::index_sequence<is...>) const {
  return Evaluator<Op>::eval(std::get<is>(operands).template eval<Evaluator>()...);
}
```

(⩾ C++14)

Extrait de code 3.43 – Fonction d'évaluation d'une expression (template)

Ainsi, plutôt que d'accepter comme paramètre template un type, elle accepte un template, et lors de l'évaluation, le *tag* correspondant à l'opérateur n'est plus donné comme premier argument

de la fonction d'évaluation (plus exactement, c'était une instance de ce *tag*) mais comme argument template à l'évaluateur.

```
template<typename> struct EvaluatorElem;

template<>
struct EvaluatorElem<tag::Add> {
  template<typename T>
  static auto eval(T lhs, T rhs) { return lhs + rhs; }
};

template<>
struct EvaluatorElem<tag::Mul> {
  template<typename T>
  static auto eval(T lhs, T rhs) { return lhs * rhs; }
};

template<>
struct EvaluatorElem<tag::Minus> {
  template<typename T>
  static auto eval(T rhs) { return -rhs; }
};
```

(⩾ C++14)

Extrait de code 3.44 – Évaluateur par arithmétique élémentaire (template)

Les codes présentés dans ce document font abstraction de difficultés techniques par souci de simplicité et de concision. Notamment, les paramètres dont le type est un paramètre template de la fonction devraient être traités spécifiquement (voir la section 2.12 sur la transmission parfaite).

3.3.3.5 Application partielle et curryfication

Un autre usage possible des ET est la construction d'une expression qui comporte la structure de celle-ci mais pas (tous) les arguments sur lesquels elle va s'appliquer. Dès lors, l'expression construite peut être réutilisée pour différentes données, et elle peut par exemple servir de fonction anonyme [Järvi et al. 2003] (fonction *lambda* [Turing 1937], dont une syntaxe propre au langage a été introduite en C++11). Adapter l'implémentation d'ET présentée jusqu'ici à ce type d'application requiert une indirection supplémentaire sur l'évaluation. Celle-ci n'est pas nécessaire si l'évaluateur est fixé à l'avance. Dans le cas présenté, le principe reste donc fondamentalement le même, à ceci près que l'évaluation est déplacée dans une nouvelle classe qui aura connaissance à la fois de l'expression et de l'évaluateur, et qui peut être construite directement par l'utilisateur ou par une fonction d'aide triviale.

La classe évaluable, pour être adaptée à une utilisation semblable à une fonction, surcharge alors `operator()` pour un nombre variadique de paramètres et doit associer ceux-ci aux paramètres attendus de l'expression. Dans le cadre des fonctions lambda, contrairement à ce qui a été présenté avant, les paramètres de l'expression ne sont pas immédiatement attachés à des valeurs. L'implémentation présentée ici permet, à l'instar de la bibliothèque Boost.Lambda de [Järvi et Powell 2003], l'utilisation de variables spéciales (extrait de code 3.45) servant à indiquer la correspondance entre les paramètres de la fonction générée et les opérandes de l'expression.

```cpp
template<std::size_t>
struct Argument {};

namespace arg {

Argument<0> _0;
Argument<1> _1;
// ...

}
```

(⩾ C++98)

Extrait de code 3.45 – Définitions d'arguments servant dans la création de fonctions anonymes

Ces variables ont seulement besoin de comporter l'information de quel argument doit les remplacer, ainsi un nombre ordinal suffit pour les désigner.

Ensuite, il faut être capable d'évaluer l'expression en utilisant les arguments donnés lors de l'appel sans qu'ils soient préalablement correctement associés comme c'était le cas jusqu'ici. Pour cela, il est possible de regrouper ces arguments dans un n-uplet grâce auquel, en utilisant un indice connu à la compilation, on peut accéder à un argument précis. Cette technique a déjà été présentée dans les exemples précédents, lorsque les opérandes étaient stockés.

Prenons un exemple concret pour expliquer le principe : `(_1 - _0)(5, 7)`. La première paire de parenthèses comporte une expression, `_1 - _0`, dont les opérandes sont de types respectifs `Argument<1>` et `Argument<0>`. Grâce aux ET, cette expression construit un arbre dont la racine est l'opérateur - et les deux feuilles sont les arguments `_1` et `_0`. Cet arbre expose un `operator()` afin d'être utilisable comme une fonction, dont les paramètres sont au nombre de deux (déterminé par le nombre ordinal maximum utilisé par les feuilles). Lorsque cette fonction est appelée (la seconde paire de parenthèses), les arguments qui lui sont donnés (dans cet exemple, 5 et 7) vont être regroupés dans un n-uplet. L'argument à l'indice 0 du n-uplet est 5 et l'argument à l'indice 1 du n-uplet est 7. Un premier pack *ad hoc* est construit pour contenir $\{0, 1\}$, de la même manière que cela avait été fait dans la section 3.3.3.2. Celui-ci permet d'obtenir à partir de l'ensemble des opérandes de l'expression (`_1` et `_0`) le pack des indices correspondant aux nombres ordinaux qui sont stockés dans ces variables, c'est-à-dire $\{1, 0\}$. De manière similaire, l'expansion de ce nouveau pack permet la génération des arguments pour l'appel à la fonction d'évaluation de l'opérateur de l'expression en produisant quelque chose de similaire à `Substract::eval(7, 5)`.

Une autre utilisation de ces fonctions anonymes est présentée dans l'extrait de code 3.46. À noter, encore une fois, que l'étape liant l'expression à un évaluateur est dispensable si ce dernier est fixé à l'avance, et les bibliothèques implémentant ce concept de fonction anonyme ne l'ont généralement pas. Pour cette raison, et pour alléger l'écriture ici, les exemples en sont exempts.

```cpp
// sort in descending order
std::sort(std::begin(c), std::end(c), _0 > _1);
// generate square values
std::transform(std::begin(c), std::end(c), std::begin(o), _0 * _0);
```

(⩾ C++98)

Extrait de code 3.46 – Exemples d'utilisation d'une fonction anonyme

Lorsque l'évaluation est ainsi reportée, l'idée d'application partielle ou de curryfication [Kuchen et Striegnitz 2002] vient rapidement.

Une application partielle consiste à créer une fonction à partir d'une autre, la nouvelle ayant une arité plus faible que l'ancienne. Prenons une fonction quelconque f acceptant n paramètres :

$$f : X_0 \times X_1 \times ... \times X_{n-1} \to Y$$

L'application partielle de f qui lie le premier paramètre à une valeur précise correspond alors à :

$$bind(f, x_0) : X_1 \times ... \times X_{n-1} \to Y$$

Pour y parvenir, une sous partie des paramètres de la fonction doit être déterminée, c'est-à-dire qu'une valeur connue doit être assignée pour chacun d'eux. Il existe une implémentation d'applications partielles de fonctions quelconques en C++ (`std::bind` [ISO et C++ committee 2011, §20.8.9]) qui permet également de réordonner les paramètres (auquel cas l'arité n'est pas réduite) ou l'utilisation d'un même paramètre de la nouvelle fonction comme argument pour plusieurs paramètres de l'ancienne fonction [Järvi et Powell 2001]. Par exemple, il est possible d'effectuer cette transformation :

$$f : X_0 \times X_1 \times X_2 \times X_3 \times X_4 \to Y$$

$$bind(f, x_0, \_0, \_0, \_2, \_1) : X_{1,2} \times X_4 \times X_3 \to Y$$

On associe au paramètre X_0 l'argument x_0, réduisant l'arité de 1, puis on utilise deux fois le premier argument de la fonction créée pour l'associer aux deux paramètres X_1 et X_2, réduisant à nouveau l'arité de 1. Enfin, le dernier argument de la fonction créée est associé au paramètre X_4 tandis que le pénultième argument, _1 est affecté au paramètre X_3. L'équivalent écrit en C++ est présenté par l'extrait de code 3.47.

```
int f(int x0, int x1, int x2, int x3, int x4) {
  return x0 + 2*x1 + 3*x2 + 5*x3 + 7*x4;
}

// the two next lines are equivalent
f(42, 73, 73, 137, 108);
std::bind(f, 42, _1, _1, _3, _2)(73, 108, 137);
```

(⩾ C++11)

Extrait de code 3.47 – Exemple d'utilisation de `std::bind`

La curryfication consiste en la transformation d'une fonction à n paramètres en une nouvelle fonction à 1 paramètre retournant une fonction à $n-1$ paramètres qui fonctionne de la même manière. Cette récursion se termine pour la fonction générée n'ayant qu'un paramètre qui, plutôt que de retourner une fonction sans paramètre, évalue la fonction d'origine. La curryfication de la fonction f présentée ci-dessus est ainsi :

$$curry(f) : X_0 \to (X_1 \to (X_2 \to ...(X_n \to Y)))$$

Il s'agit donc d'une forme d'application partielle qui permet de fournir à une fonction ses arguments un à un plutôt que tous en même temps. Cela permet par exemple de séparer certains processus qui seraient autrement liés par le fait qu'ils produisent chacun un sous-ensemble des arguments à donner à une même fonction. Après curryfication, celle-ci est en charge de contenir

les arguments en attente jusqu'à application complète.

Par exemple, pour une fonction ayant deux paramètres $g : a, b \mapsto a + b$, l'application de la curryfication donnerait une fonction $g_0 : a \mapsto (g_1 : b \mapsto a + b)$. Autrement dit, il est possible d'appeler la nouvelle fonction en fournissant un seul argument, par exemple $g_0(9)$, elle retourne alors la fonction $g_1 : b \mapsto 9 + b$.

Ce mécanisme peut être implémenté dans le cadre des patrons d'expression en changeant la manière de fonctionner de la fonction d'évaluation, à laquelle sont donnés les arguments. Plutôt que d'accepter uniquement exactement le bon nombre d'arguments, celle-ci doit accepter un nombre inférieur, idéalement un seul pour correspondre exactement au concept de curryfication. En permettant d'en accepter une quantité quelconque comprise entre 1 et l'arité de la fonction, la fonction est cependant plus adaptable.

La fonction d'évaluation de l'expression doit accepter un nombre variable de paramètres, et procéder à différents traitements selon la quantité d'arguments fournis. À chaque appel, si le nombre d'arguments fournis en tout n'est pas égal à l'arité de l'expression, la fonction d'évaluation conserve l'ensemble des arguments. Le nombre d'arguments total est déterminé par la somme du nombre d'arguments ainsi conservés et du nombre d'arguments fournis à l'appel de la fonction d'évaluation. Si ce nombre dépasse l'arité lors d'un appel, la compilation échoue. Lorsqu'un appel fait que ce nombre atteint exactement l'arité de l'expression, celle-ci est évaluée en utilisant tous les arguments, ceux conservés ainsi que ceux fournis en dernier.

3.4 Conclusion

Ce chapitre a présenté les outils de métaprogrammation qui sont nécessaires dans les propositions qui sont faites dans les chapitres 4 et 5. La métaprogrammation en général et les différentes méthodes permettant son application ont été détaillées avec les avantages de chacune, ainsi que les raisons nous ayant conduit à choisir la métaprogrammation template.

Dans la suite de ce chapitre, la métaprogrammation template a été présentée en introduisant les concepts qui lui sont propres et les moyens techniques permettant leur implémentation. Durant cette introduction, nous avons montré que la métaprogrammation template permet effectivement d'exécuter une partie d'un programme durant sa compilation. Cela nous permet de profiter de certaines optimisations auxquelles les compilateurs procèdent, voire de les forcer en produisant un code que l'on sait propice à être optimisé. Nous avons ensuite présenté la spécificité de la métaprogrammation template qui est capable de traiter des types comme des variables.

S'agissant de programmation, il existe plusieurs manières d'atteindre un même résultat. Nous avons donc présenté certains de nos usages afin de faciliter la lecture des codes créés durant cette thèse.

Enfin, sous la forme d'un cas pratique, ce chapitre a présenté les patrons d'expression et une implémentation simplifiée de ceux-ci. Les patrons d'expression sont d'abord conçus pour répondre à une grammaire très simple, puis complétés pour supporter une arité quelconque, permettre une écriture naturelle grâce à la surcharge d'opérateurs, ...

La métaprogrammation nous permet notamment, pour les applications présentées dans cette thèse, d'acquérir des informations sur le code en utilisant la technique des patrons d'expression dont l'interface donnée à l'utilisateur est alors un EDSL. Ces informations nous permettent ensuite, à nouveau par l'utilisation de la métaprogrammation template, de procéder à des analyses puis à de la génération de code. C'est ce que nous verrons dans le chapitre suivant.

Chapitre 4

Analyse et parallélisation automatique de boucles

4.1 Introduction

Les principes généraux de la parallélisation automatique ont été introduits dans la section 1.4. Dans ce chapitre, nous nous intéressons en particulier à la parallélisation logicielle, et plus spécifiquement au cas des boucles, puisque le temps d'exécution d'un programme est majoritairement localisé au sein de celles-ci.

S'agissant de parallélisation automatique, aucune information ne doit être apportée par le développeur quant à savoir si le code peut effectivement être exécuté en parallèle de manière valide, l'unique entrée du système étant le code source original devant être parallélisé. Notre objectif n'est cependant pas de modifier la structure de l'algorithme pour le rendre parallélisable s'il ne l'était pas. Ainsi, il s'agit de détecter si le code fourni peut être parallélisé, c'est-à-dire s'il est possible de générer un programme dont le comportement global est le même que celui de sa version séquentielle, donc produisant la même sortie.

Ce chapitre présente dans un premier temps les solutions existantes au problème de la parallélisation automatique en général. Ensuite, notre approche sous forme de bibliothèque [1] est présentée par étape, en expliquant d'abord les conditions qui doivent être respectées pour permettre la parallélisation, puis comment notre bibliothèque acquiert l'information nécessaire à l'aide d'ET (*Expression Templates*). Ce chapitre explique alors les différentes parties de l'EDSL (*Embedded Domain Specific Language*) fourni pour la génération de l'ET, de la représentation des instructions à la caractérisation des fonctions d'indice d'accès à des tableaux. Nous présentons ensuite la vérification des conditions introduites au début du chapitre en analysant, durant la compilation, les informations acquises pour grouper les instructions interdépendantes et déterminer si elles sont parallélisables pour ensuite générer le programme dont seules les parties qui peuvent l'être sont exécutées en parallèle. Enfin, ce chapitre présente des mesures de performances obtenues en utilisant notre bibliothèque et les compare à des programmes équivalents mais écrits sans l'utiliser.

4.2 Travaux connexes

La parallélisation est un sujet très étudié comme montré dans le chapitre 1. La parallélisation automatique a été introduite dans la section 1.4. Celle-ci présente des techniques appliquées par le matériel, ainsi que certaines pouvant être effectuées au niveau logiciel.

Généralement, les outils de parallélisation automatique logicielle fonctionnent directement comme des compilateurs [Blume et al. 1995 ; Fonseca et al. 2016], générant un exécutable parallèle, ou comme des méta-compilateurs [Zima et al. 1988 ; Ahmad et al. 1997], produisant un nouveau code source parallèle à partir d'un code source séquentiel. Il existe également des langages intrinsèquement parallèles [Roscoe et Charles Antony Richard Hoare 1988 ; Loveman 1993 ; Chamberlain et al. 2007] car ils incluent dans leur définition des éléments faisant de la parallélisation automatique par le compilateur une fonctionnalité obligatoire, à l'inverse de la majorité des langages pour lesquels cette optimisation est un atout propre au compilateur qui la propose. Enfin, il existe des bibliothèques logicielles [Chan et Abdelrahman 2004] qui procèdent à l'analyse et à la parallélisation durant l'exécution du programme.

Ces outils peuvent fonctionner en analysant l'ASA (arbre syntaxique abstrait) qu'ils ont construit à la lecture du code source afin de déterminer les dépendances entre les données

1. https://phd.pereda.fr/dev/pfor

utilisées et, lorsque les dépendances observées le permettent, produire une implémentation parallèle [Lazarescu et Lavagno 2012].

Le modèle polyédral de compilation [M. Griebl et al. 1998] est une autre méthode. Dans ce modèle, on représente des boucles imbriquées sous la forme de polytopes sur lesquels des optimisations peuvent être appliquées, ce qui permet la génération d'un code parallèle [Bondhugula et al. 2008].

Ce modèle est très utilisé en pratique car la parallélisation logicielle est fréquemment appliquée aux boucles [Collard 1995 ; Artigas et al. 2000 ; Ramon-Cortes et al. 2018 ; Neth et Strout 2019]. En effet, dans la plupart des programmes, la majorité du temps d'exécution est localisé dans celles-ci, et il semble donc normal d'y consacrer beaucoup d'efforts si l'on souhaite améliorer les performances.

Puisqu'il est nécessaire d'analyser le programme afin de déterminer les portions qui sont parallélisables, les outils proposés sont, comme nous l'avons vu, des compilateurs, méta-compilateurs, des extensions pour des compilateurs existants ou encore des bibliothèques, lesquelles agissent durant l'exécution du programme. À part les bibliothèques, ces solutions contraignent le développeur à l'utilisation d'un environnement de développement dédié, ce qui peut ne pas être applicable pour diverses raisons : non implémentation des standards les plus récents du langage, incompatibilité avec certaines extensions, ... Quant aux bibliothèques, de par leur fonctionnement dynamique, elles induisent un surcoût en temps d'exécution qui peut réduire l'intérêt de la parallélisation.

La métaprogrammation template du C++, introduite dans le chapitre 3, permet l'écriture de bibliothèques dites actives. Cela signifie qu'elles ont la possibilité d'agir, en addition de leur action dynamique, durant la phase de compilation. Ainsi, elles permettent l'acquisition et le traitement d'informations nécessaires à la prise de décision souhaitée. Il s'agit d'une utilisation standard du langage, ce qui signifie une compatibilité *a priori* indépendante des évolutions de celui-ci. Elle est utilisée pour différents aspects de la parallélisation automatique [Joel Falcou et al. 2008 ; Videau et al. 2018].

Nous proposons donc dans ce chapitre une solution de parallélisation automatique de boucles, au niveau logiciel, à l'aide de la métaprogrammation template qui nous permet, par rapport aux solutions évoquées ci-avant, de fournir un outil indépendant du compilateur mais dont le surcoût en temps d'exécution, usuellement impliqué par de telles abstractions, est contrôlé et minimisé.

4.3 Conditions pour la parallélisation de boucles

Cette section présente un ensemble de conditions devant être respectées pour permettre la parallélisation d'une boucle. Par parallélisation d'une boucle, il est entendu que ce sont les itérations de cette boucle qui sont exécutées en parallèle sans altérer le comportement global du programme.

Deux types de conditions vont être abordés : celles permettant d'identifier des dépendances entre les instructions au sein d'une même itération et celles permettant de vérifier l'indépendance des itérations entre elles.

Au cours de ce chapitre, nous utiliserons l'exemple de boucle présenté par l'extrait de code 4.1. Dans cet exemple, les variables `a`, `b`, `c`, `d`, `e` et `f` sont des tableaux de taille suffisante. Cette boucle présente des particularités qui vont permettre de mettre en exergue différentes problématiques et donc de présenter les solutions apportées par la bibliothèque introduite par ce chapitre.

```cpp
// a, b, c, d, e and f are arrays
for(int i = 0; i < n; ++i) {
  a[i] = a[i] * b[i];
  c[i] = c[i+1] - d[i];
  b[i] = b[i] + i;
  d[i] = std::pow(c[i], e[i]);
  f[i*i] = 2 * f[i*i];
}
```

(⩾ C++98)

Extrait de code 4.1 – Boucle travaillant sur des tableaux

4.3.1 Dépendances au sein d'une itération

Au sein d'une même itération la boucle n'est pas à considérer, il s'agit simplement d'une séquence d'instructions. Considérons deux sous-ensembles contigus P_a et P_b qui sont des segments de cette séquence. Pour que ceux-ci puissent être exécutés en parallèle, aucune des instructions du segment P_a (respectivement P_b) ne doit dépendre d'une quelconque instruction du segment P_b (respectivement P_a). Pour formaliser cette contrainte, notons W_i l'ensemble des variables pour lesquelles au moins une instruction du segment P_i fait un accès en écriture et R_i l'ensemble des variables auxquelles au moins une instruction du segment P_i accède en lecture. Les conditions pour considérer les deux segments indépendants, et donc parallélisables, ont été introduites par [Bernstein 1966] et peuvent être exprimées par les équations (4.1) à (4.3).

$$W_b \cap R_a = \emptyset \tag{4.1}$$

$$R_b \cap W_a = \emptyset \tag{4.2}$$

$$W_a \cap W_b = \emptyset \tag{4.3}$$

Les équations (4.1) et (4.2) empêchent l'utilisation d'une même variable dans les deux segments si l'un d'eux la modifie. Cela détecte les dépendances de flux (RAW (*Read After Write*)) et les anti-dépendances (WAR (*Write After Read*)). L'équation (4.3) empêche l'existence d'une variable qui serait modifiée par les deux segments. Cela détecte les dépendances d'écriture (WAW (*Write After Write*)).

Ainsi, lorsqu'une variable est partagée par les deux segments, il est nécessaire que celle-ci n'apparaisse que dans les ensembles R_a et R_b (c'est-à-dire qu'il n'y ait pour cette variable que des accès en lecture) pour ne pas empêcher la parallélisation.

Un segment $P = \{I_1, ..., I_n\}$ peut être analysé et partitionné en r sous-ensembles d'instructions dépendantes D_i tels que :

- $\cup_{i=1}^{r} D_i = P$;
- $\forall i \in [\![1, r]\!]$, il existe une dépendance, directe ou non, entre toutes les instructions de D_i ;
- $\forall i, j \in [\![1, r]\!]$, si $i \neq j$, D_i ne possède aucune instruction dépendante de l'une de celles de D_j.

L'algorithme 4.1 représente une manière d'implémenter cela en utilisant une fonction Test-Bernstein(x, y) qui est vraie si et seulement si les segments x et y respectent les conditions de Bernstein.

En partant d'un ensemble vide G, dont l'objectif est de contenir les sous-ensembles indépendants D_i, pour chaque instruction I_k, considérée comme un segment atomique $P_a = \{I_k\}$, le test d'indépendance est effectué avec chacun des ensembles P_b présents dans G. En cas de

dépendance, l'ensemble P_b est retiré de G et ses instructions sont ajoutées à P_a. Lorsque tous les éléments de G ont été testés, l'ensemble P_a lui est ajouté.

Algorithme 4.1 Identification des groupes d'instructions dépendantes

entrée : un ensemble d'instructions $P = \{I_1, ..., I_n\}$
sortie : l'ensemble des sous-ensembles d'instructions dépendantes
fonction GrouperInstructionsDépendantes(P)
  $G \leftarrow \emptyset$
  pour tout $I_k \in P$ **faire**
    $P_a \leftarrow \{I_k\}$
    $P_c \leftarrow P_a$
    pour tout $P_b \in G$ **faire**
      si not TestBernstein(P_a, P_b) **alors**
        $G \leftarrow G \setminus \{P_b\}$
        $P_c \leftarrow P_c \cup P_b$
    $G \leftarrow G \cup \{P_c\}$
  retourner G

Grâce à cette identification, il est possible de séparer une boucle en de multiples boucles dont on sait que, à l'intérieur de chacune, les instructions sont dépendantes. L'extrait de code 4.1 peut donc être transformé en l'extrait de code 4.2 sans en changer le comportement. En effet, les instructions aux lignes 3 et 5 sont dépendantes à cause de l'utilisation de `b` à la fois en lecture et écriture ; les lignes 4 et 6 sont également dépendantes entre elles à cause des variables `c` et `d` ; enfin, la ligne 7 ne dépend d'aucune autre.

```
1  // a, b, c, d, e and f are arrays
2  for(int i = 0; i < n; ++i) {
3    a[i] = a[i] * b[i];
4    b[i] = b[i] + i;
5  }
6  for(int i = 0; i < n; ++i) {
7    c[i] = c[i+1] - d[i];
8    d[i] = std::pow(c[i], e[i]);
9  }
10 for(int i = 0; i < n; ++i)
11   f[i*i] = 2 * f[i*i];
```

(⩾ C++98)

Extrait de code 4.2 – Boucles séparées implémentant une boucle unique (extrait de code 4.1)

Lors de cette transformation, il est important de conserver l'ordre des instructions dépendantes (au sein d'une même boucle), en revanche, les boucles elles-mêmes peuvent être dans un ordre quelconque.

4.3.2 Dépendances entre les itérations

Les dépendances entre les itérations empêchent une parallélisation triviale (sans avoir recours à des transformations sur le code en entrée). Pour les identifier, il est possible de considérer une boucle exécutant une séquence d'instructions n fois comme une séquence d'instructions composée de n fois la séquence initiale. Cela permet d'utiliser à nouveau les conditions de Bernstein pour déterminer leur indépendance.

Nous allons considérer deux cas de variables auxquelles il peut y avoir des accès au sein d'une boucle : les tableaux et les scalaires. L'intégralité d'un tableau n'est pas utilisée à chaque itération d'une boucle, un indice est généralement employé pour en atteindre une partie spécifique. Cet indice est alors la plupart du temps fonction de l'itération courante. Chaque cellule d'un tableau correspond donc à une variable distincte, au même titre que les scalaires, et le tableau a de dimension 1 sera représenté par le vecteur $a = (a_j)_{j \in [\![0, n_a - 1]\!]}$ où chaque élément a_j correspond à un élément scalaire à l'indice j du tableau.

Ainsi, pour les scalaires, W^s et R^s correspondent aux ensembles des variables auxquelles on accède, respectivement, en écriture et en lecture. Dans le cas des tableaux, les ensembles correspondants sont plus complexes. Pour tout tableau a, F_a est l'ensemble des fonctions $f : \mathbb{N} \to \mathbb{N}$ utilisées pour calculer l'indice d'accès à l'élément du tableau en fonction de l'indice de l'itération. Cet ensemble F_a se décompose en un ensemble F_a^w pour les accès en écriture et F_a^r pour les accès en lecture. Par exemple, pour le tableau `c` de l'extrait de code 4.1, on a $F_c^w = \{i \mapsto i\}$ et $F_c^r = \{i \mapsto i, i \mapsto i+1\}$. À partir de ces fonctions, il est possible de construire les ensembles des éléments des tableaux auxquels accède l'itération i en écriture, W_i^t (équation (4.4)), et en lecture, R_i^t (équation (4.5)). Les ensembles W^t et R^t contiennent les tableaux dont au moins un élément est utilisé respectivement en écriture ou en lecture.

$$W_i^t = \bigcup_{a \in W^t} \bigcup_{f \in F_a^w} a_{f(i)} \tag{4.4}$$

$$R_i^t = \bigcup_{a \in R^t} \bigcup_{f \in F_a^r} a_{f(i)} \tag{4.5}$$

Il est ensuite possible de définir les ensembles W_i (équation (4.6)) et R_i (équation (4.7)). Le premier comporte toutes les variables (scalaires ou éléments d'un tableau) pour lesquelles il y a un accès en écriture, le second en lecture, durant l'itération i.

$$W_i = W^s \cup W_i^t \tag{4.6}$$

$$R_i = R^s \cup R_i^t \tag{4.7}$$

L'adaptation des conditions de Bernstein, en utilisant ces nouvelles variables, à l'identification des dépendances entre plusieurs itérations correspond aux équations (4.8) à (4.10).

$$\forall i \in [\![1, k]\!], \quad W_i \cap \left(\bigcup_{j \neq i} R_j\right) = \emptyset \tag{4.8}$$

$$\forall i \in [\![1, k]\!], \quad R_i \cap \left(\bigcup_{j \neq i} W_j\right) = \emptyset \tag{4.9}$$

$$\bigcap_{i=1}^{k} W_i = \emptyset \tag{4.10}$$

Pour le cas des scalaires, les contraintes peuvent être traduites ainsi : l'accès en lecture ne présente pas de problème, en revanche, l'accès en écriture va systématiquement empêcher la parallélisation dès lors que la boucle possède au moins 2 itérations (non-respect de l'équation (4.10) puisque tout W_i contient la variable).

Il est possible de transformer les équations (4.8) à (4.10) afin d'obtenir les équations (4.8′)

à (4.10′) dans le cas des tableaux.

$$\forall a \in W^t \cap R^t, \quad \forall i \in [\![1,k]\!], \quad \nexists f \in F_a^w \ / \ \exists j \neq i, g \in F_a^r, f(i) = g(j) \tag{4.8′}$$
$$\forall a \in W^t \cap R^t, \quad \forall i \in [\![1,k]\!], \quad \nexists f \in F_a^r \ / \ \exists j \neq i, g \in F_a^w, f(i) = g(j) \tag{4.9′}$$
$$\forall a \in W^t, \quad \forall i \in [\![1,k]\!], \quad \nexists f \in F_a^w \ / \ \exists j \neq i, g \in F_a^w, f(i) = g(j) \tag{4.10′}$$

L'équation (4.10′) est respectée à condition que l'accès à un même indice d'un même tableau en écriture ne se fasse jamais dans deux itérations distinctes. Les équations (4.8′) et (4.9′) empêchent l'accès en lecture à un indice donné dans un tableau durant une itération si cet élément est utilisé en écriture durant une autre itération.

Appliqué à l'extrait de code 4.2, ces équations permettent de déterminer que des trois boucles, seule la deuxième ne peut être parallélisée. Cela est dû à l'instruction accédant au tableau `c` à la fois à l'indice `i` en écriture et `i+1` en lecture, ne respectant donc pas les équations (4.8′) et (4.9′).

Il s'agit alors de produire deux boucles : l'une parallélisée comprenant les instructions de la première et de la dernière ; l'autre exécutée de manière séquentielle correspondant à la deuxième. L'extrait de code 4.3 présente un possible regroupement des boucles, l'ordre des segments d'instructions issus de chaque boucle n'ayant aucune importance au sein de la boucle résultante. La réunion des boucles en fonction de leur capacité à être parallélisées est cruciale pour les performances : sans cela, le nombre total d'itérations est augmenté. Si toutes les boucles intermédiaires partagent la propriété d'être, ou non, parallélisables, alors une seule boucle doit être produite pour la même raison.

```
// a, b, c, d, e and f are arrays
// possibly parallel loop
for(int i = 0; i < n; ++i) {
  a[i] = a[i] * b[i];
  b[i] = b[i] + i;
  f[i*i] = 2 * f[i*i];
}
// sequential loop
for(int i = 0; i < n; ++i) {
  c[i] = c[i+1] - d[i];
  d[i] = std::pow(c[i], e[i]);
}
```

(⩾ C++98)

Extrait de code 4.3 – Boucles groupées en fonction de leur capacité à être exécutée en parallèle (depuis l'extrait de code 4.2)

4.4 Analyse lexicale et syntaxique par métaprogrammation

Il existe plusieurs manières d'obtenir les informations nécessaires à la vérification des conditions énoncées dans la section précédente. Principalement, il est possible d'agir au niveau du compilateur (directement ou au moyen d'une extension) ou durant l'exécution du programme. La métaprogrammation template est une solution intermédiaire qui permet à une bibliothèque (écrite dans le langage du développeur) d'agir sur le déroulement de la compilation.

Notre objectif est de représenter durant la compilation l'ASA (AST (*Abstract Syntax Tree*) en anglais) exécuté par la boucle d'origine. Les patrons d'expression [Veldhuizen 1995] (voir la section 3.3.3) sont adaptés puisqu'ils permettent précisément cela. Grâce à ceux-ci et à la

surcharge des opérateurs permise en C++, nous pouvons fournir un EDSL dont l'utilisation construit un ASA pouvant être analysé durant la compilation par métaprogrammation template.

Cette section traite de deux ASA distincts dont nous nous servons : dans un premier temps celui qui représente l'expression des instructions qui doivent être exécutées et dans un second temps celui qui encode les fonctions d'indice utilisées pour accéder aux éléments des tableaux auxquels les instructions recourent.

4.4.1 Représentation d'une expression

L'ASA d'une expression globale est construit avec des nœuds internes représentant des opérations et des feuilles représentant leurs opérandes. Au sein du code, il va donc s'agir d'utiliser la notion de patron d'expression (voir la section 3.3.3). Pour identifier les feuilles de cet arbre (et ne pas appliquer les opérateurs spécifiques en dehors de notre cas d'utilisation), un type spécifique contenant la variable effective est employé. L'extrait de code 4.4 montre la définition d'une variable de ce type. En réalité, il s'agit d'un template, et ce pour deux raisons. La première est qu'il ne dépend pas du type contenu, à l'instar de ce que font les conteneurs de la bibliothèque standard (par exemple `std::vector`), expliquant la présence de `int*` en premier argument template (celui-ci pourrait être automatiquement déduit en utilisant le CTAD (*Class Template Argument Deduction*), voir la section 2.5). Le second argument permet d'identifier l'opérande durant la compilation. En effet, distinguer une variable d'une autre peut être accompli durant l'exécution, en comparant leur adresse, mais pas durant la compilation : il faut donc ajouter une information permettant cette disjonction. Plusieurs techniques permettent d'automatiser partiellement la génération d'un identifiant différent pour chaque opérande, cependant aucune n'est parfaitement fiable dans tous les contextes, aussi notre bibliothèque n'en intègre pas par défaut.

```
int aData[] = {1, 2, 3, 4, 5};
Operand<int*, 0> a(aData);
```

(⩾ C++14)

Extrait de code 4.4 – Définition d'un opérande de patron d'expression

Un problème qui peut survenir est l'*aliasing* [Landi et Ryder 1991] : une même zone mémoire accessible à partir d'au moins deux noms. Celui-ci existe également dans le cadre des patrons d'expression [Härdtlein et al. 2005]. Dans l'extrait de code 4.5, les opérandes `a` et `b` peuvent éventuellement adresser une même portion de mémoire en fonction des indices utilisés. En effet, le premier élément de `b` est à la même adresse que le sixième élément de `a`, or les deux opérandes possèdent un identifiant différent, faisant d'eux des variables distinctes du point de vue de nos patrons d'expression. À l'inverse, les opérandes `c` et `d` qui partagent exactement la même section

```
int aData[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
int cData[] = {0, 1, 2, 3, 4};

Operand<int*, 0> a(aData);
Operand<int*, 1> b(aData+5); // {5, 6, 7, 8, 9}
Operand<int*, 2> c(cData);
Operand<int*, 2> d(cData);
```

(⩾ C++14)

Extrait de code 4.5 – Définition d'opérandes avec *aliasing*

de mémoire (représentée par `cData`) ne créent pas de problème puisqu'ils ont un identifiant commun, faisant d'eux une seule et même variable au sein d'une expression malgré leur distinction en C++. À noter qu'un problème ne survient que lorsque deux opérandes ayant ce souci d'*aliasing* sont utilisés dans une même expression.

Ainsi que nous l'avons déjà dit, l'adresse en mémoire d'une variable ne peut être connue que durant l'exécution du programme. La détection d'un problème d'*aliasing* ne peut donc pas être effectuée durant la compilation. De plus, pour y parvenir durant l'exécution, une information supplémentaire est nécessaire : le domaine d'utilisation des données qui sont représentées par un opérande. En effet, si l'opérande `a` n'accède qu'aux 5 premiers éléments de son tableau sous-jacent, il ne peut y avoir d'*aliasing* avec `b` (dès lors que des indices négatifs ne sont pas utilisés).

Une bonne pratique reste néanmoins d'éviter l'utilisation de tableaux partiels et de recourir à `std::array` au lieu des tableaux et pointeurs « nus ». Ces derniers sont utilisés dans l'unique but de rendre les exemples plus compacts.

L'utilisation des opérandes est simplifiée par la surcharge des opérateurs pour générer des expressions. Par exemple, dans l'extrait de code 4.6 l'expression `a + b`, stockée dans `e`, fait appel à une surcharge de `operator+` qui produit une instance d'un type représentant l'arbre de la figure 4.1. Ce type est une instanciation d'un template avec pour arguments un type identifiant l'opération effectuée (+) et les deux types des opérandes. Cette instance comporte également les opérandes elles-mêmes afin de les utiliser ultérieurement.

```
int aData[] = {0, 1, 2, 3, 4}, bData[] = {5, 6, 7, 8, 9};
Operand<int*, 0> a(aData);
Operand<int*, 1> b(bData);

auto e = a + b;
// ^ Expression<op::addition, Operand<int*, 0>, Operand<int*, 1>>
```

(≥ C++14)

Extrait de code 4.6 – Expression créée par la surcharge de l'`operator+`

Une instruction C++ est typiquement terminée par un point-virgule. Ceci n'est pas un opérateur et ne peut donc pas être surchargé, c'est pourquoi, afin de permettre la capture d'une expression composée de multiples instructions, nous avons utilisé à la place l'opérateur virgule (`operator,`). La virgule est utilisée de plusieurs manières dans le langage selon le contexte : comme opérateur ou comme élément lexical de ponctuation. Pour s'assurer que la virgule sera utilisée comme opérateur, il est possible de placer l'expression dans une paire de parenthèses. Par exemple, dans l'extrait de code 4.7, `e` est initialisée par une expression utilisant cet opérateur virgule. Dans ce contexte, l'absence de parenthèses aurait conduit à la tentative de définition de trois variables (`e`, `d` et `b`).

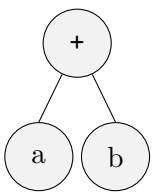


FIGURE 4.1 – ASA représenté par l'expression de l'extrait de code 4.6

```
// a, b, c, d are operands
auto e = (
  a = b + c,
  d = d + 1,
  b = 2 * b
);
```

(⩾ C++14)

Extrait de code 4.7 – Expression comportant plusieurs instructions séparées par des virgules

Cette expression est représentée par l'arbre de la figure 4.2. L'`operator,` est utilisé pour accumuler 3 instructions. On notera l'utilisation d'arguments « bruts » qui ne sont pas explicitement des opérandes de patron d'expression : `1` et `2`. Deux cas doivent être distingués quant à l'utilisation de tels arguments : opérateur unaire et opérateur binaire ; les opérateurs pouvant être surchargés en C++ à ce jour étant tous de l'une de ces deux arités. Si un argument brut est utilisé avec un opérateur unaire, ce dernier sera appliqué directement, résultant en une nouvelle donnée qui n'est pas, elle non plus, un opérande de patron d'expression. Si un argument brut est utilisé avec un opérateur binaire, pour que cela forme une expression, l'autre argument de cet opérateur doit être un opérande de patron d'expression.

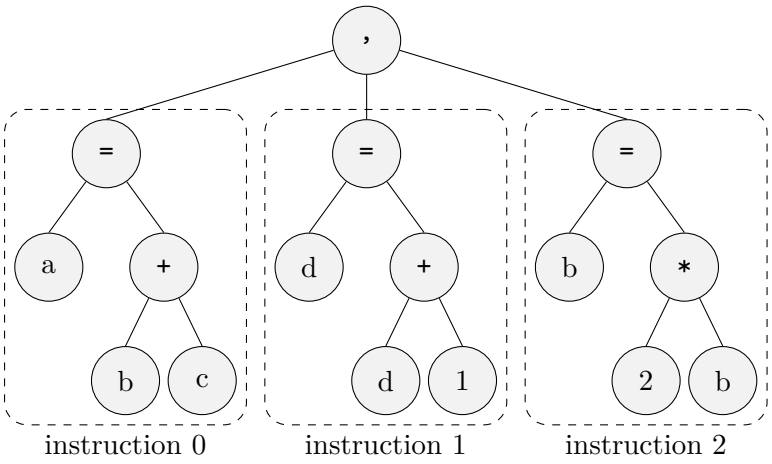


FIGURE 4.2 – ASA représenté par l'expression de l'extrait de code 4.7

4.4.2 Représentation des fonctions d'indice

Jusqu'à présent, les expressions, bien qu'utilisant des opérandes représentant des tableaux, n'utilisent pas l'opérateur d'accès à un élément (`operator[]`) : cela revient à utiliser implicitement $i \mapsto i$ comme fonction d'indice. Ainsi, dans les extraits présentés jusqu'ici, chaque accès aux opérandes `a`, `b`, `c` et `d` étaient équivalents à `a[i]`, `b[i]`, `c[i]` et `d[i]` respectivement. Rendre cette information explicite permet éventuellement d'améliorer la lisibilité du code, mais cela permet surtout de définir une fonction différente. Dans l'extrait de code 4.8, des indices explicites sont utilisés grâce au type `Index`. La représentation des fonctions d'indice est acquise au moyen d'un patron d'expression annexe à celui construit jusqu'ici (qui permet la représentation des instructions). Ce patron d'expression supporte un sous-ensemble d'opérations spécifiques, propres aux fonctions d'indice.

Afin de permettre l'analyse des expressions d'indice, les constantes numériques les composant doivent être connues durant la compilation. Au moment de la rédaction, la seule syntaxe permise

```
// a, b, c are operands
Index i;
auto e = (
  a[i] = a[i] * b[i],
  c[i] = c[i+ctv<1>]
);
```

(⩾ C++14)

Extrait de code 4.8 – Expression avec des indices explicites

par le langage pour atteindre cet objectif dans ce contexte est d'utiliser ces constantes numériques comme arguments d'un template. C'est à cela que sert le template `ctv` (pour *compile time value*) qui correspond à une variable utilisable dans une expression d'indice et dont le type comporte l'information utile. L'utilisation de fonctions (dans ce cas précis, d'opérateurs) `constexpr` ne suffit pas. Une telle fonction peut s'exécuter aussi bien durant la compilation que durant l'exécution du programme et ses paramètres sont donc considérés comme inconnus durant la compilation.

De nouveaux éléments arrivant en C++ (standard C++20) auraient pu apporter une solution : `consteval` et `std::is_constant_evaluated` [R. Smith et al. 2018b]. Le premier, `consteval`, bien que correspondant à une fonction ne pouvant être exécutée que durant la compilation du programme, ne permet pas l'utilisation de ses paramètres comme arguments d'un template. Il est possible que cette fonctionnalité fasse partie d'une future révision du standard. Le second, `std::is_constant_evaluated`, également apparu en C++20, correspond à une fonction, dont l'implémentation est spécifique au compilateur, déterminant si l'exécution en cours est effectuée par le compilateur ou non. Elle peut s'utiliser avec un `if constexpr` pour distinguer les deux cas possibles d'une fonction `constexpr`. Cependant, cela ne permet pas non plus l'utilisation des paramètres de la fonction comme arguments d'un template. [Revzin et al. 2020] fait par ailleurs remarquer que même les fonctions `consteval` ne sont alors pas permises et introduit le `if consteval`. Le cas des templates n'est en revanche pas abordé dans ce document (révision 2).

La figure 4.3 correspond à ce qui est représenté par cette nouvelle expression. Cet arbre inclut plusieurs patrons d'expression : un pour chaque occurrence de l'opérateur d'accès à un élément d'un tableau (`operator[]`) en plus de celui encodant l'expression globale. Toutes ces informations

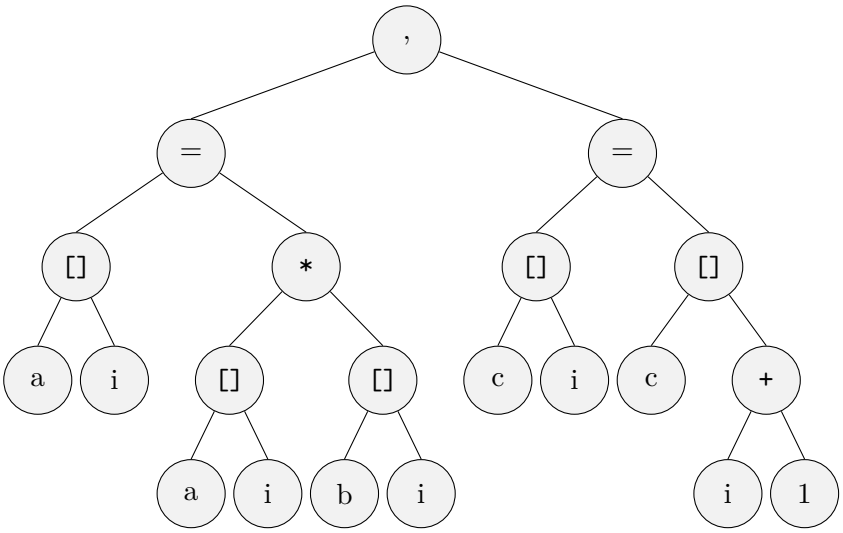


FIGURE 4.3 – ASA représenté par l'expression de l'extrait de code 4.8

permettent de connaître durant la compilation à la fois quelles variables sont utilisées et comment elles le sont.

Par exemple, un opérateur tel que celui effectuant une addition, ou celui effectuant une multiplication, ne modifie pas ses opérandes : celles-ci sont seulement lues. En revanche, l'opérande à gauche d'un opérateur d'affectation peut être modifié : il s'agit d'un accès en écriture. Ainsi, en parcourant cet arbre, il est possible de déduire $F_a = \{i \mapsto i\}$ donc $|F_a| = 1$, $F_b = \{i \mapsto i\}$ donc $|F_b| = 1$ et $F_c = \{i \mapsto i, i \mapsto i + 1\}$ donc $|F_c| = 2$. Plus précisément, $F_c^w = \{i \mapsto i\}$ et $F_c^r = \{i \mapsto i + 1\}$, ce qui invalide la condition de l'équation (4.8′) puisqu'il existe un élément de F_c^w $(i \mapsto i)$ et un élément de F_c^r $(i \mapsto i + 1)$ tels qu'ils produisent la même sortie pour deux entrées différentes (par exemple les entrées 2 et 1, dans cet ordre, pour lesquelles ces fonctions retournent toutes les deux 2). Cette information permet de déterminer que l'utilisation faite de la variable `c` empêche la parallélisation de ces instructions.

4.4.3 Propriétés des expressions d'indice

Les expressions d'indice peuvent être marquées d'un ensemble de propriétés afin d'aider le système à prendre des décisions. Nous présenterons comment ces propriétés peuvent être utilisées dans la section suivante, en particulier pour « injective » et « affine ».

Les propriétés peuvent être ajoutées sur une expression d'indice au moyen de fonctions dédiées à celles-ci. Il est par exemple possible de marquer une expression d'indice de la propriété « injective » (qui indique que la fonction d'indice représentée est déclarée injective par l'utilisateur pour les valeurs possibles en entrée) comme le montre l'extrait de code 4.9.

```
injective(i*i)
```

(⩾ C++14)

Extrait de code 4.9 – Application de la propriété d'injectivité sur une expression d'indice

Lorsque des propriétés sont présentes sur une expression utilisée comme opérande d'une fonction (par exemple dans `a + b` où `a` et `b` sont des expressions), la bibliothèque permet de calculer les propriétés de l'expression résultante. Le calcul considère les propriétés de chaque opérande et l'action que représente la fonction appliquée afin de générer de nouvelles propriétés. Ce mécanisme correspond à un ensemble de spécialisations d'un template.

Pour illustrer cela, considérons deux expressions respectivement marquées strictement croissante (par la fonction `strictinc`) et strictement décroissante (`strictdec`). Ces propriétés correspondent en pratique à trois propriétés : « monotone », « stricte » et respectivement « croissante » et « décroissante ». Lorsque l'on calcule la différence de ces deux expressions, l'expression produite conserve les propriétés « monotone » et « stricte », et obtient « croissante » ou « décroissante » selon l'ordre des opérandes. L'extrait de code 4.10 présente la construction d'une expression dont les opérandes sont marquées comme décrit ci-avant et dont l'expression résultante possède automatiquement les propriétés l'indiquant comme strictement croissante.

```
Index i;
auto e = strictinc(i) - strictdec(-i);
```

(⩾ C++14)

Extrait de code 4.10 – Expression ayant la propriété d'être strictement croissante déduite des propriétés de ses opérandes

En plus des propriétés ainsi acquises, une expression peut automatiquement obtenir des propriétés inférées à partir de celles qu'elle possède. À l'instar des propriétés calculées, les propriétés inférées sont produites par des fonctions définies par spécialisation d'un template. Celles-ci produisent un ensemble de nouvelles propriétés sachant un ensemble initial de propriétés. Par exemple, lorsque la propriété « monotone » est présente, s'il y a aussi « stricte », « injective » est déduite. Ainsi, l'expression `e` de l'extrait de code 4.10 possède également cette dernière propriété.

L'ASA construit pour les indices est plus contraint que celui qui représente les instructions dans leur ensemble. Les opérations permises sont moins nombreuses (on ne calculera pas la factorielle d'un indice, par exemple) et n'opèrent que sur des entiers relatifs. Les constantes sont toutes, elles aussi, entières, mais sont surtout connues durant la compilation. Grâce à ces contraintes, il est possible de procéder à certaines transformations sur ces arbres.

Le plus souvent, l'indice d'accès à un élément d'un tableau est une fonction affine de l'indice courant de la boucle, donc de la forme $i \mapsto a\,i + b$, où $a, b, i \in \mathbb{Z}$. La figure 4.4 montre les formes d'arbres considérées comme affines (c'est-à-dire que l'expression représentée est une fonction affine de la variable de boucle). Elles correspondent aux cas suivants : $a = 0$; $a = 1$ et $b = 0$; a quelconque et $b = 0$; $a = 1$ et b quelconque ; et enfin le cas général. Dans tous ces cas, la propriété « affine » est automatiquement ajoutée à une expression sans nécessiter un marquage explicite par le développeur. Cette propriété implique par ailleurs l'injectivité de la fonction.

Un problème de détection survient si le développeur écrit une expression complexe qui aurait pu être simplifiée en une expression trivialement affine, par exemple $i \mapsto 2 \times (i + 1)$ dont l'ASA ne fait pas partie des cas identifiés mais qui pourrait être transformée en $i \mapsto 2\,i + 2$. Cette remarque peut être étendue à des cas plus complexes tels que $i \mapsto (2\,i + 5) + 2 \times (i - 2)$ qui correspond plus simplement à $i \mapsto 4\,i + 1$.

Transformer une expression complète en une version simplifiée et reconnaissable suivant les motifs de la figure 4.4 est une opération difficile. Cependant, la construction d'une expression est effectuée par étape : les opérateurs sont traités selon un ordre dépendant de l'associativité et de la priorité de chacun. En conséquence, il suffit de maintenir, tant que l'expression est affine, une écriture simplifiée tout au long de la procédure.

Prenons en exemple la première fonction introduite ci-avant, $i \mapsto 2 \times (i + 1)$: la première étape est la construction de l'arbre représentant $i + 1$. Lorsque la multiplication intervient, les deux opérandes sont des expressions affines : on peut obtenir par *type traits* la valeur de leur a et b respectifs (ici 0 et 2 à gauche et 1 et 1 à droite). Il s'agit dans ce cas d'une multiplication : pour que le résultat soit une expression affine, il faut qu'au moins l'un des deux a des expressions opérandes soit nul. C'est validé ici, et les valeurs de a et b de l'expression résultante peuvent être calculées facilement. À partir de celles-ci, il est possible de générer un nouvel arbre représentant l'expression affine. En fonction des valeurs spécifiques de a et b, le motif le plus adapté est choisi parmi ceux de la figure 4.4.

Le comportement primitif des patrons d'expression fait qu'une expression comme $(a\,i + b) + (c\,i + d)$ produit l'arbre à droite dans la figure 4.5. Grâce à ce système, l'arbre est produit en suivant la règle représentée par la figure 4.6. La figure 4.7 schématise la règle utilisée pour la transformation dans le cas de la multiplication par un scalaire.

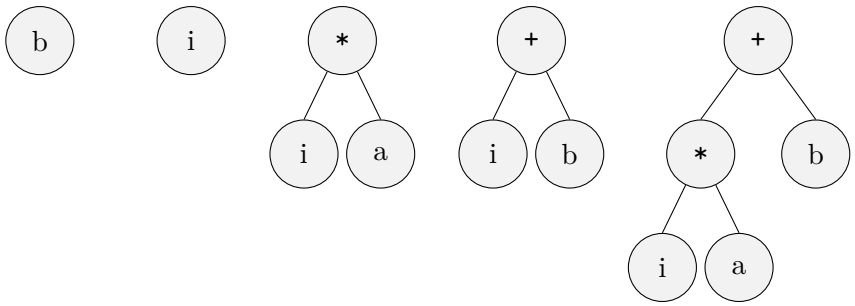


FIGURE 4.4 – ASA des expressions reconnues comme affines

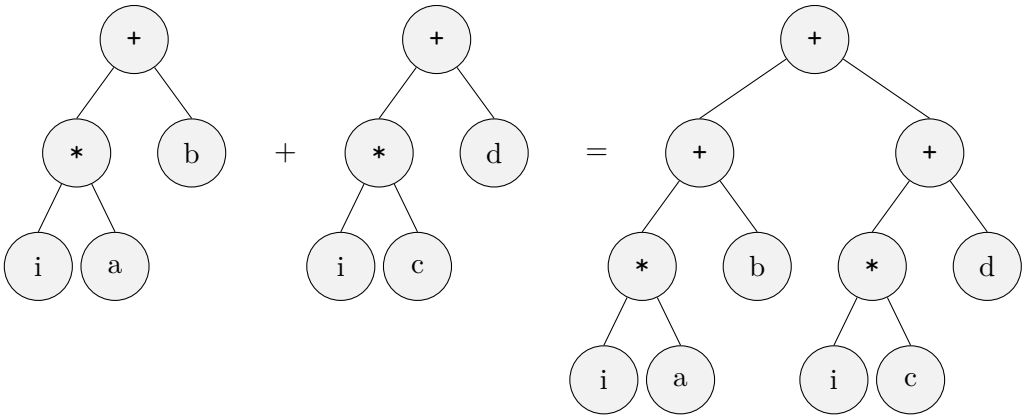


FIGURE 4.5 – Addition non optimisée de deux ASA d'expressions affines

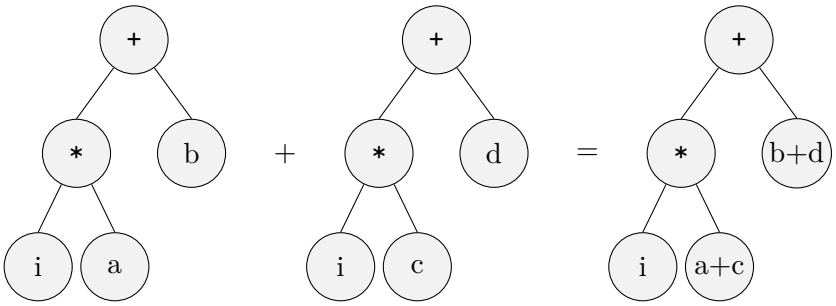


FIGURE 4.6 – Addition optimisée de deux ASA d'expressions affines

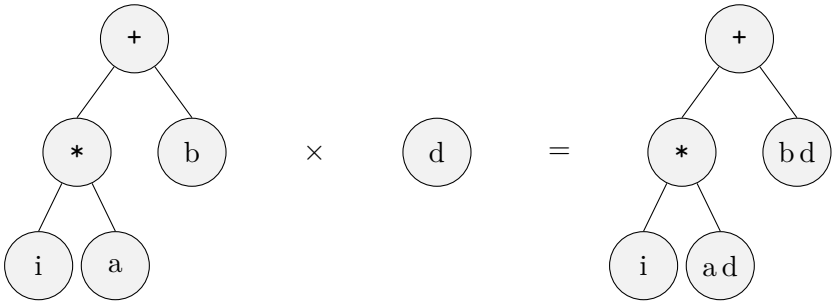


FIGURE 4.7 – Multiplication optimisée de deux ASA d'expressions affines

4.4.4 Intégration des fonctions

Le patron d'expression est initialement conçu au sein de la bibliothèque pour prendre en compte les opérateurs fournis par le langage. En effet, les opérateurs (de surcroît ceux qui peuvent être surchargés) sont en nombre fini : il est possible de tous les considérer. En revanche, ce n'est pas le cas lorsque l'on veut généraliser aux fonctions quelconques.

Or, un utilisateur peut souhaiter utiliser de telles fonctions au sein d'une expression, comme c'est le cas dans l'extrait de code 4.1 avec `std::pow`. Cette fonction ne peut pas être utilisée directement au sein d'une expression : les opérandes qui lui seraient transmises sont de types non numériques (puisque ce sont des opérandes de patron d'expression) et cette fonction n'est pas capable de retourner un type exprimant un ASA.

Il est donc nécessaire d'écrire une nouvelle fonction dont les paramètres et le type de retour sont compatibles avec le patron d'expression, et fournissant un mécanisme d'évaluation utilisant la fonction désirée. En se référant à la section 3.3.3.3 pour définir la fonction utilisable dans l'expression et à la section 3.3.3.2 pour la définition du type qui implémente l'appel effectif à la fonction cible, un développeur peut implémenter le nécessaire pour chaque nouvelle fonction.

Afin de simplifier cette procédure, la bibliothèque fournit un outil pour générer automatiquement ce qui est nécessaire et retourner un fonctionoïde [2] pouvant être utilisé dans une expression. Celui-ci repose sur une classe générique permettant de gérer l'appel sous-jacent à n'importe quel type de fonction d'arité quelconque.

Cette classe (extrait de code 4.11) doit conserver la fonction à appeler, contrairement à ce qui est habituellement fait avec les opérateurs pour lesquels chaque évaluateur connaît directement la fonction à exécuter. Les deux fonctions d'évaluation attendues (l'une sans indice, l'autre avec) sont implémentées en transmettant automatiquement tous les arguments à la fonction les évaluant pour utiliser leur valeur effective comme argument (voir la section 2.12 sur la transmission parfaite).

Afin d'éviter une éventuelle ambiguïté, la fonction d'évaluation qui n'accepte pas comme premier paramètre un entier s'assure d'être considérée pour résoudre l'appel uniquement lorsque ses arguments sont des expressions. Pour cela, la SFINAE (*Substitution Failure Is Not An Error*) est utilisée à l'aide de `std::enable_if_t` (voir la section 2.7).

```
template<typename F>
struct UserOperator {
  F f;

  template<typename... Args, std::enable_if_t<allExpression<Args...>>* = nullptr>
  inline decltype(auto) eval(Args&&... args) {
    return f(std::forward<Args>(args).eval()...);
  }

  template<typename... Args>
  inline decltype(auto) eval(std::size_t i, Args&&... args) {
    return f(std::forward<Args>(args)[i]...);
  }
};
```

(⩾ C++14)

Extrait de code 4.11 – Évaluateur générique pour les extensions par l'utilisateur

2. https://isocpp.org/wiki/faq/pointers-to-members#functionoids

L'extrait de code 4.12 montre la fonction produisant une nouvelle fonction, compatible avec les expressions de la bibliothèque, elle-même retournant une expression dont l'opérateur est une instance de l'évaluateur générique présenté ci-avant.

Il n'est pas nécessaire de s'assurer au préalable que les paramètres de la fonction générée sont effectivement des expressions : elle est conçue pour être spécifiquement utilisée dans ce contexte, contrairement aux opérateurs. En revanche, si un argument n'est pas une expression, il faut le transformer. Le template `Expression` est donc instancié avec les types des arguments transformés en expression si nécessaire par `AsExpression`.

```
template<typename F>
auto makeOperator(F&& f) {
  return [f = std::forward<F>(f)](auto&&... args) {
    using Expr = Expression<
      UserOperator<decltype(f)>,
      AsExpression<std::remove_reference_t<decltype(args)>>...
    >;
    return Expr{{f}, decltype(args)(args)...};
  };
}
```

(⩾ C++14)

Extrait de code 4.12 – Fonction générant une fonction compatible avec les expressions à partir d'une fonction quelconque

Cet outil permet de créer aisément de nouvelles fonctions comme présenté dans l'extrait de code 4.13. La première indique que l'on crée localement `factorial` à partir de la fonction portant le même nom (mais issue de l'espace de noms global pour éviter la confusion, d'où le préfixe `::`). La deuxième procède à partir d'une fonction lambda. Enfin, la troisième montre comment faire lorsqu'il existe des surcharges : il est nécessaire d'effectuer une conversion de type explicite, normalement dénotée par un `static_cast` dans ce cas. Puisque le premier paramètre template de la fonction `makeOperator` est le type de la fonction acceptée en argument, il est possible de le spécifier afin de simplifier l'écriture de la conversion.

```
auto factorial = makeOperator(::factorial);
auto add3 = makeOperator([](auto a, auto b, auto c) { return a+b+c; });
auto pow = makeOperator<float(&)(float, float)>(std::pow);
```

(⩾ C++14)

Extrait de code 4.13 – Génération de fonctions utilisables dans une expression

Si l'on reprend le code introduit au début de ce chapitre (extrait de code 4.1), on peut donc écrire les instructions du corps de la boucle comme dans l'extrait de code 4.14, en utilisant cette fonction générée `pow`.

À noter que la ligne 6 est permise grâce à une transformation automatique d'une expression d'indice en expression générale. Dans certains cas, il peut être utile de forcer cette encapsulation (une fonction est fournie pour cet usage) par exemple pour permettre l'utilisation de valeurs non connues à la compilation avec un indice (comme dans `i*2`, qui nécessite donc l'encapsulation `xpr(i)*2`).

```
// a, b, c, d, e and f are expression template operands
Index i;
auto expr = (
  a[i] = a[i] * b[i],
  c[i] = c[i+ctv<1>] + d[i],
  b[i] = b[i] + i,
  d[i] = pow(c[i], e[i]),
  f[i*i] = 2 * f[i*i]
);
```

Extrait de code 4.14 – Expression des instructions du corps de la boucle de l'extrait de code 4.1

4.5 Analyse sémantique et génération du programme

L'objectif est de générer un programme correct à partir du patron d'expression en entrée consistant en l'encodage d'une boucle exécutant une série d'instructions, le patron d'expression étant tel que décrit dans la section précédente. Le programme correct consiste, au maximum, en deux boucles : une dont l'exécution doit être séquentielle, l'autre dont l'exécution peut être parallèle. Pour cela, il s'agit dans un premier temps de regrouper les instructions interdépendantes d'une même itération. Il est alors possible de tester, pour chaque groupe d'instructions, s'il est possible de procéder à une exécution parallèle. Enfin, il faut générer la ou les boucles en conséquence. Cette section présente ces trois étapes principales dans cet ordre.

4.5.1 Groupement des instructions

Nous appliquons au cours de cette section les conditions de Bernstein présentées dans la section 4.3.1 afin de créer des ensembles d'instructions dépendantes tels que ces ensembles sont indépendants entre eux. Pour rappel, l'information dont nous disposons est un patron d'expression qui liste les instructions qu'exécutera la boucle, obtenu comme expliqué dans la section 4.4.

Ce patron d'expression est traité comme une liste d'expressions (les arbres fils de la racine représentée par l'opérateur virgule, à défaut de cette racine, l'expression complète est la seule de la liste). Une première étape consiste en la création, à partir de cette liste d'expressions, d'une liste d'informations les concernant : le mode d'accès à chaque variable que l'expression emploie.

Une première métafonction prend donc en entrée une liste d'expressions pour produire une liste de triplets de la forme (id_e, W, R) où id_e est l'identifiant de l'expression, et où W et R sont des n-uplets contenant respectivement les variables auxquelles on accède en écriture et celles auxquelles on accède en lecture. Par exemple, le résultat obtenu à partir de l'expression définie dans l'extrait de code 4.14 est représenté par l'équation (4.11).

$$\begin{Bmatrix} (0, (a), (a, b)), \\ (1, (c), (c, d)), \\ (2, (b), (b)), \\ (3, (d), (c, e)), \\ (4, (f), (f)) \end{Bmatrix} \tag{4.11}$$

Sur cette information, il est possible d'appliquer l'algorithme 4.1 présenté dans la section 4.3.1. Pour chaque instruction, le test d'indépendance est effectué avec les ensembles déjà construits en utilisant les n-uplets W et R.

Le résultat de cette opération est l'ensemble des groupes d'instructions (c'est-à-dire des ensembles) interdépendantes représentées par leur triplet (id_e, W, R). Ainsi, pour l'exemple de l'équation (4.11), nous avons l'ensemble de l'équation (4.12).

$$\begin{Bmatrix} \{(0,(a),(a,b)),(2,(b),(b))\}, \\ \{(1,(c),(c,d)),(3,(d),(c,e))\}, \\ \{(4,(f),(f))\} \end{Bmatrix} \tag{4.12}$$

Les informations W et R étant alors inutiles, une métafonction permet de construire l'ensemble ne comportant que les identifiants id_e des expressions qui seront ensuite utilisés pour reconstruire un arbre lorsque la possibilité de paralléliser chaque groupe aura été déterminée. Ce que retourne cette dernière métafonction pour l'exemple de l'équation (4.12) est présenté dans l'équation (4.13).

$$\begin{Bmatrix} \{0,2\}, \\ \{1,3\}, \\ \{4\} \end{Bmatrix} \tag{4.13}$$

4.5.2 Test de parallélisabilité

Dans cette section, nous présentons l'application des conditions présentées dans la section 4.3.2. Il faut pour cela associer à chaque opération un mode d'accès pour chacun des opérandes. Le système est par conception extensible : les opérateurs fournis par la bibliothèque permettent un ensemble de constructions possibles, mais un développeur peut fournir des fonctions supplémentaires. Par exemple, il est possible de créer la fonction `sin` permettant la création d'un nœud dans l'ASA correspondant à une fonction unaire, dont l'exécution sera de calculer le sinus de l'argument qui lui est transmis. Pour cela, nous utilisons donc un système de *type traits* qui associe au couple (Opération, Indice d'opérande) un mode d'accès, lecture ou écriture. Grâce à cela, il est possible de parcourir durant la compilation l'ASA construit par le patron d'expression et d'appliquer une transformation afin d'associer à chaque feuille un mode d'accès.

Afin de vérifier exactement les conditions décrites par les équations (4.8′) et (4.9′), il serait nécessaire de vérifier pour chaque itération de la boucle que les indices utilisés ne causent pas de problème. Cette solution peut être implémentée si le n-uplet des indices des itérations de la boucle est connu. Pour exemple, il est possible de considérer le cas trivial montré dans l'extrait de code 4.15. Dans ce code, sans savoir que `i` prend ses valeurs dans $\{0,2,4,6,8\}$, les accès semblent indiquer que l'instruction exécutée par cette boucle ne peut pas l'être en parallèle.

Il est possible de déduire ce n-uplet à partir de la valeur initiale, du pas et de la condition de terminaison, ce qui nécessite de connaître ces informations dès la compilation. Nous voulons proposer une solution qui n'apporte pas une contrainte si forte : il est très commun d'utiliser une

```
for(int i = 0; i < 10; i += 2)
  a[i] = a[i+1];
```

(⩾ C++98)

Extrait de code 4.15 – Boucle nécessitant de connaître les indices pour déterminer qu'elle peut être exécutée en parallèle

boucle pour traiter les éléments d'une collection (ce que nous faisons dans nos exemples) dont la taille peut n'être connue que durant l'exécution du programme. La bibliothèque peut cependant fournir une interface alternative bénéficiant de cette contrainte lorsque cela est applicable.

Sans cette connaissance, il n'est pas possible de produire un algorithme ayant une sensibilité parfaite (correctement détecter qu'un code peut être exécuté en parallèle) sans affecter la spécificité (correctement détecter qu'un code ne peut pas être exécuté en parallèle). Pour qu'un système de parallélisation automatique soit utilisable, un faux positif (mauvaise spécificité) ne peut être permis, puisqu'il causerait la génération d'un programme dont le comportement serait incorrect. En admettant certaines inconnues (par exemple ce n-uplet d'indices), il nous a semblé impossible d'éviter automatiquement l'existence de faux négatifs sans risquer de produire des faux positifs. Sans connaissance *a priori* du pas, dans l'exemple donné par l'extrait de code 4.15 nous devons supposer qu'il est impossible de paralléliser cette boucle. En utilisant l'hypothèse contraire, lorsque le pas est de 1 la déduction serait erronément positive.

Nous avons en conséquence proposé [Pereda et al. 2018] une condition (équation (4.14)), conçue à partir des conditions exactes, spécifique mais non parfaitement sensible, afin de déterminer la capacité d'une expression à être exécutée en parallèle :

$$F_a^w = \emptyset \vee |F_a| = 1. \tag{4.14}$$

Ce test est particulièrement limité : soit aucun accès en écriture n'est fait sur les éléments du tableau `a`, soit les accès sont effectués en utilisant une seule et unique fonction d'indice. Il repose en outre, lorsque $F_a^w \neq \emptyset$, sur l'hypothèse que cette fonction est injective pour le domaine utilisé. Dans le cas contraire, par exemple avec $i \mapsto i^2$ et $i \in \{-1, 0, 1\}$, il est possible d'accéder à une même variable à partir de deux itérations distinctes malgré l'unicité de la fonction.

La création de l'image de la fonction (nécessitant une énumération à partir du domaine) n'étant pas envisageable pour des raisons de performances, voire de capacité, ces cas particuliers doivent être traités de manière spécifique. Ainsi, ce test peut être appliqué lorsque l'expression d'indice possède la propriété « injective », que ce soit dû à un marquage explicite de l'utilisateur ou issu d'une déduction automatique. En cas d'erreur dû à un marquage explicite incorrect, la responsabilité incombe au développeur qui doit alors vérifier la fonction ou son domaine.

Depuis cette proposition, nous avons amélioré la sensibilité du test par analyse des fonctions d'indice pour le cas polynômial de degré 1. En effet, les indices d'accès ainsi que les différents coefficients employés pour construire les expressions d'indice sont des entiers. En conséquence, ces expressions, lorsqu'elles sont polynômiales et de degré 1, correspondent à des fonctions affines entières. Il est donc possible de déterminer par analyse s'il existe deux entrées distinctes permettant de produire la même sortie en considérant un ensemble de fonctions de ce type.

Deux de ces fonctions peuvent donc prendre la forme générale $i \mapsto a\,i + b$ et $i \mapsto c\,i + d$, où $i, a, b, c, d \in \mathbb{Z}$. Le mécanisme de détection expliqué dans la section 4.4.2 garantit que toute expression affine est représentée sous cette forme, éventuellement avec un coefficient nul.

Considérer ces fonctions nous permet d'éviter un faux négatif qu'aurait donné le test initial

pour des fonctions telles que $i \mapsto 2\,i$ et $i \mapsto 2\,i + 1$. Les images de celles-ci ont une intersection vide pour $i \in \mathbb{Z}$, ainsi, bien que ce soit les images de deux fonctions distinctes (faisant échouer $|F_a| = 1$), même en utilisant l'une pour un accès en lecture et l'autre pour un accès en écriture (faisant échouer $F_a^w = \emptyset$), il faut déterminer qu'elles n'empêchent pas la parallélisation.

Une expression utilisant deux fonctions affines pour accéder aux éléments d'un même tableau sera toujours parallélisable si l'équation (4.15) est respectée, c'est-à-dire s'il n'est pas possible d'obtenir le même résultat avec les deux fonctions pour des indices différents :

$$\forall i, j \in \mathbb{Z}, i \neq j,\ a\,i + b \neq c\,j + d, \qquad a, b, c, d \in \mathbb{Z}. \tag{4.15}$$

La contrainte $i \neq j$ correspond à l'objectif de comparer les accès sur deux itérations distinctes d'une boucle. Pour vérifier les contraintes données par l'équation (4.15), il est possible d'utiliser une équation Diophantienne affine à deux inconnues dont les coefficients sont a, c et $d - b$ (équation (4.16)) :

$$a\,i - c\,j = d - b, \qquad i, j \in \mathbb{Z}. \tag{4.16}$$

La résolution de cette équation lorsqu'elle est utilisée pour $a = c$ et $d = b$ donne la solution triviale $i = j$ et correspond au cas exclu par la contrainte $i \neq j$. Cependant, cette contrainte peut être mise de côté puisque lorsque les fonctions à comparer correspondent en réalité à une même fonction, on ne cherchera pas à résoudre cette équation Diophantienne.

L'inexistence de solutions à cette équation prouve donc que l'expression peut être exécutée en parallèle. Pour tester cela, nous utilisons le théorème de Bachet-Bézout, lequel indique l'existence de solutions si $d - b$ est un multiple du PGCD (plus grand commun diviseur) de a et c.

Grâce au travail présenté dans la section 4.4.2 sur la représentation des expressions affines, il est possible, durant la compilation, de vérifier cette propriété dans le cas où les fonctions sont effectivement affines, sinon d'utiliser par défaut le test moins sensible évoqué précédemment.

Ce résultat s'étend encore lorsque l'on peut connaître le pas de la boucle durant la compilation. Si la valeur initiale et la condition d'arrêt sont des informations habituellement dépendantes de valeurs dynamiques (taille d'une collection par exemple), le pas est en revanche souvent connu. Grâce à cette information, il est possible de détecter correctement que le cas présenté dans l'extrait de code 4.15 dont les fonctions sont $i \mapsto i$ (en écriture) et $i \mapsto i + 1$ (en lecture) peut être parallélisé. Pour utiliser cette information, nous transformons la fonction initiale de sorte à en obtenir une nouvelle fonction équivalente mais pour laquelle le pas serait de 1.

Considérons une fonction affine quelconque $a\,I + b$ utilisée comme fonction d'indice au sein d'une boucle dont l'indice varie de sa valeur initiale v_i jusqu'à sa valeur finale v_f exclue par un pas de p. L'image de cette fonction est générée par l'équation (4.17) :

$$\{a\,I + b \mid I \in [\![v_i, v_f[\![, I = v_i + k\,p, k \in \mathbb{N}\}, \qquad a, b, v_i, v_f \in \mathbb{Z}, p \in \mathbb{Z}^*. \tag{4.17}$$

Il est important de noter que l'on a $v_i \leqslant v_f$. Ainsi, lorsque $p < 0$, v_i correspond à la valeur finale plutôt qu'initiale et, à l'inverse, v_f correspond à la valeur initiale plutôt que finale. Cette inversion sémantique conditionnelle suffit à généraliser pour $p \in \mathbb{Z}^*$ au lieu de limiter le pas aux entiers positifs.

En posant $I = i_p + v_i$, où i_p est la nouvelle variable servant d'indice, on peut décrire la même

image en utilisant une valeur initiale de 0 comme dans l'équation (4.18) :

$$\{a\,(i_p + v_i) + b \mid i_p \in p\,\mathbb{N}, i_p < v_f - v_i\}, \qquad a, b, v_i, v_f \in \mathbb{Z}, p \in \mathbb{Z}^*. \tag{4.18}$$

Enfin, en posant $i_p = p\,i$, où i est la nouvelle variable servant d'indice, la description de cette image évolue à nouveau et utilise un nouveau pas de 1 en addition de la valeur initiale de 0. La nouvelle description correspond à l'équation (4.19) :

$$\{a\,(p\,i + v_i) + b \mid i \in \mathbb{N}, i < \left\lceil \frac{v_f - v_i}{p} \right\rceil\}, \qquad a, b, v_i, v_f \in \mathbb{Z}, p \in \mathbb{Z}^*. \tag{4.19}$$

En utilisant cette transformation pour deux fonctions $a\,I + b$ et $c\,J + d$ avec une valeur initiale v_i, une valeur finale v_f et un pas p, on cherche alors à résoudre l'équation (4.20) :

$$\begin{aligned} & a\,p\,i + a\,v_i + b = c\,p\,j + c\,v_i + d \\ \iff \qquad & a\,p\,i - c\,p\,j = d - b + (c - a)\,v_i. \end{aligned} \tag{4.20}$$

Lorsque le terme $A = (c - a)\,v_i$ n'est pas insignifiant dans la résolution de l'équation, la valeur initiale v_i devient nécessaire. Il nous faut donc déterminer quels critères permettent d'ignorer A. Puisque cette équation ne sera utilisée que pour tester l'existence de solutions, c'est la divisibilité de la partie droite $(d - b + A)$ par le PGCD de $a\,p$ et $c\,p$ qui nous intéresse. Ainsi, A peut être ignoré lorsqu'il est divisible par le PGCD de $a\,p$ et $c\,p$.

Le PGCD de $a\,p$ et $c\,p$ vaut $p \times \mathrm{pgcd}(a, c)$. Or, le PGCD de a et c divise leur différence $(c - a)$. Ainsi, il reste à savoir si v_i est un multiple de p ou non. Ce résultat permet de montrer que v_i peut être ignoré lorsque $p = 1$. On retrouve alors l'équation (4.16).

Lorsque $p \neq 1$ en revanche, nous utilisons l'équation (4.20), c'est-à-dire les coefficients $a\,p$, $c\,p$ et $d - b + (c - a)\,v_i$. Lorsque le pas et la valeur initiale sont connus, les fonctions d'indice affines $a\,i + b$ sont donc mises à jour pour le test en utilisant $A = a\,p$ et $B = b + a\,v_i$.

Dans le cadre de cette thèse, nous avons donc plusieurs tests. Pour déterminer celui qui peut être appliqué, l'arbre construit par le patron d'expression est d'abord transformé en une liste par un parcours en profondeur (de type NLR), en appliquant une transformation qui associe aux feuilles un triplet (id_v, rw, i) comportant l'identifiant de la variable, id_v, le mode d'accès rw (soit lecture, r, soit écriture, w) et le patron d'expression représentant la fonction d'indice i. Les branches de l'arbre ne sont pas représentées dans cette liste.

Par exemple, si l'on considère les instructions des lignes 5 et 7 de l'extrait de code 4.14, qui sont interdépendantes (voir l'équation (4.13)), l'équation (4.21) représente le résultat de la transformation expliquée.

$$\begin{aligned} \{ & \\ & (c, w, i \mapsto i), \\ & (c, r, i \mapsto i + 1), \\ & (d, r, i \mapsto i), \\ & (d, w, i \mapsto i), \\ & (c, r, i \mapsto i), \\ & (e, r, i \mapsto i) \\ \} & \end{aligned} \tag{4.21}$$

Jusqu'à ce que l'ensemble soit vide, on en retire tous les éléments référant à une même variable pour les isoler dans un nouvel ensemble à traiter, et on applique cette procédure récursivement sur l'ensemble restant.

En isolant par exemple e, on obtient deux ensembles $\{(e, r, i \mapsto i)\}$ et $\{(c, w, i \mapsto i), (c, r, i \mapsto i+1), (d, r, i \mapsto i), (d, w, i \mapsto i), (c, r, i \mapsto i)\}$. Le premier, ne contenant que des triplets à propos de la variable e, peut alors être traité pour vérifier s'il empêche la parallélisation. Toutes les expressions d'indice concernées (c'est-à-dire $\{i \mapsto i\}$) ayant la propriété d'être affines, le test appliqué est celui spécifique aux fonctions affines. S'il n'avait pu être utilisé, dans ce cas précis le test moins sensible (équation (4.14)) aurait tout de même été correct puisque l'on a $F_e^w = \emptyset$.

L'étape suivante isolera par exemple d, afin d'obtenir les ensembles $\{(d, r, i \mapsto i), (d, w, i \mapsto i)\}$ et $\{(c, w, i \mapsto i), (c, r, i \mapsto i+1)\}$. Le premier ensemble permet à nouveau, à partir du seul test simplifié, de déterminer que d ne cause pas de problème : bien que $F_d^w \neq \emptyset$, on a $|F_d| = 1$ puisque la seule fonction est $i \mapsto i$. À l'instar du cas précédent, cependant, dans la pratique il s'agit encore du test spécifique aux fonctions affines qui sera appliqué ici.

Enfin, en isolant les triplets correspondants à la variable c, on obtient l'ensemble $\{(c, w, i \mapsto i), (c, r, i \mapsto i+1), (c, r, i \mapsto i)\}$ et l'ensemble vide, il s'agit donc de la dernière étape. Pour ce cas, le test simplifié affirmerait que les instructions ne doivent pas être exécutées en parallèle à cause des accès effectués à la variable c. Les fonctions $i \mapsto i$ et $i \mapsto i+1$ étant toutes deux affines, le test spécifique est utilisé dans ce cas.

En observant les autres instructions de l'extrait de code 4.14, en particulier la ligne 8, on observe un cas pour lequel toutes les fonctions ne sont pas affines. Si le développeur précise pour chacune des fonctions qu'elle est injective (en écrivant `injective(i*i)`), ce qui est correct par exemple si i prend ses valeurs dans $\mathbb{N}$, alors le test de l'équation (4.14) peut être utilisé. Dans ce cas précis, $|F_f| = 1$ donc l'instruction peut être exécutée en parallèle.

En revanche, si au moins une expression n'est ni affine, ni injective, aucun test ne peut être effectué et, en l'état, le groupe d'instructions concernées est considéré comme ne pouvant pas être exécuté en parallèle.

4.5.3 Génération du programme

Chaque ensemble d'instructions interdépendantes peut être testé pour sa parallélisabilité. Une fois cette propriété déterminée, il est possible de les réunir en deux groupes : l'un contenant les instructions parallélisables et l'autre contenant les instructions non parallélisables.

Pour chaque groupe, le patron d'expression correspondant est reconstruit, les instructions sont associées à un nœud père commun représentant une séquence d'exécution (l'opérateur virgule). Il est possible de créer un patron d'expression vide si un groupe est vide (si toutes les instructions peuvent, ou à l'inverse aucune d'elles ne peuvent, être exécutées en parallèle). Aucun code n'est généré pour un tel patron d'expression.

Cette étape est accomplie par l'interface principale de la bibliothèque, la fonction `parallelFor`. L'extrait de code 4.16 présente son utilisation pour l'exemple initial de l'extrait de code 4.1. La fonction accepte en premier argument un élément définissant l'évolution de l'indice : dans cet exemple, i variera de 0 à n (exclu) par un pas, par défaut, de 1. Pour choisir un pas p, il est possible d'écrire `Range{0, n, p}`.

```
// a, b, c, d, e and f are expression template operands
Index i;
parallelFor(Range{0, n},
  a[i] = a[i] * b[i],
  c[i] = c[i+ctv<1>] + d[i],
  b[i] = b[i] + i,
  d[i] = pow(c[i], e[i]),
  f[injective(i*i)] = 2 * f[injective(i*i)]
);
```

(⩾ C++14)

Extrait de code 4.16 – Boucle de l'extrait de code 4.1 utilisant `parallelFor`

Utilisant ce qui est expliqué dans ce chapitre, la fonction `parallelFor` produit dans ce cas deux boucles. Pour le patron d'expression issu du groupe d'instructions non parallélisables, le code généré correspond directement à une boucle `for` dont chaque itération exécute l'expression avec pour argument la valeur de l'itérateur courant tel que le présente l'extrait de code 4.17.

```
for(int i = 0; i < n; ++i) {
  c[i] = c[i+1] + d[i];
  d[i] = std::pow(c[i], e[i]);
}
```

(⩾ C++98)

Extrait de code 4.17 – Boucle séquentielle générée par l'extrait de code 4.16

Pour le patron d'expression résultant du groupe d'instructions parallélisables, un traitement spécifique est appliqué. L'implémentation par défaut fournie par notre bibliothèque repose sur l'utilisation de la bibliothèque OpenMP, en l'occurrence en ajoutant une directive indiquant au compilateur qu'il faut exécuter les itérations de la boucle en parallèle. En conservant ce comportement, le code généré peut correspondre à l'extrait de code 4.18.

```
#pragma omp parallel for
for(int i = 0; i < n; ++i) {
  a[i] = a[i] * b[i];
  b[i] = b[i] + i;
  f[i*i] = 2 * f[i*i];
}
```

(⩾ C++98)

Extrait de code 4.18 – Boucle parallèle générée par l'extrait de code 4.16 en utilisant OpenMP

La méthode de parallélisation employée peut être sélectionnée par le développeur (il est possible d'étendre la bibliothèque en implémentant d'autres méthodes) au moyen d'un argument optionnel à fournir lors de l'appel à `parallelFor`. Par exemple, prenons une version faisant directement usage des *threads* telle que dans l'extrait de code 4.20. Deux spécialisations sont fournies : l'une pour le cas séquentiel (le booléen est faux), l'autre pour le cas parallèle (le booléen est vrai).

L'extrait de code 4.19 montre l'utilisation de `parallelFor` avec une implémentation sous-jacente de la parallélisation grâce aux *threads*.

```
// a, b, c, d, e and f are expression template operands
Index i;
parallelFor<ForLoopThread>(Range{0, n},
  a[i] = a[i] * b[i],
  c[i] = c[i+ctv<1>] + d[i],
  b[i] = b[i] + i,
  d[i] = pow(c[i], e[i]),
  f[injective(i*i)] = 2 * f[injective(i*i)]
);
```

(⩾ C++14)

Extrait de code 4.19 – Utilisation de la stratégie de parallélisation par `std::thread`

Il est possible d'utiliser une alternative à `Range` pour indiquer un pas connu durant la compilation : `RangeCT<begin, step>{end}`. S'il est employé, `parallelFor` utilise automatiquement la valeur initiale et le pas fourni pour améliorer la détection de la parallélisabilité du code.

La spécialisation utilisée pour le cas séquentiel est librement définie par le développeur afin de permettre des variations dans ce cas également, et non seulement lorsque le code peut être exécuté en parallèle. Un exemple concret d'utilisation est le déroulage de boucles [Davidson et Jinturkar 1995]. Cela consiste en la transformation d'une boucle de n itérations en une boucle de k itérations, où k est plus petit que n. Chaque itération de la nouvelle boucle procède à l'exécution de $\lfloor \frac{n}{k} \rfloor$ itérations de la boucle d'origine. Lorsque k ne divise pas n, il existe un reste entier r tel que $n = k \lfloor \frac{n}{k} \rfloor + r$. Ces r instructions peuvent alors être exécutées dans un bloc après la boucle.

```cpp
template<bool, typename, typename> struct ForLoopThread;

template<typename E, typename Range>
struct ForLoopThread<false, E, Range> {
  static void eval(Range const& range, E e) {
    if(range.step() > 0)
      for(auto it = +range.begin(); it < range.end(); it += range.step()) e[it];
    else
      for(auto it = +range.begin(); it > range.end(); it += range.step()) e[it];
  }
};

template<typename E, typename Range>
struct ForLoopThread<true, E, Range> {
  using Index = typename Range::ValueType;

  static void eval(Range const& range, E e) {
    using SeqRange = decltype(makeRange(+range.begin(), +range.end(),
    ↪  +range.step()));
    auto const& sequence = &ForLoopThread<false, E, SeqRange>::eval;

    Index const count = (range.end() - range.begin()
      + (range.step()-(range.step() > 0? +1 : -1)))/range.step();
    std::size_t const nThreads =
      std::min<decltype(nThreads)>(ParallelForParameters::nThreads, count);

    std::vector<std::thread> threads(nThreads-1);
    for(std::size_t k = 0; k < nThreads-1; ++k) {
      auto lRange = makeRange(
        range.begin() + static_cast<Index>(k*range.step()*count/nThreads),
        range.begin() + static_cast<Index>((k+1)*range.step()*count/nThreads),
        +range.step()
      );
      threads[k] = std::thread{sequence, lRange, e};
    }

    {
      auto lRange = makeRange(
        range.begin() +
        ↪  static_cast<Index>((nThreads-1)*range.step()*count/nThreads),
        range.end(),
        range.step()
      );
      sequence(lRange, e);
    }

    for(auto&& thread: threads) thread.join();
  }
};
```

(≥ C++14)

Extrait de code 4.20 – Stratégie de parallélisation par `std::thread`

4.6 Performances

Cette section présente des mesures de temps de compilation et d'exécution de programmes utilisant la bibliothèque présentée dans ce chapitre. Toutes les mesures ont été effectuées sur une machine dotée d'un Intel Xeon E7-8890 v3, cadencé à 2,5 GHz et ayant 72 cœurs physiques (18 processeurs ayant chacun 4 cœurs physiques). Les programmes sont compilés en utilisant GCC (g++) 8.2.0 avec notamment le pack d'optimisations `O2`. Les valeurs données dans ce document sont obtenues par une moyenne sur 20 exécutions (du programme pour les temps d'exécution, du compilateur et de l'éditeur de liens lorsqu'il s'agit du temps de compilation). Lorsque cela est pertinent (suffisamment visible), l'intervalle de confiance à 99 % est affiché. La compilation n'est pas exécutée en parallèle et n'utilise donc qu'un cœur. Pour ce qui est des exécutions, l'affinité du processus a été réglée afin qu'au maximum un cœur soit utilisé pour chaque processeur. Cela limite donc à 18 cœurs pour protéger d'un biais de mesure potentiel dû à l'utilisation simultanée de plusieurs cœurs d'un même processeur.

Pour permettre une utilisation de cette bibliothèque en pratique, il faut que celle-ci n'induise pas un surcoût en temps de compilation trop important. De par sa nature de bibliothèque active, il est inévitable d'observer des temps de compilation plus longs car la bibliothèque agit durant cette phase. La métaprogrammation template est par ailleurs connue pour son effet significatif sur ces temps, bien que celui-ci soit réduit au fur et à mesure des évolutions des compilateurs.

Afin d'évaluer ces temps de compilation, nous avons généré plusieurs ensembles de codes source qui utilisent la bibliothèque, en suivant différentes contraintes. Nous avons évalué l'effet de la variation du nombre de boucles utilisées et celui de la variation du nombre d'instructions que doit traiter une même boucle. Dans les deux cas, le nombre de variables créées est fixé à 100.

De plus, l'accès aux variables peut se faire de plusieurs manières. D'abord, les fonctions d'indice utilisées peuvent être soit identiques pour tous les accès (étiquette « fixed »), soit différentes pour tous les accès auquel cas nous utilisons des fonctions d'indice affines dont les coefficients sont aléatoires (étiquette « rand »). Ensuite, au sein d'une même boucle, les variables peuvent être dépendantes les unes avec les autres (étiquette « dep ») ou non (étiquette « indep »). Cela fait donc 4 cas à distinguer :

— « dep/fixed » : des instructions interdépendantes dont les fonctions d'indice sont identiques ;
— « indep/fixed » : des instructions indépendantes dont les fonctions d'indice sont identiques ;
— « dep/rand » : des instructions interdépendantes dont les fonctions d'indice sont différentes ;
— « indep/rand » : des instructions indépendantes dont les fonctions d'indice sont différentes.

La figure 4.8 montre les temps de compilation lorsque le nombre de boucles varie de 10 à 100 par pas de 10. Le nombre d'instructions est alors invariable et défini à 100. Pour maintenir le nombre d'instructions à 100 pour n boucles, en posant la division euclidienne $100 = q \times n + r$, on fait exécuter q instructions à $n - r$ boucles et, si $r > 0$, $q + 1$ instructions à r boucles.

On observe globalement une croissance lente du temps de compilation quelles que soient les caractéristiques des instructions. Le surcoût initial de quelques secondes est dû à l'emploi de la métaprogrammation template : dès lors qu'au moins une boucle est utilisée, la bibliothèque instancie de nombreux templates, imposant donc du travail au compilateur.

On observe également une influence notable, en particulier, de l'identité des fonctions d'indice. Procéder à l'analyse des instructions si celles-ci utilisent de nombreuses fonctions d'indice différentes est logiquement plus coûteux. Cependant, cet effet est volontairement exagéré par la configuration des codes sources générés : pour les 100 instructions, plusieurs centaines de

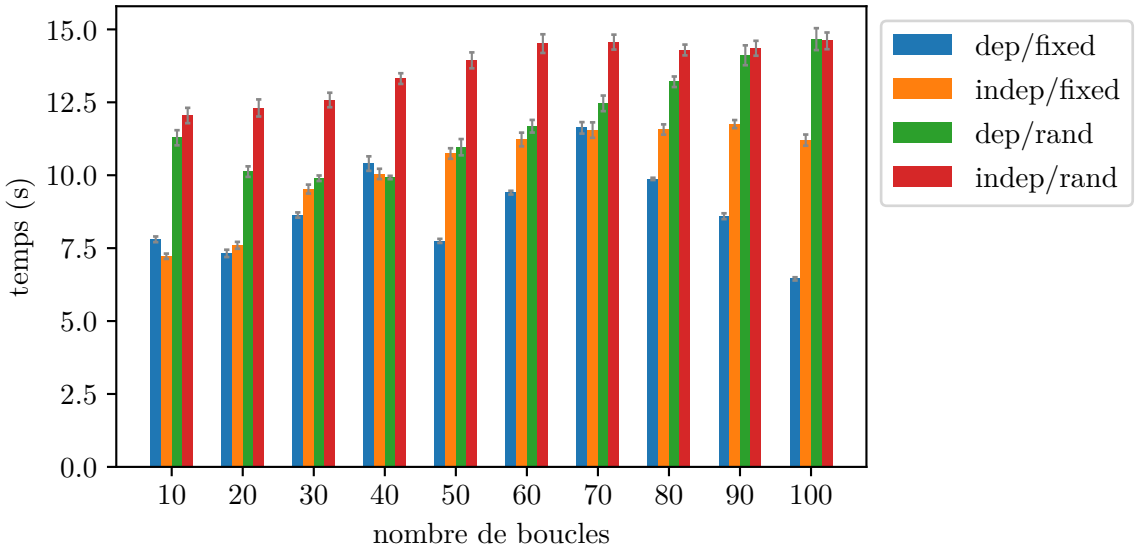


FIGURE 4.8 – Temps de compilation en fonction du nombre de boucles dans le code source

fonctions d'indice différentes sont utilisées. En pratique, il est plus fréquent d'avoir de nombreuses fonctions d'indice identiques, et éventuellement quelques unes distinctes. Les temps réalistes sont donc compris entre les valeurs mesurées pour des fonctions d'indice aléatoires et celles pour des fonctions d'indice identiques.

L'influence de l'interdépendance des instructions semble moins prononcée mais s'explique notamment par le fonctionnement de l'algorithme servant à grouper les instructions dépendantes. Enfin, nous n'expliquons pas la décroissance du temps de compilation pour le cas des instructions interdépendantes utilisant une unique fonction d'indice.

La figure 4.9 montre les temps de compilation lorsque le nombre d'instructions au sein d'une même boucle varie. L'axe des ordonnées est affiché en échelle logarithmique. Une seule boucle exécute 10 à 100 instructions (par pas de 10) utilisant jusqu'à 100 variables (lesquelles sont toujours définies).

Contrairement au cas précédent, on observe un temps de compilation exponentiel par rapport au nombre d'instructions au sein d'une même boucle. Cela se justifie principalement par les algorithmes utilisés qui sont exécutés durant la compilation. Ces temps de compilation peuvent donc être améliorés en changeant ces algorithmes, soit en trouvant de nouvelles méthodes pour procéder aux tests effectués par la bibliothèque, soit en améliorant les algorithmes actuels si cela est possible. Si l'on considère une utilisation réaliste de la bibliothèque, c'est-à-dire de nombreuses boucles ayant chacune quelques instructions, le temps de compilation est donc au maximum d'un peu plus de 10 s. Même en allant jusqu'à quelques dizaines d'instructions au sein des boucles, ce temps ne dépasse pas la minute.

Par rapport aux temps d'exécution, nous avons également pris en compte différents critères. Les figures 4.10 et 4.11 correspondent à l'exécution en boucle de 4 instructions effectuant chacune une multiplication, une addition et une affectation sur des entiers conservés dans des tableaux. La taille des tableaux (et donc le nombre d'itérations de la boucle) varie de 10^2 à 10^7. À noter que ces figures sont affichées en échelle logarithmique pour les deux axes. Les instructions exécutées étant particulièrement courtes, la boucle complète est elle-même répétée 1000 fois.

La figure 4.10 présente les temps mesurés lorsque les instructions sont telles qu'il n'est pas

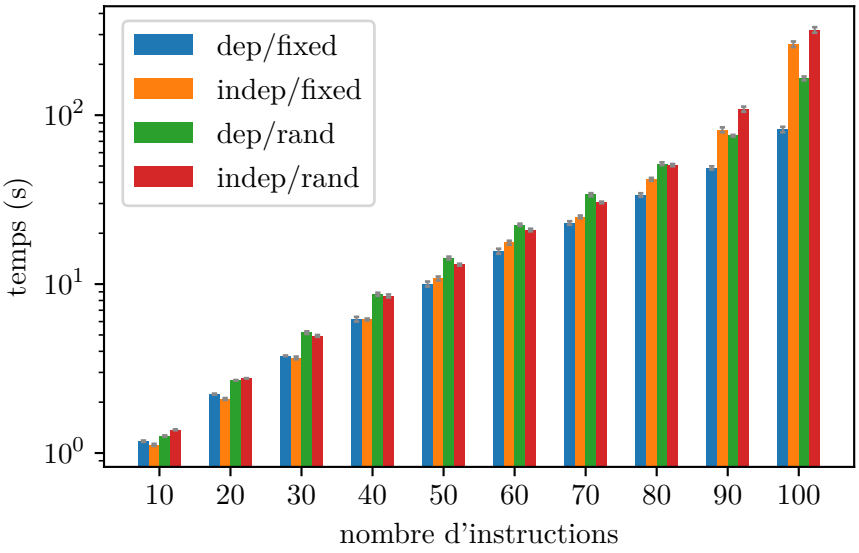


FIGURE 4.9 – Temps de compilation en fonction du nombre d'instructions par boucle dans le code source

possible d'exécuter en parallèle les itérations de la boucle. Les trois cas, dont les étiquettes sont « seq », « gen_omp » et « gen_thread », correspondent :

— « seq » : à un programme écrit classiquement, sans utiliser la bibliothèque ;
— « gen_omp » : à un programme écrit en utilisant la bibliothèque, laquelle utilise OpenMP pour la parallélisation (même s'il n'y a, dans cet exemple, aucune parallélisation possible) ;
— « gen_thread » : à un programme écrit en utilisant la bibliothèque, laquelle utilise cette fois-ci des *threads*.

À noter qu'il n'est pas possible d'utiliser OpenMP directement puisqu'il ne faut pas exécuter les itérations en parallèle. La bibliothèque, que ce soit avec OpenMP ou les *threads*, produit automatiquement, dans ce cas, un code n'effectuant aucune parallélisation.

Puisque l'objectif de la bibliothèque, lorsque les itérations de la boucle ne peuvent pas être exécutées en parallèle, est de générer un code aussi proche que possible du code d'origine, ce que l'on souhaite observer est l'absence de surcoût significatif en temps d'exécution par rapport à la version n'utilisant pas la bibliothèque. Sur cette figure, on observe effectivement un léger surcoût apporté par l'utilisation de la bibliothèque. Celui-ci pourrait être évité en agissant directement au niveau du compilateur. Cependant une bibliothèque, même active, ne peut pas générer des instructions immédiates : de nombreuses indirections sont nécessitées par les mécanismes mis en place par la bibliothèque. Cela dit, le surcoût observé à partir de 10^4 itérations est assez faible pour être accepté. Quant aux mesures pour 10^3 itérations et moins, les écarts étant minimes, ils peuvent être attribués au fait que les instructions (au niveau assembleur) sont légèrement différentes à cause des indirections causées par la bibliothèque et sont négligeables.

La figure 4.11 présente les temps mesurés lorsque les instructions sont telles qu'il est possible, contrairement au cas précédent, d'exécuter en parallèle les itérations de la boucle. Un quatrième cas s'ajoute donc aux trois précédents : un programme écrit en utilisant directement OpenMP et donc sans la bibliothèque, avec l'étiquette « omp ». Toutes les exécutions ont été faites en ayant 16 cœurs à disposition.

Le résultat le plus important à observer ici est le faible surcoût lorsque l'on compare les deux

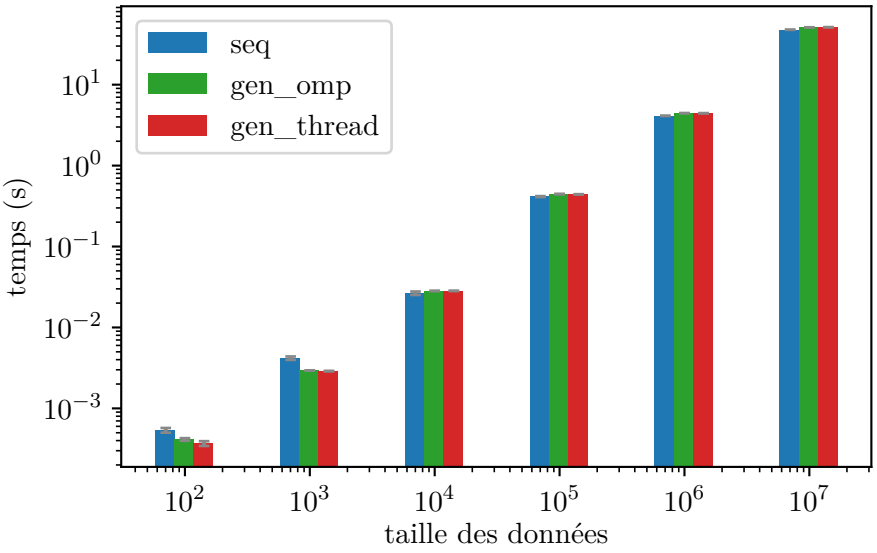


FIGURE 4.10 – Temps d'exécution séquentielle en fonction de la taille des données

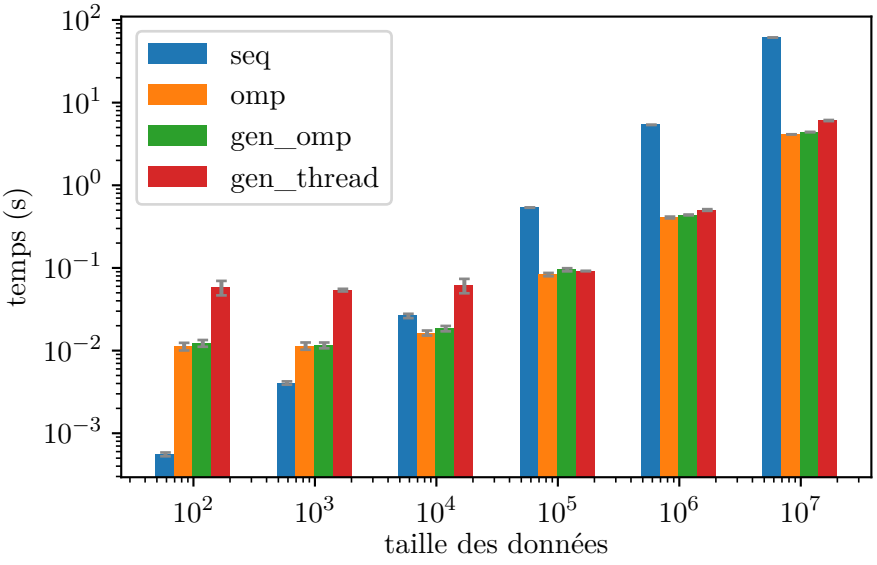


FIGURE 4.11 – Temps d'exécution parallèle en fonction de la taille des données (16 cœurs alloués)

cas utilisant OpenMP pour la parallélisation : avec et sans la bibliothèque. Quelle que soit la taille des données, et donc le nombre d'itérations, ce surcoût en temps d'exécution est négligeable.

La version séquentielle et celle utilisant les *threads* pour la parallélisation sont présentées pour référence. Cette dernière semble moins efficace que les versions employant OpenMP, surtout pour un nombre faible d'itérations. Cela s'explique notamment par le coût de création des *threads* (dont l'importance est donc relative à la taille des données) qui sont recréés à chaque nouvelle boucle, donc dans notre cas 1000 fois. Au coût d'une section critique, la mise en place d'un *thread pool* pourrait réduire ce surcoût : les *threads* créés serviraient donc pour toutes les boucles parallélisées. De plus, chaque boucle ne donnant au maximum qu'une tâche par *thread* disponible (une portion de la boucle), la section critique ne devrait pas causer de ralentissement significatif.

Ces premiers résultats sont issus de l'exécution de tâches très courtes (seulement quelques instructions assembleur par itération). Les figures 4.13 et 4.14 correspondent à un traitement plus complexe, un parcours d'image pour la modifier *in situ*. Celui-ci nous permet de montrer les performances de la bibliothèque lorsque les instructions sont plus longues, mais aussi d'en rappeler l'intérêt.

Le traitement correspond à la modification de certaines cellules : à l'exception de la première et de la dernière ligne, une cellule sur deux est modifiée de sorte que celles modifiées ne soient pas contiguës. Pour chacune d'entre elles, on utilise la valeur des cellules dans le voisinage de von Neumann pour effectuer un calcul dont le résultat est la nouvelle valeur de la cellule à modifier. La figure 4.12 montre, en grisé, quelles cellules seront modifiées dans une image de largeur 11 et de hauteur 6 et au moyen de flèches quelles cellules sont utilisées en lecture pour chacune d'entre elles. Afin de simplifier l'expression du parcours, les cellules en bordure à gauche et à droite de l'image utilisent des cellules voisines issues de lignes adjacentes.

Le code correspondant est celui de l'extrait de code 4.21. Les constantes (`constexpr`) `W` et `H` correspondent respectivement à la largeur et à la hauteur de la matrice, laquelle est conservée dans un tableau mono dimensionnel. Ainsi, par exemple la cellule au-dessus de la cellule courante i est en $i - W$.

On devine que ce traitement peut être effectué en parallèle en regardant la figure 4.12. Pour le savoir, en revanche, il faut analyser les accès au tableau. Les 5 fonctions d'indice, considérant le pas utilisé (2) et la valeur de départ $(W + 1)$, sont $i \mapsto 2\,i + W + 1$, utilisée pour un accès en écriture, et $i \mapsto 2\,i + 1$, $i \mapsto 2\,i + W$, $i \mapsto 2\,i + 2\,W + 1$ et $i \mapsto 2\,i + W + 2$, utilisées pour des accès en lecture.

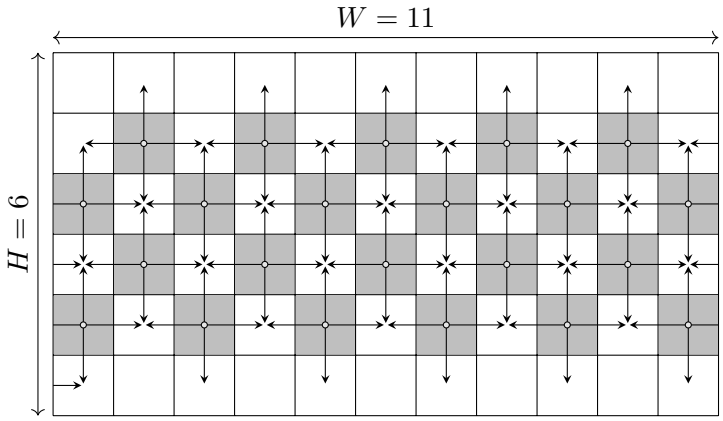


FIGURE 4.12 – Traitement d'image matricielle avec voisinage de von Neumann

```
pfor::Index i;
pfor::parallelFor(pfor::RangeCT<W+1, 2>{(H-1)*W},
  img[i] = calc(
    img[i-pfor::ctv<W>],  // north cell
    img[i-pfor::ctv<1>],  // west cell
    img[i+pfor::ctv<W>],  // south cell
    img[i+pfor::ctv<1>]   // east cell
  )
);
```

(⩾ C++14)

Extrait de code 4.21 – Traitement matriciel avec `parallelFor`

On doit alors vérifier l'inexistence de solutions aux équations Diophantiennes suivantes :

$$2\,i + W + 1 = 2\,j + 1 \tag{4.22}$$
$$2\,i + W + 1 = 2\,j + W \tag{4.23}$$
$$2\,i + W + 1 = 2\,j + 2\,W + 1 \tag{4.24}$$
$$2\,i + W + 1 = 2\,j + W + 2. \tag{4.25}$$

Lesquelles correspondent, une fois simplifiées, aux équations suivantes :

$$2\,i + W = 2\,j \tag{4.22′}$$
$$2\,i + 1 = 2\,j \tag{4.23′}$$
$$2\,i = 2\,j + W \tag{4.24′}$$
$$2\,i = 2\,j + 1. \tag{4.25′}$$

Puisque i et j sont entiers, les équations (4.23′) et (4.25′) n'ont trivialement pas de solution de par la différence de parité de part et d'autre de l'égalité. Quant aux équations (4.22′) et (4.24′), la même observation peut être faite à condition que W soit lui-même impair. En revanche, si W est pair, il apparaît que les itérations de la boucle ne peuvent pas être exécutées en parallèle. Cela peut être observé de manière plus empirique en reproduisant la figure 4.12 avec une largeur W paire. En utilisant la bibliothèque, cette analyse est effectuée automatiquement : si W peut varier, la décision de paralléliser ou non sera donc adaptée sans intervention.

Nous avons utilisé $W = 10^5 + 1$, $H = 10^5$ et une fonction `calc` définie de telle manière que le calcul effectué ne soit pas aussi court que dans les premiers résultats présentés. En pratique, la durée d'une exécution de ce calcul prend un peu moins de 10 ns. Avec ces paramètres, les figures 4.13 et 4.14 montrent respectivement le temps d'exécution en fonction du nombre de cœurs alloués et l'accélération obtenue correspondante.

Quelle que soit la manière de paralléliser, avec ou sans utiliser la bibliothèque, on observe sur la figure 4.13 quasiment les mêmes temps d'exécution. Le surcoût engendré par l'utilisation des *threads* s'estompe lorsque le nombre de cœurs croît. Il ne s'agit pas cette fois-ci d'une atténuation relative à la taille des données, puisque celle-ci est fixe, mais cela peut en revanche être lié à un coût statique (ou évoluant suffisamment lentement en fonction du nombre de *threads* créés) de mise en place des *threads*.

Dès lors que les itérations ne sont pas particulièrement courtes, l'accélération obtenue en utilisant la bibliothèque est linéaire et pratiquement identique à celle obtenue en utilisant

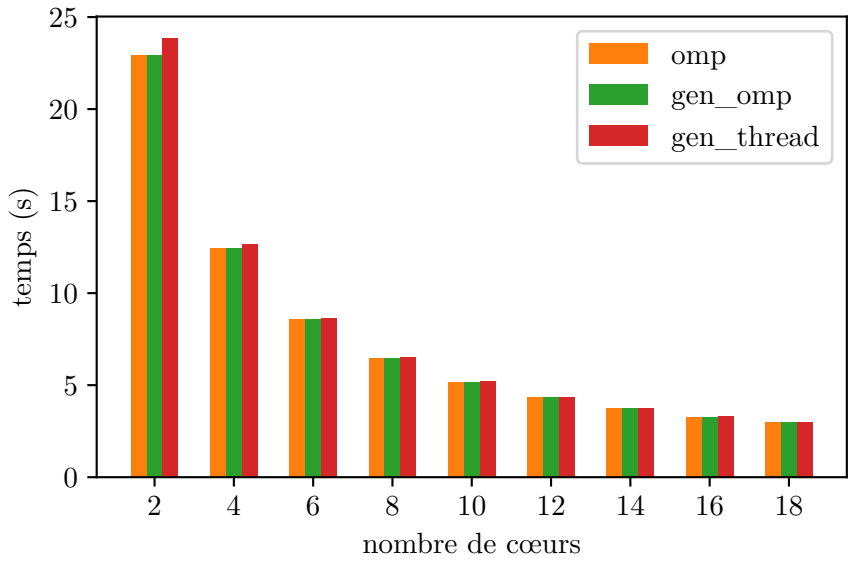


FIGURE 4.13 – Temps d'exécution parallèle en fonction du nombre de cœurs alloués

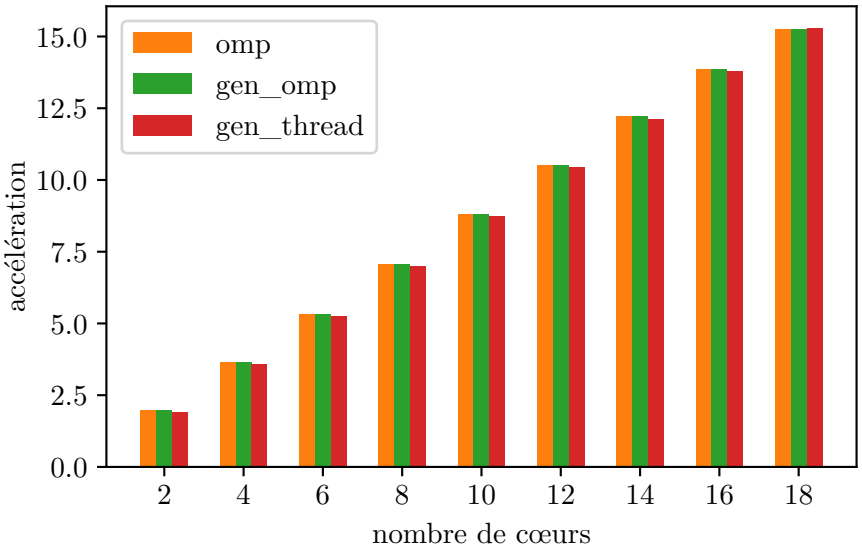


FIGURE 4.14 – Accélération en fonction du nombre de cœurs alloués

directement OpenMP comme le montre la figure 4.14. Lorsque les itérations sont trop courtes, même en utilisant directement OpenMP, comme l'ont montré les résultats ci-avant, l'accélération est limitée et ce n'est donc pas intrinsèque à notre bibliothèque. Pour éventuellement obtenir dans ce cas une meilleure accélération, il faut procéder à une parallélisation sur mesure, requérant potentiellement des modifications profondes dans le code source initial.

En conclusion, nous avons montré que le surcoût engendré par la bibliothèque durant la compilation, inévitable puisque celle-ci est active durant cette phase, n'empêche pas son utilisation puisqu'il reste faible pour des utilisations classiques. Nous avons également montré qu'elle permet d'obtenir de bonnes performances parallèles, c'est-à-dire une bonne accélération. Plus généralement, les temps d'exécution en utilisant cette bibliothèque sont aussi bons que sans l'utiliser, que ce soit lorsque la boucle doit être exécutée en parallèle ou non. Ainsi, il est effectivement possible de bénéficier de l'abstraction apportée sans devoir subir une contrepartie significative.

4.7 Conclusion

Dans ce chapitre, nous avons présenté notre proposition de bibliothèque C++ utilisant la métaprogrammation template pour automatiser la parallélisation d'une boucle. Celle-ci se distingue de solutions de parallélisation automatique par sa compatibilité avec tout compilateur C++. C'est un atout que n'ont pas les extensions, ni les compilateurs dédiés. Elle se différencie également par son utilisation de la métaprogrammation template pour effectuer, bien qu'étant une bibliothèque, les traitements nécessaires autant que possible durant la compilation afin de minimiser le surcoût induit par l'abstraction proposée, faisant d'elle une bibliothèque dite « active ».

Nous avons également détaillé les conditions nécessaires pour garantir une exécution parallèle conforme à l'exécution séquentielle : le comportement du programme ne doit pas être modifié en procédant à sa parallélisation. Pour pouvoir tester ces conditions, nous avons eu besoin d'acquérir des informations sur le programme devant être parallélisé. Nous avons donc expliqué comment les patrons d'expression sont utilisés pour disposer d'une représentation, durant la compilation, à la fois des instructions composant ce programme et aussi des fonctions utilisées pour calculer les indices permettant l'accès aux données. Ces fonctions peuvent être marquées de certaines propriétés, lesquelles sont ensuite propagées et peuvent également être automatiquement déduites.

Grâce à ces représentations, nous avons pu appliquer un test de parallélisabilité dont la sensibilité est encore améliorable pour le cas des fonctions non affines. La spécificité du test est en revanche parfaite, ce qui permet de garantir que seuls des codes pouvant effectivement être exécutés en parallèle seront positifs. Lorsque les fonctions d'indice sont affines, nous proposons un test parfaitement sensible. À défaut d'être affines, si les fonctions sont injectives, un test moins sensible est utilisé. Enfin, si les fonctions n'ont aucune propriété utilisable, par précaution, elles sont considérées comme non parallélisables. Le test peut être étendu pour traiter davantage de cas, mais certains peuvent nécessiter davantage de connaissance dès la compilation. En revanche, il semble difficile de maximiser la sensibilité sans sacrifier partiellement la spécificité.

Enfin, nous avons expliqué comment notre bibliothèque génère un programme après avoir déterminé quelles instructions peuvent être exécutées en parallèle et lesquelles ne peuvent pas l'être (soit directement soit indirectement par dépendance avec une autre instruction qui ne peut elle-même pas l'être). Nous avons montré comment notre méthode de génération permet

l'implémentation d'optimisations différentes telles que le déroulage de boucle.

Dans l'ensemble, nous avons donc introduit un cadre utilisable par des développeurs pour automatiser la parallélisation de boucles sans se préoccuper de la faisabilité de cette action. En effet, nous avons montré que le coût de l'abstraction est négligeable que ce soit pour générer des programmes séquentiels ou parallèles, mais aussi indépendamment de la granularité des instructions à paralléliser. Par ailleurs, ce cadre permet à des développeurs dont la parallélisation est l'expertise d'intégrer de nouvelles analyses, par exemple pour compléter les tests déjà implémentés, en tirant également profit de cette absence de surcoût en temps d'exécution.

Si l'on souhaite laisser davantage de contrôle au développeur quant à la parallélisation de son programme par rapport à une solution de parallélisation automatique, par exemple telle que présentée dans ce chapitre, la parallélisation assistée est une possibilité. Ce sujet est traité dans le chapitre suivant.

Chapitre 5

Parallélisation et répétabilité par les squelettes algorithmiques

5.1 Introduction

La parallélisation automatique permet au développeur de ne pas se préoccuper du tout de la problématique de parallélisation au sein de son programme. En contrepartie, il perd une partie du contrôle dont il pourrait disposer sur sa parallélisation. Ce chapitre traite d'une solution de parallélisation assistée utilisant le concept de squelettes algorithmiques [Cole 1989].

L'objectif principal est de proposer une bibliothèque active [1] permettant à un développeur de décrire ses algorithmes en spécifiant lui-même quelles parties peuvent être exécutées en parallèles, et de quelle manière elles doivent l'être en assemblant un ensemble de motifs d'exécution fournis.

Ce chapitre présente d'abord la recherche existante autour des squelettes algorithmiques puis introduit un ensemble d'outils classiques de RO (Recherche Opérationnelle) ainsi qu'un problème de RO. Ceux-ci sont utilisés comme application aux travaux présentés dans ce chapitre car ils ont des propriétés intéressantes pour mettre en évidence des problématiques liées à la parallélisation et donc les solutions proposées, notamment pour le problème de la répétabilité. Ce chapitre se poursuit avec la présentation de notre conception des squelettes algorithmiques, en détaillant en particulier leur structure et la possibilité de décrire les transmissions de données entre les différentes parties de l'algorithmique représenté. Ensuite, le chapitre explique comment se fait la répartition des tâches de l'algorithme afin de le rendre parallèle et expose différentes stratégies pour ce faire. Après cela, nous abordons le problème de la répétabilité, notamment pour le cas de l'utilisation de nombres pseudo-aléatoires, et des méthodes pour optimiser la procédure initiale en tenant compte de la stratégie de répartition des tâches adoptée et du degré de parallélisation. Ce chapitre introduit ensuite un EDSL (*Embedded Domain Specific Language*) servant d'interface pour simplifier l'utilisation de la bibliothèque. Enfin, il se termine sur une étude des performances obtenues en utilisant la bibliothèque en la confrontant à des codes écrits sans l'utiliser.

5.2 Travaux connexes

Les squelettes algorithmiques, introduits par [Cole 1989], proposent une solution de parallélisation assistée. L'objectif des squelettes est de masquer l'implémentation parallèle d'un algorithme en fournissant une interface permettant au développeur de choisir parmi des patrons de conception celui qui correspond à son besoin. Un avantage des squelettes par rapport à des interfaces plus simples, comme un ensemble de fonctions correspondant directement aux différents patrons, réside dans leur capacité à être composés [Benoit et Cole 2005].

[Cole 1989] les définit comme des fonctions d'ordre supérieur, c'est-à-dire des fonctions capables d'utiliser des fonctions comme paramètres ou retournant d'autres fonctions. L'exemple de la fonction d'ordre supérieur « map » y est donné ainsi :

$$map : (a \to b) \to ([a] \to [b]).$$

Il s'agit d'une fonction acceptant une autre fonction en paramètre et retournant une nouvelle fonction. Cette dernière agit sur un vecteur d'éléments du type de l'unique paramètre de la fonction unaire en entrée (a) et retournant un vecteur d'éléments du même type que ce qui est retourné par la fonction en entrée (b). Pour produire le vecteur de type $[b]$, la fonction en entrée est appliquée sur chaque élément du vecteur de type $[a]$. Des fonctions d'ordre supérieur ont été

1. https://phd.pereda.fr/dev/alsk

présentées dans le chapitre 3, section 3.3.3.5.

De nombreux travaux ont proposé des modèles et implémentations de squelettes algorithmiques dans différents langages de programmation depuis l'introduction de ce concept. Ceux-ci proposent généralement des patrons classiques [Campbell 1996] : parmi ceux orientés vers la parallélisation de données, on trouve *map*, *zip*, *reduce* ; et parmi ceux travaillant sur la parallélisation de tâches, il existe *farm*, *pipeline*, *divide and conquer* [Kuchen 2002].

Cette section illustre certains patrons. Si l'on considère, dans un premier temps, le patron *fork-join* qui consiste simplement en l'exécution de deux tâches T_0 et T_1 en parallèle, on peut utiliser la représentation faite dans la figure 5.1. Dans celle-ci, les tâches exécutées sont représentées par des blocs et nommées T_i où i est un entier en l'absence duquel toutes les tâches T effectuent la même chose. Les zones englobant plusieurs tâches indiquent une exécution parallèle de celles-ci.

Le patron *map* a déjà été abordé dans l'introduction de ce chapitre et correspond à l'exécution de n instances d'une tâche T sur n données, comme illustré par la figure 5.2. En remplaçant l'ensemble défini de données par un flux, un *map* devient un *farm*.

À nouveau comparable à *map*, la fonction d'ordre supérieure *zip* peut être définie ainsi :

$$zip : ((a, b) \to c) \to ([a], [b] \to [c]).$$

Cela correspond donc à l'application d'une fonction d'arité 2 sur chaque élément de deux vecteurs de données de même cardinalité ($|[a]| = |[b]|$) pour produire un unique vecteur. Il est possible de généraliser le principe à des arités quelconques.

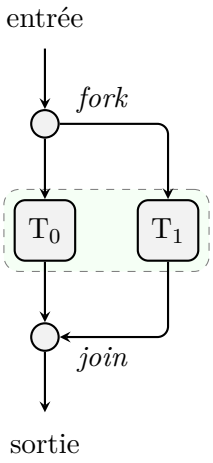


FIGURE 5.1 – Patron d'exécution parallèle *fork-join*

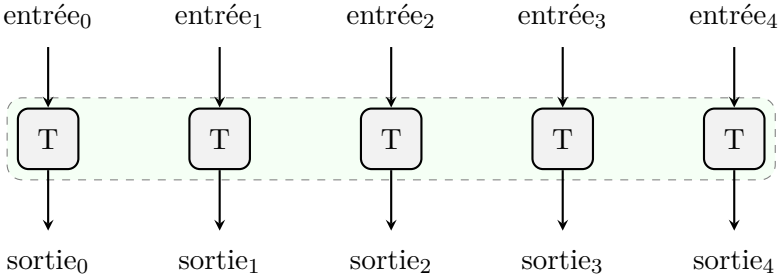


FIGURE 5.2 – Patron d'exécution parallèle *map*

L'application d'une fonction de projection (`min`, `sum`, ...) sur un ensemble de données pour le réduire à une seule est appelée réduction, et le patron associé *reduce* ou plus rarement *fold*. Il est très souvent appliqué après un *map*, ce qui correspond alors au patron *map-reduce*.

Le *pipeline* permet l'exécution parallèle de plusieurs tâches T_0, T_1, ... T_n par lesquelles les données en entrée passent successivement (figure 5.3). Le principe de « diviser pour régner », en anglais *divide and conquer*, est de répartir le travail initial entre les travailleurs par séparation successives de l'ensemble de données en entrée. Après cette étape, les résultats sont fusionnés successivement jusqu'à obtenir un résultat final. Il est possible de diviser les données jusqu'à atteindre un niveau atomique (sans fixer le nombre de travailleurs) ou de définir la profondeur maximale (limitant le nombre de travailleurs).

Les structures de contrôle classiques (branches conditionnelles, boucles, ...) peuvent également être proposées comme squelette. Il existe d'autres patrons, mais ceux qui ont été présentés ci-dessus et les combinaisons concevables entre ceux-ci sont plus que suffisants pour le propos de cette thèse.

La bibliothèque C++ TBB (*Threading Building Blocks*) propose des fonctions correspondant à divers patrons parallèles tels que `parallel_for`, `parallel_reduce`, `parallel_pipeline`, ... Ces fonctions sont conçues de manière similaire à la bibliothèque standard du langage (génériques par template, utilisation d'itérateurs).

L'exécution est parallélisée et équilibrée dynamiquement, et TBB détecte l'utilisation imbriquée de plusieurs de ses fonctions. Cette fonctionnalité permet de distinguer cette bibliothèque d'un simple ensemble de fonctions et la rapproche des squelettes algorithmiques.

De manière similaire, le langage de programmation Cilk++ implémente des mots-clés pour faciliter l'exécution parallèle ainsi qu'une structure de contrôle utilisant la syntaxe de la boucle `for` du C++, `cilk_for`. Ainsi, bien que cela ne soit pas un outil dont la conception repose spécifiquement sur les squelettes algorithmiques (à l'instar de TBB), les éléments proposés s'en rapprochent.

En revanche, [Rieger et al. 2019] propose un langage, Musket, fondamentalement basé sur le concept de squelettes algorithmiques. Celui-ci est un DSL (*Domain Specific Language*), dont la syntaxe est volontairement proche de celle du C++, l'objectif étant de permettre la génération de code C++ parallèle. Des types de données spécifiques, permettant la parallélisation, sont fournis (des types primitifs jusqu'aux matrices). Les motifs de parallélisation proposés sont *map*, *reduce*, *zip* et *shift partition*.

Les auteurs de Musket ont détaillé les raisons pour lesquelles ils ont choisi de développer un langage plutôt qu'une bibliothèque [Wrede et al. 2020]. Celles-ci entraînent un surcoût en temps d'exécution par rapport à une implémentation manuelle ou une génération de code comme peut le faire le compilateur d'un langage. D'autre part, les bibliothèques étant contraintes à respecter la syntaxe du langage hôte, il est plus facile de proposer une meilleure lisibilité du code au sein d'un DSL. Enfin, cette dernière contrainte apporte un autre argument au choix d'un langage dédié : cela permet davantage de flexibilité dans les transformations appliquées sur le code pour produire un programme parallèle.

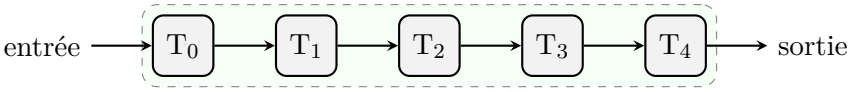


FIGURE 5.3 – Patron d'exécution parallèle *pipeline*

De nombreuses bibliothèques de squelettes algorithmiques ont été proposées [Aldinucci et al. 2009 ; Ciechanowicz et al. 2009 ; Leyton et Piquer 2010 ; Legaux et al. 2013 ; Ernstsson et al. 2018 ; Philippe et Loulergue 2019]. [Striegnitz 2000] a montré que le langage C++ permettait probablement l'implémentation de bibliothèques de squelettes algorithmiques. Plus tard, [J. Falcou et al. 2006] l'a prouvé et a également montré que l'utilisation des templates permettait d'obtenir un moindre surcoût en temps d'exécution. La différence notable entre une bibliothèque « classique » et ce que permettent les templates en C++ est la possibilité de réellement agir durant la compilation, les rendant « actives » [Veldhuizen et Gannon 1998]. Enfin, il est possible de proposer un DSL au sein de certains langages existants, comme c'est le cas du C++ : on parle alors d'un EDSL [Saidani et al. 2009]. Cela permet de mitiger la scission entre langage et bibliothèque en termes de performances et de flexibilité, au moins dans le cadre du C++.

Les travaux effectués durant cette thèse sur les squelettes algorithmiques ont ainsi été orientés vers le C++, et plus spécifiquement vers l'implémentation de bibliothèques faisant usage de la métaprogrammation template de ce langage. Les inconvénients des bibliothèques par rapport à la création d'un langage ou même par rapport à l'implémentation d'une extension pour un compilateur sont amoindris par l'utilisation de la métaprogrammation. En revanche, une bibliothèque apporte des avantages que nous avons jugé importants. Premièrement, elles s'inscrivent dans un cadre qui peut déjà être connu par le développeur, et dans le cas du langage C++, qui est très vastement utilisé, c'est particulièrement le cas. L'implémentation d'un nouveau langage nécessite également la mise en place des nombreuses optimisations déjà présentes et à venir sur les compilateurs de langages existants, alors qu'une bibliothèque profitera de celles-ci automatiquement. Certes, un nouveau langage peut, plutôt que d'être compilé directement vers de l'assembleur, générer du code dans un autre langage bénéficiant de ces optimisations [Rieger et al. 2019], cependant, cela requiert une maintenance parallèle des deux langages. Il existe également le cas des extensions pour un compilateur, mais celles-ci doivent alors être implémentées pour chaque compilateur, ou bien imposer aux utilisateurs de n'utiliser qu'un sous-ensemble des compilateurs disponibles. Une bibliothèque dont l'implémentation ne repose que sur le standard du langage ne crée aucune de ces contraintes.

5.3 Application

L'objectif de cette section est d'introduire les problèmes sur lesquels nous avons appliqué notre solution. Nous avons choisi de travailler sur la résolution de problèmes de RO. Ceux-ci offrent plusieurs niveaux de parallélisation qui peuvent être entrelacés avec des niveaux non parallélisables, ce qui nous a intéressé pour valider notre représentation globale d'un algorithme. L'utilisation intensive de nombres pseudo-aléatoires dans ce domaine nous a également permis d'éprouver notre solution en ce qui concerne l'obtention de résultats répétables.

Le voyageur de commerce (TSP (*Travelling Salesman Problem*)) [Robinson 1949] est un problème très classique de RO. Une instance de ce problème consiste en un graphe $G = (V, A, c)$ où V est un ensemble de sommets, A un ensemble d'arêtes et c une fonction de coût qui associe une distance minimale à un arc de A [Dantzig et al. 1954]. Une solution peut être exprimée comme un vecteur s de sommets contenant une fois chaque sommet de V, sa valeur c_s étant alors donnée par l'équation (5.1), où (a, b) correspond à l'arête entre le sommet a et le sommet b.

$$c_s = \sum_{i=2}^{|V|} c((s[i-1], s[i])) \tag{5.1}$$

L'objectif est alors de trouver un cycle qui minimise c_s. La figure 5.4 est un exemple d'instance de TSP avec 10 sommets pour lequel nous considérerons que les coûts sont les distances entre les deux sommets concernés (précisément la norme euclidienne). Deux solutions sont montrées dans la figure 5.5, celle de droite étant optimale.

Les algorithmes peuvent être évalués selon un critère de complexité, temporelle ou spatiale. Par extension, il est possible d'associer une complexité à un problème comme étant la meilleure de celles des algorithmes possibles permettant de le résoudre. Afin de catégoriser les problèmes en fonction de leur complexité, des classes de complexité ont été définies. Il existe par exemple la classe P dont les problèmes peuvent être résolus en temps polynomial, et sont donc considérés comme « faciles ». En revanche, pour les problèmes de classe NP, ce n'est que la vérification d'une solution qui peut être faite en temps polynomial. Un problème NP-complet est un problème de classe NP qui est au moins aussi difficile que tous les autres problèmes de la classe NP et le TSP est NP-complet : s'il est possible de vérifier une solution en temps polynomial, en trouver une est difficile. Cette catégorie de problèmes peut être traitée en RO par des heuristiques et des métaheuristiques.

Le principe d'une heuristique est de trouver rapidement une solution au détriment de sa qualité. Un exemple d'heuristique est l'algorithme glouton (algorithme 5.1). Appliqué au TSP, celui-ci accepte en entrée une représentation du problème P contenant notamment l'ensemble des sommets. Une solution est alors construite progressivement en lui ajoutant le sommet jugé optimal à chaque étape. La figure 5.6 présente la solution obtenue par l'application de cet algorithme sur l'instance de la la figure 5.4 en démarrant du point grisé. Il existe des variantes non déterministes, par exemple en considérant les $n > 1$ meilleurs ajouts possibles à chaque étape et en ajoutant l'un d'eux au hasard.

Algorithme 5.1 Algorithme glouton

```
fonction GLOUTON(P)
    S ← ∅
    Construire la liste L des éléments insérables dans S à partir de P
    tant que L ≠ ∅ faire
        Évaluer le coût d'insertion de chaque élément de L
        Retirer de L l'élément de coût minimal
        Insérer cet élément dans S
    retourner S
```

Une métaheuristique est une heuristique générique pouvant s'appliquer à différents problèmes sans changements importants de l'algorithme. Les exemples sont nombreux [Toussaint 2010] : recuit simulé, recherche tabou, recherche à voisinage variable, ...

Ceux qui vont nous intéresser particulièrement, étant utilisés comme exemples dans la suite de ce chapitre, sont :

— le GRASP (*Greedy Randomized Adaptive Search Procedure*) ;
— l'ILS (*Iterative Local Search*) ;
— l'ELS (*Evolutionary Local Search*).

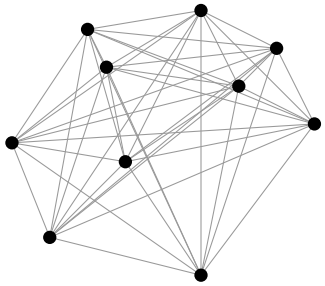


FIGURE 5.4 – Instance de TSP

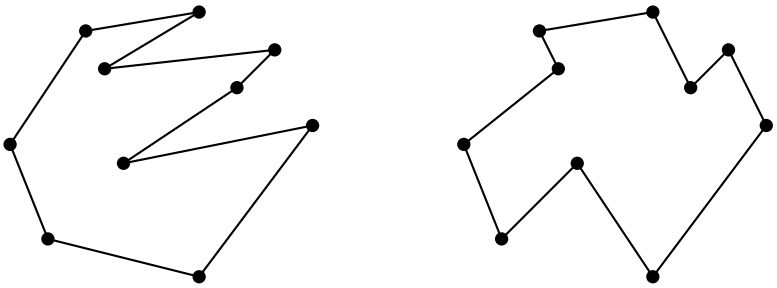


FIGURE 5.5 – Solutions de l'instance de TSP de la figure 5.4

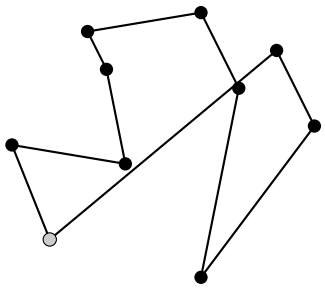


FIGURE 5.6 – Solution d'une instance de TSP par un algorithme glouton

Le GRASP [Feo et Resende 1989] (algorithme 5.2) consiste en la création de multiples solutions à différents endroits de l'espace des solutions (l'intérêt est amoindri si les solutions sont trop proches) suivie de l'amélioration par modifications successives (des déplacements « locaux » dans l'espace des solutions). La création est accomplie par une heuristique constructive, par exemple un algorithme glouton non déterministe, et l'amélioration par une recherche locale. Une recherche locale consiste en l'exploration partielle ou complète du voisinage d'une solution dans l'espace des solutions. Le voisinage d'une solution est l'ensemble des solutions que l'on peut obtenir en appliquant une transformation sur celle-ci, aussi appelée mutation.

Algorithme 5.2 GRASP

```
fonction GRASP(N, P)
    pour i = 1..N faire
        S_i ← HEURISTIQUECONSTRUCTIVE(P)
        S_i ← RECHERCHELOCALE(P, S_i)
    S* ← SÉLECTION({S_1, S_2, ..., S_N})
    retourner S*
```

Les transformations qu'il est possible de faire dépendent du problème : dans le cadre du TSP, il existe par exemple l'échange qui consiste en l'échange aléatoire de deux sommets dans la solution (voir la figure 5.7). La descente est un exemple de recherche locale qui va remplacer la solution par son meilleur voisin jusqu'à ce qu'il n'existe pas de voisin au moins aussi intéressant.

Les différentes itérations de création (par heuristique constructive (HC)) et amélioration d'une solution (par recherche locale (RL)) sont indépendantes et peuvent donc être exécutées en parallèle. La sélection (S) de la solution doit en revanche être accomplie de manière séquentielle.

La figure 5.8 représente un GRASP en tenant compte de cela. Les tâches à exécuter sont représentées par des cercles tandis que les triangles indiquent une section (ouverte par le triangle qui sépare le flot d'exécution, et fermée par le triangle qui les réunit) dont les flots d'exécutions peuvent être exécutés en parallèle.

L'ILS [Lourenço et al. 2003] (algorithme 5.3) améliore une solution S plutôt que d'en créer une. Dans un premier temps, la solution S donnée en paramètre est améliorée au moyen d'une recherche locale (initiale afin de la différencier de la recherche locale utilisée ensuite). L'amélioration de cette solution est ensuite effectuée en itérant N fois l'amélioration par recherche locale d'une mutation de la solution courante S. À chaque itération, la solution courante est mise à jour pour prendre la nouvelle solution créée ou non en fonction d'un critère d'acceptation. Si cette solution est, par ailleurs, meilleure que la meilleure solution S^* retenue jusqu'alors, elle la remplace.

Chaque itération de cet algorithme dépendant du résultat de l'itération précédente, il n'est

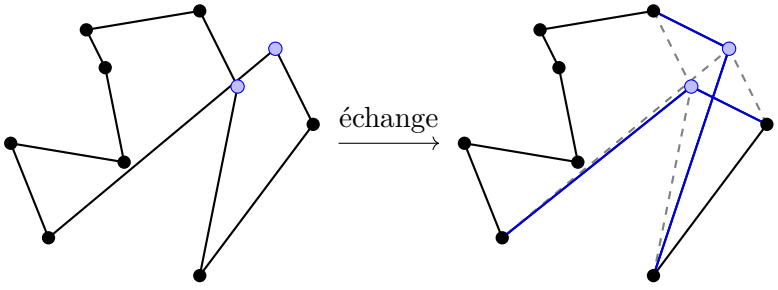


FIGURE 5.7 – Échange de deux sommets d'une solution d'une instance de TSP

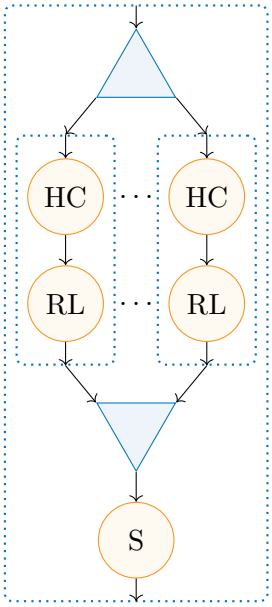


FIGURE 5.8 – Schéma d'un GRASP

Algorithme 5.3 ILS

fonction ILS(P, S, N)
  $S \leftarrow$ RECHERCHELOCALEINITIALE(P, S)
  $S^* \leftarrow S$
  pour $i = 1..N$ **faire**
    $S' \leftarrow$ MUTATION(S)
    $S' \leftarrow$ RECHERCHELOCALE(P, S')
    $S^* \leftarrow$ SÉLECTION(S^*, S')
    si CRITÈREACCEPTATION(S') **alors**
      $S \leftarrow S'$
  retourner S^*

pas possible de le paralléliser. La figure 5.9 représente cela par l'utilisation de carrés pour la boucle. Les tâches de mutation (M), de recherche locale (RL), de sélection (S) et de vérification du critère d'acceptation (A) sont ainsi exécutées séquentiellement à chaque itération, elles aussi exécutées en séquence.

Enfin, l'ELS [Wolf et Merz 2007] (algorithme 5.4) fonctionne de manière similaire à l'ILS. On retrouve la recherche locale initiale et une boucle principale permettant d'améliorer une solution S donnée en paramètre. À chaque itération, M solutions sont générées par une mutation suivie d'une recherche locale et la meilleure, si elle vérifie le critère d'acceptation, est conservée pour l'itération suivante.

Cet algorithme est plus intéressant que le précédent quant à la parallélisation. Bien qu'il possède une boucle externe non parallélisable, la boucle interne peut l'être. La figure 5.10 présente ainsi une structure externe très semblable à la figure 5.9 dont la séquence « mutation, recherche locale » est remplacée par une tâche plus complexe introduisant une boucle parallélisable.

Ces métaheuristiques peuvent également être combinées : [Prins 2009] a proposé GRASP×ILS et GRASP×ELS, des métaheuristiques GRASP dont la recherche locale est, respectivement, ILS et ELS. C'est en particulier le GRASP×ELS qui va servir d'exemple dans ce chapitre pour illustrer l'utilisation des squelettes algorithmiques que nous proposons [2]. Sa structure générale possède deux niveaux de boucles parallélisables entrecoupés par un niveau de boucle devant être exécuté séquentiellement, ce qui en fait une application intéressante pour la validation de notre modèle.

Algorithme 5.4 ELS

```
fonction ELS(P, S, N, M)
    S ← RECHERCHELOCALEINITIALE(P, S)
    S* ← S
    pour i = 1..N faire
        pour j = 1..M faire
            S_j ← MUTATION(S)
            S_j ← RECHERCHELOCALE(P, S_j)
        S' ← SÉLECTION1({S_1, S_2, ..., S_M})
        S* ← SÉLECTION2(S*, S')
        si CRITÈREACCEPTATION(S') alors
            S ← S'
    retourner S*
```

2. Une bibliothèque implémentant des algorithmes de RO, notamment le GRASP×ELS, fait partie des travaux effectués durant la thèse (`https://phd.pereda.fr/dev/rosa`).

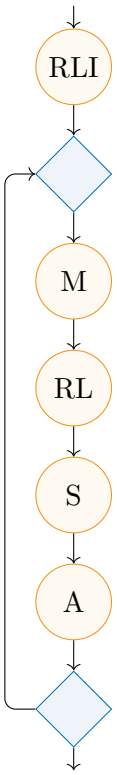


FIGURE 5.9 – Schéma d'un ILS

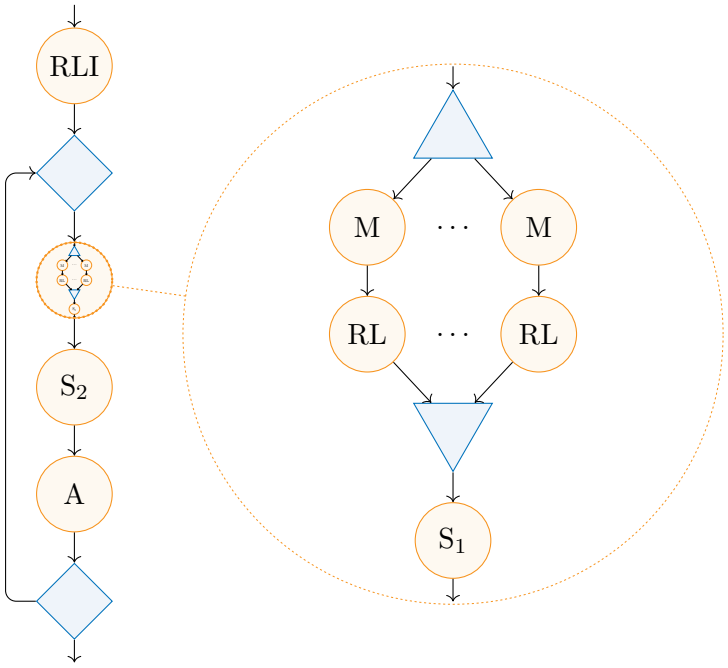


FIGURE 5.10 – Schéma d'un ELS

5.4 Conception

Un squelette algorithmique est le résultat de l'assemblage de plusieurs composants détaillés dans cette section. Dans les figures 5.8 à 5.10, les cercles représentent les tâches devant être exécutées et qui sont fournies par le développeur utilisateur. Celles-ci sont communément appelées des muscles et sont, dans notre implémentation, de simples fonctions ou, plus exactement, tout ce qui peut être utilisé comme une fonction. En C++, grâce à la surcharge de la fonction membre `operator()`, il est possible pour une instance d'une classe de se comporter comme une fonction. On peut parler de fonctionoïdes [3], terme que nous utiliserons dans ce document pour regrouper l'ensemble des éléments ressemblant à une fonction.

Dans ces les figures 5.8 à 5.10, les arêtes liant les muscles représentent les transitions entre ceux-ci. En particulier, cela représente les transferts de données. En général, les bibliothèques de squelettes algorithmique font le choix d'imposer à leurs utilisateurs comment les données sont transmises en définissant implicitement la nature du transfert. Pour un patron d'exécution *pipeline*, cela peut, par exemple, se traduire par l'utilisation de la valeur retournée par le premier muscle comme premier argument du deuxième, et ainsi de suite. Ce choix peut être fait pour simplifier l'interface, mais, en conséquence, celle-ci est moins flexible. Nous avons choisi de laisser le développeur définir lui-même les contraintes qui décrivent comment les données sont transférées et que nous avons appelées des liens. Ce mécanisme permet en outre l'intégration d'autres fonctionnalités qui seront présentées dans la section 5.6.

Enfin, la structure générale du squelette correspond à la manière d'assembler les différents patrons. Dans notre cas, les patrons, séquentiels ou parallèles, sont les unités structurelles minimales. Pour cette raison, nous les avons nommés « os ». Tout squelette est composé uniquement d'os, mais la construction d'un squelette peut utiliser, de manière récursive, un assemblage sans distinction d'os et de squelettes.

Cette section a pour but de présenter notre implémentation de la structure des squelettes, dans un premier temps, puis, dans un second temps, des liens que ces squelettes utilisent. Pour cela, nous utilisons l'exemple du GRASP×ELS, introduit dans la section précédente. Notre objectif sera de représenter cet exemple avec des squelettes algorithmiques.

5.4.1 Structure

Comme cela a été dit, les os sont les éléments de base de la construction d'un squelette. Ils possèdent des caractéristiques, rendues accessibles par une spécialisation d'un template de classe de *traits* (voir la section 3.2.2), qui permettent par exemple de savoir si celui-ci peut exécuter des tâches en parallèle. Un os fournit son implémentation par la spécialisation d'un template de classe dédié, dont l'interface principale est l'`operator()`.

Cette implémentation est propre à chaque os puisqu'elle dépend du comportement que celui-ci représente. Elle dépend également de paramètres extrinsèques comme le mode d'exécution. À ce jour, la bibliothèque ne supporte que deux modes : séquentiel et parallèle ; mais il est possible d'étendre cette capacité en créant de nouveaux *tags* (voir la section 3.3.3.4) et les implémentations d'os associées.

Ainsi, un os tel que `Serial`, dont la nature est d'exécuter en séquence plusieurs tâches, aura une implémentation unique indépendante du mode d'exécution (qui ne peut pour le moment

3. https://isocpp.org/wiki/faq/pointers-to-members#functionoids

qu'être séquentiel ou parallèle). En revanche, un os représentant une ferme de tâches devra s'adapter et donc proposer deux implémentations distinctes. Il est techniquement possible de n'en proposer qu'une seule qui fonctionnerait pour un nombre quelconque de *threads* (incluant 1), cependant le programme séquentiel généré serait moins efficace : la généricité sur le nombre de *threads* implique au moins l'existence de tests qui n'apparaîtraient pas dans un code dédié.

L'extrait de code 5.1 définit la structure d'un squelette minimal (composé d'un seul os, `Serial`) qui exécutera 4 tâches `Task1` à `Task4` en série. Le template `S` (pour structure) permet de stocker l'os et les muscles associés pour une utilisation durant la compilation. La structure d'un squelette étant faite pour être utilisée ultérieurement (et possiblement dans différents contextes), il est utile de la nommer, comme on le fait pour des variables. S'agissant d'un type ou d'un template, `using` peut être utilisé (voir l'extrait de code 5.2).

```
S<Serial, Task1, Task2, Task3, Task4>
```

(⩾ C++14)

Extrait de code 5.1 – Structure d'un squelette composé d'un seul os

D'autre part, les muscles utilisés (`Task1` à `Task4`) n'influent aucunement sur la structure et sont habituellement inconnus au stade de la conception d'un squelette. Ainsi, il est souhaitable de paramétrer cette structure sur les muscles qui y seront insérés comme dans l'extrait de code 5.2.

```
template<typename Task1, typename Task2, typename Task3, typename Task4>
using MinStruct = S<Serial, Task1, Task2, Task3, Task4>;
```

(⩾ C++14)

Extrait de code 5.2 – Structure paramétrée d'un squelette composé d'un seul os

Lorsqu'un squelette est complet, c'est-à-dire que l'on dispose de la structure avec les liens (prochaine section) et les muscles, il est possible de produire une implémentation de celui-ci. Cette implémentation fournit un `operator()` (dont les spécificités seront expliquées dans la section sur les liens), ce qui en fait donc un fonctionoïde. Pour cette raison, l'implémentation produite par un squelette peut être utilisée comme muscle d'un autre squelette. La bibliothèque tire profit de cette caractéristique pour, au sein de la définition d'une structure de squelette, traiter une autre structure comme un muscle. Grâce à cela, il est possible de définir des squelettes plus complexes, eux-mêmes composés de squelettes.

La structure d'un GRASP (figure 5.8) peut alors être définie par un patron exécutant plusieurs fois une même tâche et sélectionnant le meilleur résultat obtenu (os nommé `FarmSel`). La tâche principale (par opposition à la tâche de sélection) est une séquence de deux tâches (os `Serial`) (extrait de code 5.3). Dans les codes, écrits en anglais, les noms des tâches des schémas sont adaptés. Ainsi l'heuristique constructive HC devient le type `CH`, la recherche locale RL devient `LS` et enfin la tâche de sélection S devient `Sel`.

```
template<typename CH, typename LS, typename Sel>
using GraspStruct =
S<FarmSel,
  S<Serial, CH, LS>,
  Sel
>;
```

(⩾ C++14)

Extrait de code 5.3 – Structure du squelette d'un GRASP

Cette construction est mieux schématisée par un arbre (figure 5.11). Celui-ci peut-être obtenu à partir de la représentation du GRASP de la figure 5.8. Chaque branche de l'arbre correspond directement à un patron utilisé dans le GRASP, et chaque feuille correspond à un muscle. Un nœud A est l'enfant d'un nœud B si les éléments homologues A' et B' au sein du squelette sont liés par le fait que B' possède A' comme tâche à exécuter. Cette représentation souligne l'aspect « patron d'expressions » qu'arborent les squelettes algorithmiques. Elle est également plus compacte, en particulier parce qu'elle ne répète aucune information.

Ainsi, le GRASP×ELS peut être schématisé de manière différente par les figures 5.12 et 5.13, mais le schéma de cette dernière figure est plus compact, car les éléments parallèles ne sont pas répétés, et aussi plus proche de la représentation que la bibliothèque a de cet algorithme. La définition de la structure du GRASP×ELS peut être obtenue en définissant une structure de l'ELS, formant le cœur du GRASP×ELS, et en l'utilisant comme un muscle qui remplace la recherche locale dans la structure du GRASP (extrait de code 5.4). À noter que pour alléger les extraits de code et les figures, le critère d'acceptation (tâche A) ne sera dorénavant pas spécifié au sein de l'ELS. Il s'agit simplement d'une tâche devant être effectuée en série, et donc d'un os `Serial`.

S'il est aussi possible de définir directement la structure globale du GRASP×ELS, la méthode

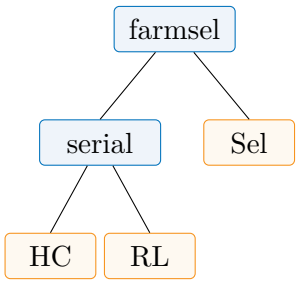


FIGURE 5.11 – Arbre représentant la structure d'un GRASP

```
template<typename InitLS, typename M, typename LS, typename Sel1, typename Sel2>
using ElsStruct =
S<Serial,
  InitLS,
  S<IterSel,
    S<FarmSel,
      S<Serial, M, LS>,
      Sel1
    >,
    Sel2
  >
>;

// GraspStruct est défini dans l'extrait de code 5.3
template<
  typename CH, typename Sel1,
  typename InitLS, typename M, typename LS, typename Sel2, typename Sel3
>
using GraspElsStruct =
GraspStruct<CH, ElsStruct<InitLS, M, LS, Sel2, Sel3>, Sel1>;
```

(≥ C++14)

Extrait de code 5.4 – Structure du squelette d'un GRASP×ELS

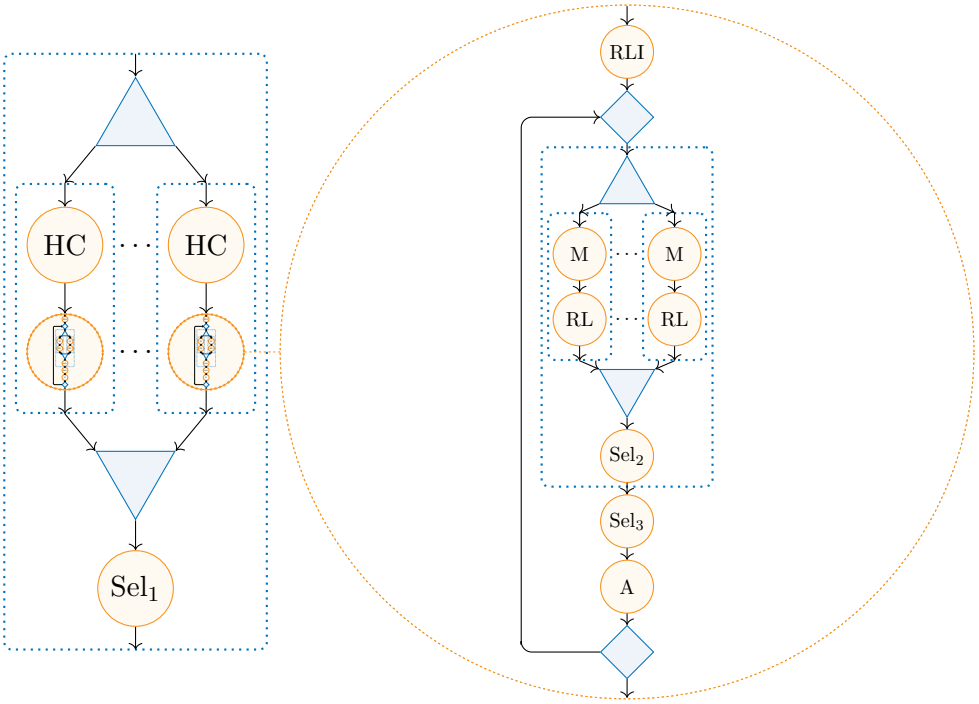


FIGURE 5.12 – Schéma d'un GRASP×ELS

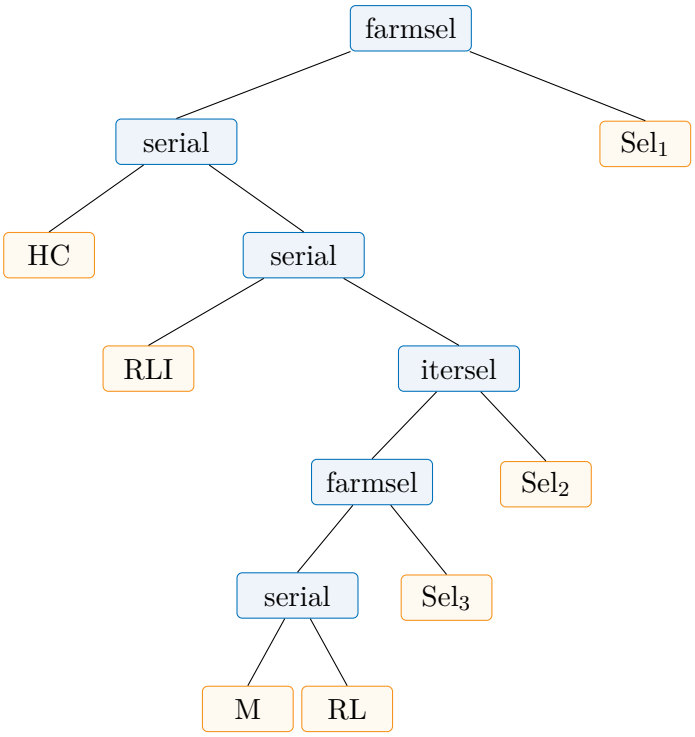


FIGURE 5.13 – Arbre représentant la structure d'un GRASP×ELS

présentée est préférable puisqu'elle offre une meilleure extensibilité. En effet, en définissant plusieurs petites structures (GRASP, ELS, ILS, ...), celles-ci peuvent être réutilisées pour construire d'autres structures (GRASP×ILS, GRASP×ELS, ...) sans la répétition de code qu'aurait induit la création sans intermédiaires de ces dernières.

5.4.2 Transmission des données

Classiquement, les squelettes algorithmiques imposent leur flux de données. Le développeur doit donc adapter ses muscles pour que leur interface (dans le cas des fonctions, leur signature : type de retour et types des paramètres) soit compatible. Cela permet de proposer un outil plus simple d'utilisation puisqu'il y a moins d'information à fournir par le développeur pour décrire un squelette. Nous avons cependant fait le choix de ne pas définir de liens (mécanisme de transmission des données) par défaut entre les différents éléments d'un squelette car cela permet une plus grande liberté pour l'utilisateur. De plus, comme le montre la section 5.6, cela permet de proposer des fonctionnalités supplémentaires. Ce mécanisme apporte beaucoup au piment des squelettes algorithmiques proposés au sein de cette thèse.

La définition complète d'un squelette (sans ses muscles) nécessite donc l'information des données en entrée et en sortie de chaque futur muscle. Chaque partie pouvant être exécutée étant un fonctionoïde, la description du type de retour et des types des paramètres est facilement faite par la syntaxe C++ décrivant le type d'une fonction : `R(P1, P2, ..., Pn)` où `R` est le type de retour et `P1` à `Pn` sont les types des n paramètres.

Suivant une syntaxe très proche de celle utilisée pour décrire la structure, l'extrait de code 5.5 définit les types de données qu'utilisent les différents fonctionoïdes impliqués dans un squelette qui exécute en séquence deux tâches. Le template `L` (pour lien) permet de stocker, pour un os donné (ici `Serial`), à la fois la signature de la fonction qu'il définit et cette même information pour chacun de ses muscles. De manière similaire à ce qui peut être accompli avec le template `S` lors de la définition d'une structure de squelette, il est possible d'utiliser une instance du template `L` en substitution d'un muscle.

```
L<Serial, int(int, int),
  int(int),
  int(int, int, int)
>
```

(⩾ C++14)

Extrait de code 5.5 – Liens d'un squelette composé d'un seul os

Cet exemple est cependant erroné car il n'apporte pas l'information attendue par le système : les liens entre les différents paramètres. Pour le GRASP, il s'agit (voir la figure 5.14) :

- de connecter le premier paramètre (P_0), le problème, aux muscles HC et RL (exécutés par l'os `Serial`) ;
- de connecter le second paramètre (P_1), un PRNG (*Pseudorandom Number Generator*), aux muscles HC et RL ;
- de connecter la sortie du muscle HC (R_0) à l'entrée du muscle RL ;
- et enfin de connecter les n sorties des n exécutions de RL au muscle de sélection.

Cette description peut être transcrite en C++ comme dans l'extrait de code 5.6. Deux templates spéciaux sont utilisés : le template `P` qui, avec un paramètre template représentant un entier i, permet d'indiquer un argument qui doit être remplacé par le $(i+1)^{\text{ème}}$ paramètre

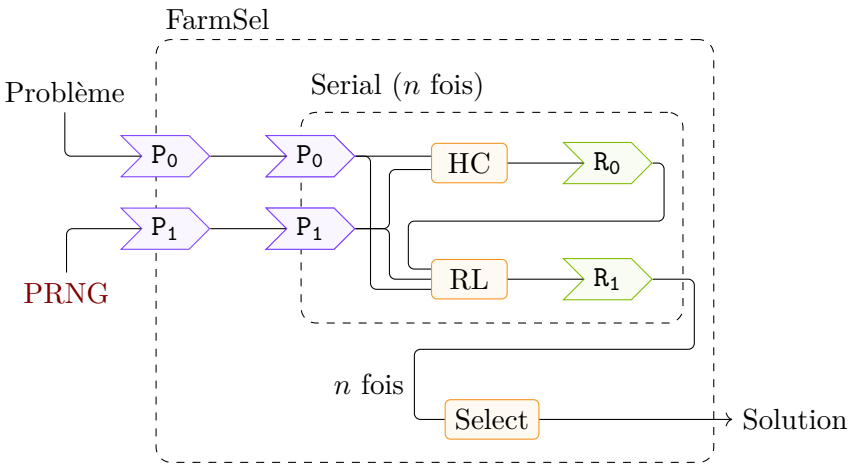


FIGURE 5.14 – Transmission de données au sein d'un GRASP

de l'appelant ; le template `R`, qui utilise de même un paramètre template représentant un entier j, correspond à la valeur retournée par le $(j+1)^{\text{ème}}$ muscle de l'os dans lequel ce template est utilisé.

```
1  template<typename Problem, typename Solution, typename PRNG>
2  using GraspLinks =
3  L<FarmSel, Solution(Problem const&, PRNG&),
4    L<Serial, R<1>(P<0>, P<1>),
5      Solution(P<0>, P<1>),
6      Solution(P<0>, R<0>, P<1>)
7    >,
8    Solution(Solution const&, Solution const&)
9  >;
```

(⩾ C++14)

Extrait de code 5.6 – Liens du squelette d'un GRASP

Ainsi, la ligne 6 indique que le muscle RL prendra en premier argument le premier argument donné à l'os `Serial`, comme deuxième argument la valeur retournée par le muscle HC (dont l'indice parmi les muscles appelées par l'os est 0, d'où `R<0>`), et comme dernier argument le deuxième de l'appelant. La valeur retournée par le muscle n'est pas contrôlée par le système de liens et doit donc être le véritable type de retour.

La ligne 4 décrit les liens d'un os plutôt que ceux d'un muscle. Le fonctionnement ne diffère que pour ce qui est retourné qui peut alors être choisi. Dans ce cas, il s'agit de la valeur retournée par le second muscle de cet os, c'est-à-dire le muscle RL.

La ligne 8 concerne le muscle Select de l'os `FarmSel`. Dans ce cas particulier, le fonctionnement interne de l'os, qui impose la comparaison de solutions, contraint les arguments qui seront donnés à ce muscle, empêchant partiellement l'utilisation des templates `P` et `R`. L'os fournit lui-même les deux solutions à comparer, et ce autant de fois que nécessaire pour déterminer la solution qui doit finalement être conservée. En revanche, si le muscle requiert des arguments supplémentaires (par exemple un PRNG), ceux-ci peuvent être transmis au moyen des templates `P` et `R`.

Enfin, la ligne 3 décrit les liens de l'os `FarmSel`. Deux particularités sont importantes à noter ici. D'abord, pour la même raison qui fait que le muscle de sélection ne peut pas entièrement contrôler ses arguments, il n'est pas possible de choisir ce qui est retourné par cet os (contrairement à l'os `Serial` qui le permet). Ensuite, parce qu'il s'agit de l'os le plus externe (cet os est la racine

de l'arbre correspondant), il ne possède pas de fonction appelante à ce stade, rendant invalides l'utilisation des templates `P` et `R` pour définir des arguments. S'il est ensuite utilisé comme muscle d'un autre squelette, dans ce nouveau contexte il sera possible d'utiliser ces templates pour établir des liens.

Comme cela avait été fait pour la structure, cette définition des liens est nommée et paramétrée. Cette fois-ci, ce sont les types des paramètres, c'est-à-dire les types des données manipulées, qui n'ont pas besoin d'être connus. Cela permet également la réutilisation de cette définition de liens au sein d'une autre définition.

5.4.3 Instanciation du squelette

En combinant la structure et les liens associés, on obtient un squelette (extrait de code 5.7). Le template `BuildSkeleton` construit un arbre exprimant le squelette complet à partir de sa structure et des liens auxquels sont transmis respectivement les muscles (premier pack de paramètres) et les types de paramètres des fonctionoïdes (second pack de paramètres). À ce stade, il est possible de produire un squelette en fixant une partie de ces informations. Dans cet exemple, toutes restent inconnues, faisant de `GraspSkel` un template ayant pour paramètres l'ensemble des types des paramètres des muscles, ainsi que ces derniers.

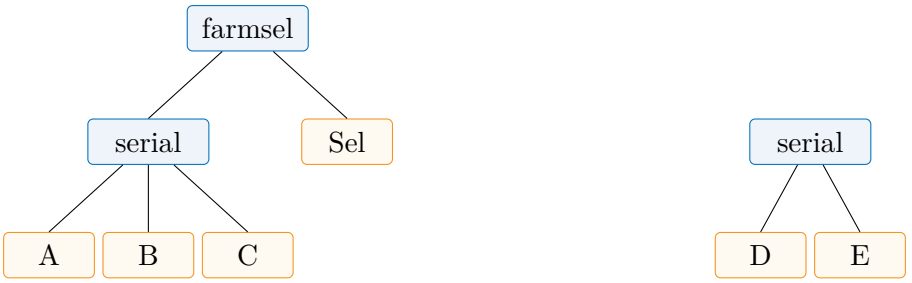
```
template<
  typename Problem, typename Solution, typename PRNG,
  typename CH, typename LS, typename Sel
>
using GraspSkel = BuildSkeleton<GraspStruct, GraspLinks>::template skeleton<
  Pack<CH, LS, Sel>,
  Pack<Problem, Solution, PRNG>
>;
```

(⩾ C++14)

Extrait de code 5.7 – Squelette d'un GRASP

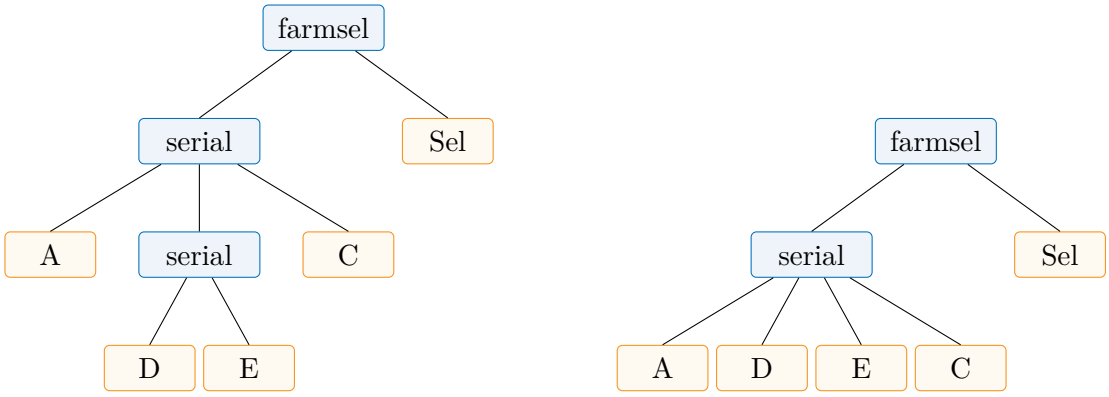
Plutôt que de décrire séparément la structure et les liens, le développeur pourrait se passer de cette étape de construction et écrire directement le squelette final. Cependant, cette méthode par assemblage permet de simplifier les éléments à définir en masquant les détails d'implémentation de la structure de données correspondant au squelette (chaque nœud de l'arbre comporte des données qui sont plus lourdes à écrire que la structure et les liens séparément). Il s'agit donc d'une abstraction de la représentation interne réelle qui permet au développeur d'exprimer plus aisément un squelette.

L'étape d'assemblage permet ainsi à la bibliothèque de modifier la structure des squelettes sans changer son interface. Cela peut être utilisé pour ajouter au sein des nœuds du squelette des informations déduites à partir de la structure ou des liens. Enfin, durant cette étape, le squelette peut être optimisé. Soient deux algorithmes X et Y dont la structure est telle que représentée, respectivement, par la figure 5.15a et la figure 5.15b. Il est possible d'utiliser l'algorithme Y comme muscle au sein de l'algorithme X (à l'instar de ce qui est accompli pour construire le GRASP×ELS, par exemple) en remplacement de la tâche « B » (figure 5.16a). On observe alors un os `Serial` exécutant un autre os `Serial`. Étant donné le fonctionnement de cet os, la figure 5.16b montre une structure équivalente dans son fonctionnement et réduisant le nombre d'intermédiaires. Cette optimisation automatique peut être accomplie durant la compilation, au moment de l'assemblage de la structure et des liens, par métaprogrammation [Pereda et al. 2020].



(a) Algorithme X (b) Algorithme Y

FIGURE 5.15 – Arbres représentant la structure de deux algorithmes



(a) Résultat brut de l'assemblage (b) Arbre optimisé

FIGURE 5.16 – Arbres représentant la structure d'un algorithme issu d'un assemblage

La séparation de la définition de la structure et des liens permet également d'imaginer des extensions à la bibliothèque pour automatiser, par exemple, la génération des liens à partir de la structure. Lorsque les liens d'un squelette sont définis, ceux-ci comportent entre autres l'information des os qui seront utilisés. Un outil au niveau du compilateur (extension ou programme agissant avant le compilateur) peut alors analyser cette description et générer le template de la structure. Une limite de ce procédé réside dans le mécanisme d'affectation automatique de noms aux paramètres représentant les muscles. Sans une compréhension globale de l'algorithme représenté, il n'est pas possible de nommer correctement ces muscles. Par ailleurs, au sein du langage lui-même, il n'est pas possible de générer un template, mais seulement des types, et c'est pourquoi la première solution proposée repose sur un outil externe. Cela peut tout de même être utilisé pour générer une structure déjà pourvue de muscles.

Si au contraire la structure est déjà définie et les liens inconnus, il est possible de proposer la génération automatique de liens. Celle-ci reposera sur des motifs établis, et le résultat sera alors similaire à ce que font par défaut les autres solutions de squelettes algorithmiques en ce qui concerne la manière dont les données sont transmises. Par exemple, il est possible d'indiquer qu'en cas d'os `serial` exécutant les n tâches $(T_0, \ldots, T_{n-1})$, il s'agit d'accepter en paramètres ceux de la première tâche (T_0), puis pour chaque tâche T_i $(i \geqslant 1)$ de prendre pour seul paramètre ce qui est retourné par la tâche T_{i-1}. Enfin, le retour global sera lié à celui de la dernière tâche, T_{n-1}.

Ces motifs peuvent cependant être fournis par le développeur (au maximum un par os). Ce faisant, un développeur voulant se contraindre à une forme pré-définie, par exemple pour réduire

la quantité de code à produire lors de l'écriture d'un nouveau squelette, en a la possibilité. Celle-ci reste non-intrusive puisque dans tous les cas, les liens peuvent être redéfinis localement afin de les adapter à un cas particulier.

Un squelette peut être complété en lui fournissant les types des paramètres et les muscles dont il a besoin. Pour l'ELS (extrait de code 5.8), les types des paramètres sont, dans l'ordre : le type représentant une instance du problème ; celui d'une instance d'une solution ; celui d'un générateur de nombres pseudo-aléatoires. Pour illustrer qu'il s'agit bien de types spécifiques au problème que l'on souhaite résoudre (pour le problème et la solution), un espace de noms est utilisé.

```
using ELS = ElsSkel<
  tsp::Problem, tsp::Solution, std::mt19937,
  Descent, Move2Opt, Descent, FN(selectMin), FN(selectMin), FN(acceptation)
>;
```

(⩾ C++14)

Extrait de code 5.8 – Corps d'un ELS

Les muscles doivent être des fonctionoïdes. En C++ cela inclut les fonctions et les instances de classes surchargeant au moins un `operator()`. Dans le cas des fonctions comme `selectMin` (extrait de code 5.9), une indirection est nécessaire afin de construire un type à partir de celle-ci. L'implémentation en C++14 ne peut être accomplie avec cette syntaxe légère qu'à l'aide d'une macro (`FN`), mais à partir de C++17 il est possible d'utiliser un template directement. Lorsqu'il s'agit déjà d'un type (comme c'est le cas des muscles `Descent` et `Move2Opt` dans l'extrait de code 5.8), celui-ci peut directement être utilisé.

```
Solution const& selectMin(Solution const& a, Solution const& b) {
  return a.value() < b.value()? a : b;
}
```

(⩾ C++98)

Extrait de code 5.9 – Fonction de sélection d'une solution à un TSP

En utilisant le même principe, l'extrait de code 5.10 fournit les types et muscles nécessaires au GRASP×ELS pour l'instancier. Cette instanciation produit un type (`GRASPxELS`) dédié à la résolution d'instances de TSP par l'utilisation de la métaheuristique GRASP×ELS.

```
// ELS est défini dans l'extrait de code 5.8.
using GRASPxELS = GraspSkel<
  tsp::Problem, tsp::Solution, std::mt19937,
  RGreedy<tsp::Solution>, ELS,
  FN(selectMin)
>;
```

(⩾ C++14)

Extrait de code 5.10 – Corps d'un GRASP×ELS

5.4.4 Hauteur parallèle du squelette

Afin d'illustrer ce qu'il est possible de faire, exclusivement durant la compilation, avec une telle information, nous proposons de calculer la « hauteur parallèle » de l'arbre, c'est-à-dire le nombre de niveaux parallélisables. Ces explications sont naturellement transposables à d'autres applications comme l'optimisation de squelettes assemblés.

Comme expliqué dans le chapitre 3, l'écriture est récursive en métaprogrammation pure puisqu'il s'agit exclusivement de programmation fonctionnelle. Cependant, afin d'assister le développeur, nous avons mis en place une bibliothèque annexe [4] permettant l'écriture de méta-programmes d'une manière plus impérative.

Cette bibliothèque annexe est utilisée dans l'écriture de certaines parties de la bibliothèque que présente ce chapitre, mais nous ne présenterons que son interface, l'implémentation étant fondée uniquement sur des principes expliqués dans le chapitre 3.

Une solution triviale pour obtenir cette hauteur parallèle est basée sur l'utilisation de la métafonction `treeAccumulate` (extrait de code 5.11). Cette métafonction accepte trois paramètres : l'arbre `T`, la métafonction `F` à appliquer et une valeur par défaut `A` à utiliser (notamment pour le cas d'un arbre vide). Le principe général de `TreeAccumulate` (à laquelle `treeAccumulate` fait appel pour extraire une valeur) est d'appeler la métafonction `F` en paramètre en lui donnant comme arguments le nœud courant `N` (racine de l'arbre en cours) ainsi que le résultat de celle-ci pour chacun des nœuds fils conservés dans le pack `Ns` (précisément, pour chacun des arbres dont les nœuds fils sont les racines). Ces résultats sont donc obtenus par un appel récursif à `TreeAccumulate`.

```
template<typename, template<typename, typename...> class, typename> struct
↪  TreeAccumulateImpl;

template<typename N, typename... Ns, template<typename, typename...> class F,
↪  typename A>
struct TreeAccumulateImpl<Tree<N, Ns...>, F, A> {
  using type = F<N, typename TreeAccumulateImpl<Ns, F, A>::type...>;
};

template<typename N, template<typename, typename...> class F, typename A>
struct TreeAccumulateImpl<Tree<T>, F, A> {
  using type = F<N, A>;
};

template<template<typename, typename...> class F, typename A>
struct TreeAccumulateImpl<Tree<>, F, A> {
  using type = A;
};

template<typename T, template<typename, typename...> class F, typename A>
using TreeAccumulate = typename TreeAccumulateImpl<T, F, A>::type;

template<typename T, template<typename, typename...> class F, typename A>
constexpr auto treeAccumulate = TreeAccumulate<T, F, A>::value;
```

(≥ C++14)

Extrait de code 5.11 – Définition de la métafonction `treeAccumulate`

Dans l'extrait de code 5.12, nous définissons donc comme hauteur par défaut la valeur 0 (nommée `CalcHeightDefault`). La métafonction `CalcHeight` sera appelée par `treeAccumulate` pour chaque nœud de l'arbre avec pour premier argument le nœud courant, `P`, et pour autres arguments l'ensemble des résultats de cette métafonction pour les nœuds fils. Deux cas doivent alors être distingués : s'il s'agit d'une feuille (cas `Leaf<T>`) la hauteur correspond à 0 (un muscle est séquentiel) ; s'il s'agit au contraire d'un nœud interne (cas `Branch<Skel>`), la hauteur

4. https://phd.pereda.fr/dev/tmp

```cpp
template<typename, typename...> struct CalcHeightImpl;

// Branch case: height = max of nodes + 1 if parallel
template<template<typename...> class Skel, typename... Ts>
struct CalcHeightImpl<Branch<Skel>, Ts...> {
  static constexpr int par = SkeletonTraits<Skel>::serial? 0 : 1;
  using type = std::integral_constant<int, par + max<Ts::type::value...>>;
};

// Leaf case: height = 0
template<typename T, typename... Ts>
struct CalcHeightImpl<Leaf<T>, Ts...> {
  using type = std::integral_constant<int, 0>;
};

template<typename P, typename... Ts>
using CalcHeight = typename CalcHeightImpl<P, Ts...>::type;

using CalcHeightDefault = std::integral_constant<int, 0>;

using Tree = TreeFromSkeleton<S>;
constexpr int value = treeAccumulate<Tree, CalcHeight, CalcHeightDefault>;
```

(⩾ C++14)

Extrait de code 5.12 – Utilisation de la métafonction `treeAccumulate` pour calculer la hauteur parallèle d'un squelette

parallèle correspond à la hauteur maximale des nœuds fils, augmentée de 1 si le nœud courant est capable de paralléliser son exécution. La métafonction `TreeFromSkeleton` est fournie par la bibliothèque (ainsi que sa réciproque `SkeletonFromTree`) pour simplifier le traitement des squelettes en permettant leur conversion en arbres.

Pour mieux se rendre compte de l'aspect impératif apporté par ces métafonctions, étudions ce même problème, résolu sans l'aide de la métafonction `treeAccumulate`. Cette solution alternative est implémentée dans l'extrait de code 5.13. Cette fois-ci, les métafonctions `checkParallelityEach` et `sumEach`, utilisées comme arguments aux métafonctions fournies, ne sont pas écrites (elles sont faciles à écrire et ne présentent pas d'intérêt pour l'explication).

```cpp
using Tree      = TreeFromSkeleton<S>;
using RtlPaths  = TreeAllRTLPaths<Tree>;
using Parallel  = Transform<RtlPaths, CheckParallelityEach>;
using ParallelC = Transform<Parallel, SumEach>;

using DefaultHeight = std::integral_constant<int, 0>;
constexpr int value = accumulate<ParallelC, Max, DefaultHeight>;
```

(⩾ C++14)

Extrait de code 5.13 – Calcul alternatif de la hauteur parallèle d'un squelette

En premier lieu, le squelette est converti en arbre. À partir de cet arbre, l'ensemble des chemins de la racine jusqu'à chaque feuille est construit avec la métafonction `TreeAllRTLPaths`. Ainsi, `RtlPaths` est un vecteur de vecteurs de nœuds de l'arbre. Chacun de ces nœuds est transformé en un booléen pour produire un vecteur de vecteurs de booléens, `Parallel`. Ce booléen sera vrai si le nœud correspond à un os correspondant à une exécution parallèle. Enfin, chaque vecteur contenu dans le vecteur principal est transformé en un entier correspondant au

nombre de booléens vrais qu'il contient. Grâce à ces transformations, nous avons alors un vecteur, `ParallelC`, contenant un entier pour chaque chemin possible de la racine de l'arbre à une de ses feuilles, lequel correspond aux nombres d'os autorisant du parallélisme qui existent sur ce chemin. Pour terminer, une projection de ce vecteur en un unique entier, par accumulation en conservant le maximum de tous les nombres, est obtenu grâce à la métafonction `accumulate`. La valeur par défaut intervient lorsque le vecteur `ParallelC` est vide.

5.5 Politique d'exécution

Jusqu'à présent, nous avons vu comment un squelette peut être assemblé par un développeur et comment lui adjoindre des muscles pour définir les tâches effectives à exécuter, formant ainsi un corps. Le squelette est composé d'une structure formée d'os et de liens indiquant la nature des flux de données entre les muscles. Deux points sont restés indéfinis : l'implémentation des os, c'est-à-dire l'application du motif d'exécution censé être décrit par l'os, et la stratégie à adopter pour l'exécution des muscles, c'est-à-dire la manière de répartir le travail sur les processeurs à disposition.

Les politiques d'exécution sont fournies au moyen de classes, des exécuteurs, présentant une interface qui peut être utilisée dans l'implémentation des os. Comme représenté par la figure 5.17, un os n'exécute pas directement lui-même ses muscles. Cette indirection est nécessaire pour dissocier les os de la politique à suivre pour l'exécution des muscles qui lui sont associés.

Nous profitons de ceci pour masquer une difficulté inhérente au travail que doit effectuer notre bibliothèque : l'appel aux fonctions en respectant les liens fournis pour la transmission des paramètres. Dans la figure 5.17, le « système interne » correspond à un template dont le but principal est d'appliquer les liens. Pour cela sont utilisés différents n-uplets (dont l'ensemble de paramètres en entrée et l'ensemble des valeurs retournées) au sein desquels sont sélectionnés les bons éléments, dans le bon ordre, pour être utilisés dans un appel de fonctionoïde. Le fonctionoïde dont il s'agit correspond alors soit directement à un fonctionoïde fourni par le développeur, soit au résultat de l'implémentation d'un sous-squelette. Ces deux aspects (l'implémentation automatique, si nécessaire, et la gestion de l'appel effectif en fournissant le bon ensemble d'arguments) ne semblent pas intéressant à présenter en détail et seront donc simplement regroupés sous le nom de système interne.

Dans un premier temps, pour chaque muscle d'un os, ce dernier génère les instances (étape 1 de la figure) du système interne qui lui sont dédiées. Ces instances sont ensuite utilisées, conjointement avec d'éventuels autres arguments, pour faire appel aux primitives procurées par une politique d'exécution (étape 2). En effet, chaque politique d'exécution doit rendre publique une interface commune de fonctions qui pourront être utilisées pour exprimer comment doivent être exécutés des ensembles de tâches. Par exemple, on pourra demander une exécution parallèle de multiples tâches ou bien une exécution parallèle d'une même tâche répétée plusieurs fois.

Les politiques d'exécution utilisent l'instance du système interne (étape 3), fournie par l'os, qui permet d'exécuter un muscle (indiqué par l'os) simplement : l'implémentation du muscle, si celui-ci correspond en fait à un autre os (étape 4), ainsi que la gestion des liens sont automatisés, comme expliqué ci-avant. L'implémentation de l'étape 4, qui arrive donc uniquement lorsque le muscle à exécuter est un os, utilise la partie « implémentation » de l'os pour produire un muscle.

Cette section présente les éléments importants de ce système, en commençant par l'écriture de la spécialisation du template dédié à l'implémentation des os pour certains d'entre eux afin

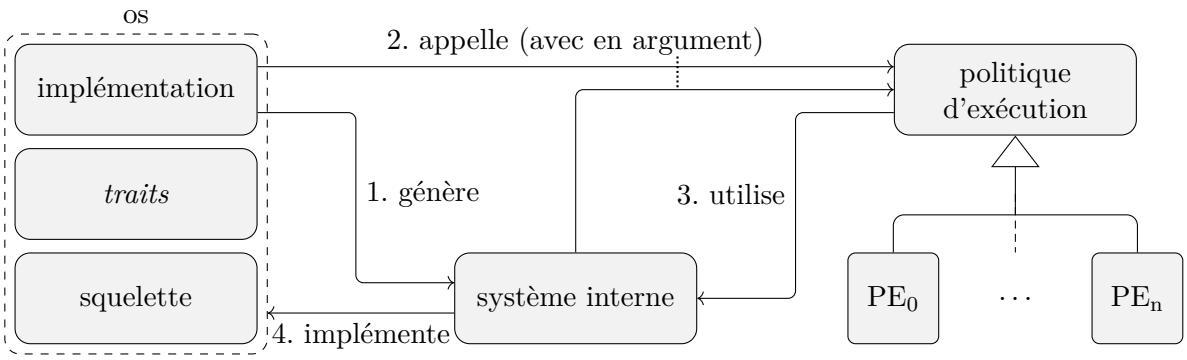


FIGURE 5.17 – Diagramme d'interaction des os avec les exécuteurs

d'identifier les fonctions qui doivent être pourvues par un exécuteur. Ensuite, elle expose le principe général des exécuteurs pour terminer par des exemples concrets de ce qu'un développeur peut écrire pour gérer lui-même, si nécessaire, le procédé à suivre durant l'exécution.

5.5.1 Implémentation des os

Les os représentent des motifs d'exécution (une série de tâches, une répétition parallèle d'une tâche, ...). Pour que la composition de squelettes, basée sur les os, fonctionne, il faut que ces os puissent se comporter comme des muscles. C'est-à-dire qu'il faut qu'un os puisse être exécuté comme une fonction.

Pour cela, la définition d'un os se fait en plusieurs parties, comme cela a été introduit dans la section 5.4.1. La première est un template de classe, comme dans l'extrait de code 5.14 par exemple pour `Farm`, qui correspond à la partie « squelette » de l'os dans la figure 5.17. Celui-ci est paramétré au minimum par la signature de la fonction qui sera associée à l'os (`Signature`) et par l'ensemble des muscles et de leurs signatures : le pack avec `Task` et `TaskLinks`, respectivement le muscle et la signature associée qui seront exécutés par `Farm`. Pour un os disposant de plusieurs tâches, ce pack est répété autant de fois qu'il y a de tâches.

```
template<typename, typename> struct Farm;

template<typename Signature, typename Task, typename TaskLinks>
struct Farm<Signature, Pack<Task, TaskLinks>>: BoneBase {
  Task        task;
  std::size_t n;
};
```

(⩾ C++11)

Extrait de code 5.14 – Os `Farm`

La classe générée doit permettre de définir l'état de l'os, par exemple pour le cas de `Farm` il y aura, en plus de l'instance de la tâche `task`, le nombre de fois `n` que la tâche `task` devra être exécutée. L'instance de `Task` est utile principalement lorsqu'elle correspond elle-même à un autre os, lequel a donc également besoin d'un état.

La partie « *traits* » de l'os dans la figure 5.17 est également importante pour le bon fonctionnement de la bibliothèque. Chaque os doit spécialiser un template de classe de *traits* (`SkeletonTraits`) pour notamment indiquer si l'os possède un comportement obligatoirement séquentiel ou non. L'extrait de code 5.15 montre cela pour l'os `Farm` qui peut, si la politique

d'exécution le permet, être exécuté en parallèle.

```
template<>
struct SkeletonTraits<Farm> {
  static constexpr bool serial = false;
  // ...
};
```

(⩾ C++11)

Extrait de code 5.15 – Spécialisation du template de classe de *traits* pour l'os `Farm`

Enfin, la partie « implémentation » de l'os dans la figure 5.17 est ce qui permet d'utiliser l'os comme une fonction. Cela se fait par un ensemble de spécialisations d'un autre template (`Impl`), lesquelles définissent le comportement de l'os en fonction de certains critères. Essentiellement, chacune de ces spécialisations consiste en un fonctionoïde dont l'`operator()` accepte les paramètres attendus par la signature de fonction affectée à l'os, et retourne, de même, le type retourné indiqué dans cette signature.

Pour le cas d'os simples comme `Farm`, il existe des primitives dans l'interface des politiques d'exécution correspondant directement au patron à respecter. Ainsi, que ce soit pour le cas séquentiel (voir l'extrait de code 5.16) ou le cas parallèle (voir l'extrait de code 5.17), un appel de fonction suffit. Dans le premier cas, il s'agit d'exécuter de manière séquentielle de multiples fois l'unique tâche. Dans le second cas, l'exécution est faite en parallèle.

```
template<typename... Args, typename TTask, typename Executor>
struct Impl<Farm<void(Args...), TTask>, tag::Sequential, Executor>:
BoneImplBase<Farm<void(Args...), TTask>, tag::Sequential, Executor>
{
  using This  = Impl;
  using Task  = Execute<typename This::Skeleton::TaskLinks>;

  typename This::Skeleton skeleton;
  typename This::Executor executor;

  void operator()(Args... args) {
    executor.template executeSequential<Task>(
      *this, skeleton.task, std::forward_as_tuple(args...), skeleton.n
    );
  }
};
```

(⩾ C++11)

Extrait de code 5.16 – Implémentation de l'os `Farm` pour le cas séquentiel

Les extraits de code 5.16 et 5.17 diffèrent par rapport au critère représenté par le deuxième paramètre du template `Impl` : un *tag* indiquant si la spécialisation concerne une exécution séquentielle (`tag::Sequential`) ou une exécution parallèle (`tag::Parallel`). Ainsi, dans le premier cas on appellera la primitive `executeSequential` de la politique d'exécution, tandis que dans le second cas il s'agira d'utiliser `executeParallel`. À noter que la politique d'exécution peut tout à fait procéder à une exécution séquentielle y compris dans le second cas, notamment dans le cas évident où il s'agit d'une politique d'exécution séquentielle.

```
template<typename... Args, typename TTask, typename Executor>
struct Impl<Farm<void(Args...), TTask>, tag::Parallel, Executor>:
BoneImplBase<Farm<void(Args...), TTask>, tag::Parallel, Executor>
{
  // ...

  void operator()(Args... args) {
    executor.template executeParallel<Task>(
      *this, skeleton.task, std::forward_as_tuple(args...), skeleton.n
    );
  }
};
```

(⩾ C++11)

Extrait de code 5.17 – Implémentation de l'os `Farm` pour le cas parallèle

La classe dont hérite la spécialisation du template `Impl` (à laquelle on accède au travers du type `This`) fournit la partie « squelette » de l'os, grâce notamment au pack `TTask` contenant les muscles et leurs signatures. En l'utilisant, précisément la partie « liens » que l'on obtient depuis ce squelette, on définit `Task` qui sert d'interface avec le « système interne » : le template `Execute` est le point d'entrée de ce système. Ce type `Task` est donc fourni à la primitive de la politique d'exécution comme argument template.

Un os plus complexe peut utiliser plusieurs primitives, c'est le cas par exemple de `While`. Celui-ci exécute le muscle permettant de déterminer s'il faut continuer, puis, le cas échéant, la tâche principale. L'ensemble des primitives des politiques d'exécution que nous avons déterminées comme nécessaires pour l'écriture des os dont dispose la bibliothèque comporte ainsi l'exécution :

— d'une tâche ;
— d'une tâche répétée (`executeSequential`) ;
— d'une tâche répétée en parallèle (`executeParallel`) ;
— d'une tâche répétée en parallèle suivi d'une réduction ;
— d'un ensemble de tâches en parallèle.

5.5.2 Identification des tâches

Comme expliqué, les muscles ne sont pas directement utilisables. S'il s'agit d'un fonctionoïde, rien n'est à faire avant de pouvoir l'exécuter, mais, en revanche, s'il s'agit d'un os, il est nécessaire dans un premier temps de l'implémenter pour obtenir un fonctionoïde. Ainsi, c'est finalement dans les deux cas un l'appel à un fonctionoïde qui devra être fait.

Pour procéder à cet appel, il faut correctement positionner les différents paramètres (indiqués par le template `P` lors de la définition des liens) et résultats (indiqués par le template `R`) dans la liste des arguments du fonctionoïde. Ceci est automatiquement effectué par le système interne et repose, en simplifiant un peu, sur de l'expansion de packs templates.

Cette étape permet en outre de produire des méta-informations sur le squelette. L'une de ces données, dont nous nous servirons par la suite, est un identifiant numérique associé à chaque tâche, muscle et os. Cet identifiant possède comme propriété d'être unique dans un contexte parallèle quelconque : deux tâches concurrentes ne peuvent partager le même identifiant. En d'autres termes, si deux tâches s'exécutent obligatoirement en séquence (c'est-à-dire s'il existe une dépendance entre elles) elles auront un identifiant identique, sinon celui-ci sera différent.

Il est possible de parvenir à cette affectation en déterminant quels os sont parallélisables.

Lorsqu'un os séquentiel est instancié, il transmet son propre identifiant à chacun de ses muscles, tandis qu'un os pouvant être parallèle leur donne des identifiants distincts. Le calcul de l'identifiant dans ce dernier cas dépend alors de l'existence de niveaux de parallélisation plus profonds.

Afin d'expliquer notre méthode, nous allons prendre l'exemple du GRASP×ELS représenté dans la figure 5.18 et annoté avec les ensembles desquels sont issus les identifiants associés à chaque nœud. L'identifiant de départ, affecté à l'os racine `farmsel`$_1$, est 0. Celui-ci est transmis tel quel au muscle `Sel`$_1$ puisque celui-ci ne peut pas être exécuté en parallèle d'une autre tâche. Pour déterminer les identifiants donnés aux a instances de `serial`$_1$, il est nécessaire de parcourir l'arbre à partir de ce nœud afin de trouver un éventuel autre os parallélisable. Dans ce cas, il y en a effectivement un, `farmsel`$_2$, exécutant b tâches parallèles.

Lorsqu'un cas de parallélisation à plusieurs niveaux est ainsi trouvé, par exemple lorsqu'un nœud P_1, correspondant à un os parallélisable, possède un nœud descendant P_2, correspondant aussi à un os parallélisable, le nombre de tâches pouvant être exécutées en parallèle correspond au produit des nombres de tâches que vont exécuter P_1 et P_2.

Ainsi, au niveau de `farmsel`$_1$, le nombre de tâches exécutées en parallèle est de $a \times b$: chaque `serial`$_1$ permettra l'exécution parallèle de b tâches. Les a identifiants donnés aux instances de `serial`$_1$ sont donc générés à partir de celui de `farmsel`$_1$ (0) et espacés de b et sont donc $\{0, b, 2\,b, \ldots, (a-1)\,b\}$.

Toutes les tâches d'un même niveau de parallélisation partagent le même identifiant, donc, pour une instance donnée de `serial`$_1$, les instances correspondantes des tâches qui ne sont exécutées qu'une seule fois dans cette instance (`HC`, `serial`$_2$, `RLI`, `itersel`, `farmsel`$_2$, `Sel`$_2$ et `Sel`$_3$) auront un identifiant unique (appartenant à l'ensemble ci-avant).

L'affectation au niveau de `farmsel`$_2$ suit le même principe que pour le premier os parallélisable, mais puisqu'aucun descendant ne permet la parallélisation, le pas utilisé pour espacer les identifiants est 1. Ainsi, pour l'instance de `farmsel`$_2$ d'identifiant 0, ce seront les nombres de 0 à

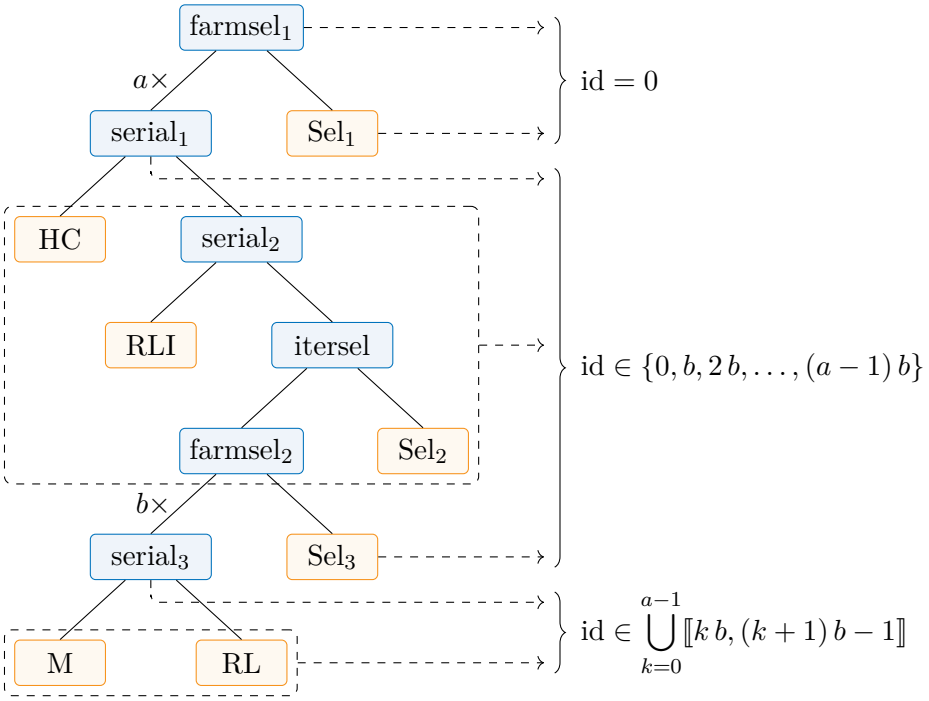


Figure 5.18 – Identification des tâches d'un squelette de GRASP×ELS

$b-1$, pour celui d'identifiant b, de b à $2b-1$, et ainsi de suite jusqu'à celui d'identifiant $(a-1)\,b$ qui produira les valeurs de $(a-1)\,b$ à $a\,b-1$.

5.5.3 Exécuteur

Le rôle rempli par les exécuteurs est donc d'implémenter une politique d'exécution, c'est-à-dire une manière d'exécuter les muscles d'un squelette sur les différents cœurs qui sont affectés. Pour cela, nous avons vu qu'un ensemble de schémas de parallélisation doivent être fournis. Ces différentes primitives seront présentées pour des cas concrets par la suite. Nous allons dans un premier temps nous intéresser aux raisons de fournir différentes politiques d'exécutions.

Considérons un squelette dont l'os extérieur est un `Farm` exécutant $N \in \mathbb{N}^*$ fois sa tâche. Nous supposons que les instances de cette tâche ont toutes le même temps d'exécution D. La figure 5.19 schématise l'exécution séquentielle d'un programme : un même *thread* traite les tâches l'une après l'autre. Le temps total d'exécution sera donc égal à $N \times D$.

Si l'on dispose de $K \in \mathbb{N}^*$ cœurs, notamment lorsque $K > 1$, il faut décider du nombre de *threads* à créer. La valeur optimale dépend de ce que fait le programme, et nous n'allons pas discuter de ce problème ici. Le développeur utilisant cette bibliothèque peut définir lui-même cette valeur, et nous allons pour cette section seulement considérer le nombre $T \in \mathbb{N}^*$ de *threads* qu'il est possible de créer.

Supposons que l'on dispose au maximum de T *threads* pour exécuter N tâches ($N, T \in \mathbb{N}^*$), il existe $n \in \mathbb{N}$ et $r \in [\![0, T[\![$ tel que $N = n\,T + r$. Une répartition équilibrée consiste alors à donner à r *threads* $n+1$ tâches à traiter et n aux $T-r$ autres *threads*. Lorsque $T > N$, on a $n = 0$ et $r = N$ et il y a donc $T - N$ *threads* auxquels sont affectées 0 tâches. Dans ce cas, seulement N *threads* sont créés.

Selon la valeur de r, deux cas sont possibles. Si $r = 0$ alors les T *threads* exécutent $n = \frac{N}{T}$ tâches et le temps total d'exécution devient $n \times D$, c'est-à-dire $\frac{1}{T} \times (N \times D)$, permettant donc une accélération théorique de T. Sinon, si $r \neq 0$ alors il existe au moins un *thread* qui exécute $n+1$ tâches où $n = \left\lfloor \frac{N}{T} \right\rfloor$, donc $n = \frac{N-r}{T}$. Le temps total d'exécution, toujours pour des durées égales entre toutes les tâches, devient $(n+1) \times D$, c'est-à-dire $\frac{N+T-r}{T} \times D = \frac{N+T-r}{N\,T} \times (N \times D)$. L'accélération théorique est alors égale à $\frac{N\,T}{N+T-r}$. Elle tend vers T quand r s'approche de T et est minimale lorsque $r = 1$.

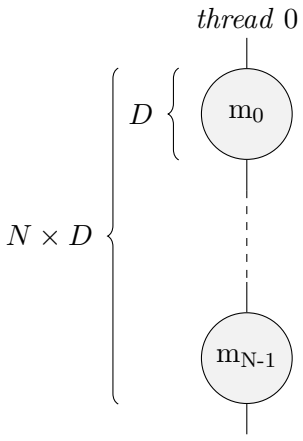


FIGURE 5.19 – Exécution séquentielle de N tâches

Par exemple, si l'on dispose de $K = 6$ cœurs et que l'on s'autorise à créer $T = 6$ *threads* pour exécuter $N = 15$ tâches, on aura alors, comme représenté sur la figure 5.20, $n = \left\lfloor \frac{15}{6} \right\rfloor$ donc $n = 2$ tâches exécutées au minimum par chacun des *threads*, et $r = 15 - 6 \times 2$ soit $r = 3$ *threads* qui en exécutent une de plus. Ainsi, 3 *threads* exécutent 3 tâches tandis que les 3 derniers n'exécutent que 2 tâches.

Si les tâches ne sont pas équilibrées en temps d'exécution, une telle procédure, sans connaissance *a priori* du déséquilibre (par exemple sous forme de poids attribué à chaque tâche), produira potentiellement une répartition peu efficace sur les *threads*. Si la connaissance *a priori* ne peut être acquise, une répartition dynamique basée sur le concept de *thread pool* permet généralement de répondre au besoin. Nous verrons cependant que cette solution possède des inconvénients et induit un surcoût inévitable.

Sur l'exemple précédent, il est possible de libérer le *thread* 5 en déplaçant les tâches qui lui sont affectées sur les *threads* 3 et 4 sans que cela n'augmente le temps d'exécution total (toujours dans l'hypothèse d'un temps d'exécution constant D). Cela est représenté par la figure 5.21. En général, cette altération est possible lorsqu'il existe un reste $r \neq 0$ (de sorte qu'au moins un *thread* soit inoccupé à la dernière ligne) et que $n \leqslant T - r - 1$ (de sorte que les n tâches affectées au *thread* T puissent être attribuées aux *threads* $r + 1$ à $T - 1$). Pour cet exemple, appliquer cette modification permet seulement de consommer effectivement au mieux un *thread* de moins, le laissant ainsi libre pour d'autres applications. En pratique, cela peut également changer le temps d'exécution, l'améliorant (par exemple en permettant de bénéficier d'une meilleure localité des données) ou le détériorant (par exemple si certaines tâches se retrouvent bloquées en attente d'une ressource). Ces variations étant très liées au programme concerné, c'est une décision qui doit être prise au cas par cas, en profilant l'exécution.

Jusqu'ici, nous avons seulement envisagé un niveau de parallélisation possible. Or, et c'est le cas justement du GRASP×ELS qui nous sert d'exemple, de nombreux algorithmes offrent plusieurs niveaux de parallélisation. La figure 5.22 reprend la figure 5.20 en détaillant les tâches. On considère alors que chacune des $N = 15$ tâches exécute en fait $M = 2$ tâches internes qui sont indépendantes et peuvent donc être exécutées en parallèle. L'accélération théorique obtenue est de $\frac{15 \times 2 \times D}{3 \times 2 \times D} = 5$ en utilisant 6 *threads*. Il apparaît alors que la solution proposée ci-avant semble pouvoir être améliorée.

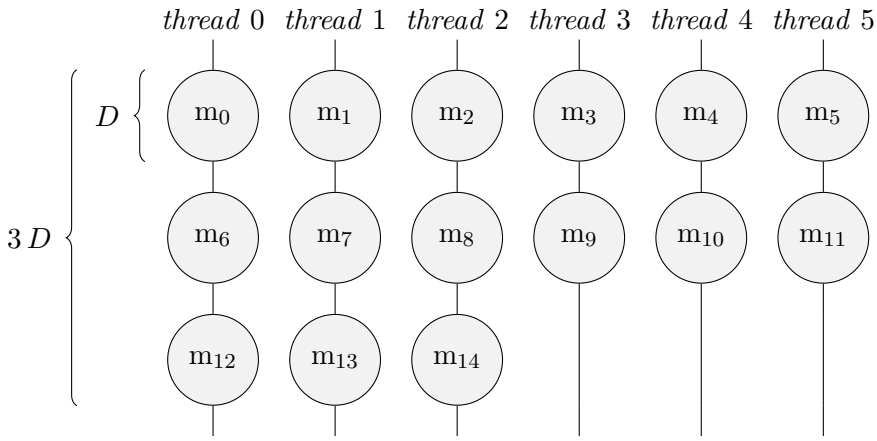


FIGURE 5.20 – Exécution parallèle de 15 tâches sur 6 *threads*

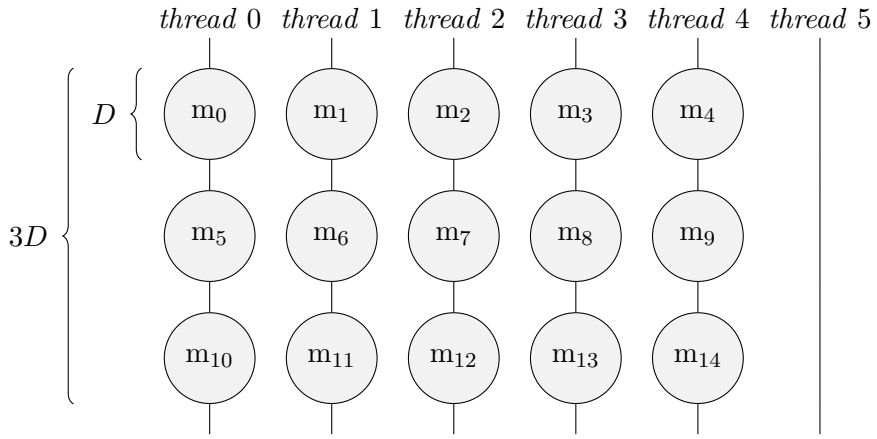


FIGURE 5.21 – Exécution parallèle plus dense de 15 tâches sur 6 *threads*

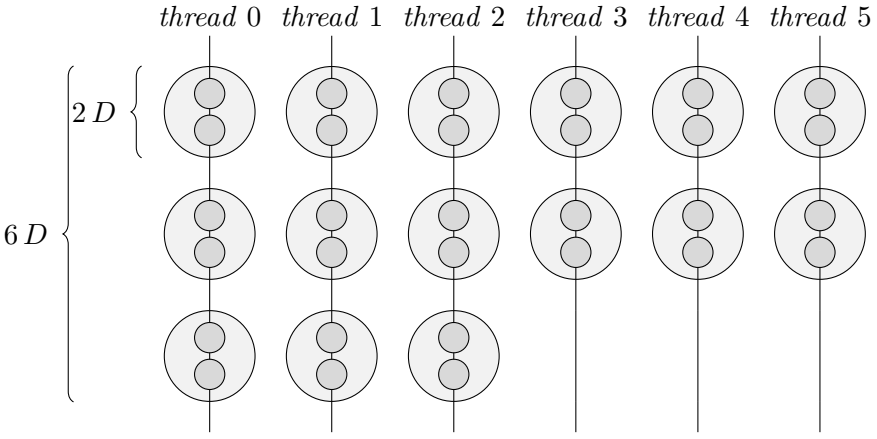


FIGURE 5.22 – Exécution parallèle de 15 tâches exécutant 2 tâches internes sur 6 *threads*

En effet, en ayant connaissance du niveau de parallélisation interne, il est possible d'obtenir une répartition des tâches semblable à ce que présente la figure 5.23. Ainsi, l'accélération théorique obtenue devient $\frac{15 \times 2 \times D}{5 \times D} = 6$.

Pour généraliser, en partant de la répartition proposée initialement pour le premier niveau (figure 5.20), il faut améliorer la répartition des tâches « restantes », c'est-à-dire celles qui ne complètent pas la dernière ligne, le cas échéant. Il faut donc intervenir lorsque $r \neq 0$. Il s'agit alors de déterminer combien de *threads* peuvent utiliser chacune des tâches restantes pour l'exécution de leurs tâches internes. Pour un équilibrage simple (où l'on affecte à toutes ces tâches autant que possible le même nombre de *threads*), on peut déterminer T^r, le nombre de *threads* minimal par tâche, et un reste $r' \in [\![0, r[\![$, le nombre de tâches qui disposeront d'un *thread* supplémentaire, ainsi : $T = r\,T^r + r'$. Il faut donc affecter $T^r + 1$ *threads* à r' tâches et T^r *threads* à $r - r'$ tâches, ce qui utilise effectivement $(T^r + 1)\,r' + T^r\,(r - r') = r\,T^r + r'$ *threads*, c'est-à-dire T. Chaque tâche i restante ayant à sa disposition $T_i^r > 1$ *threads* pourra procéder au même calcul pour répartir ses tâches internes.

Durant la compilation, un exécuteur peut détecter l'existence d'un niveau parallélisable au sein d'une branche de l'arbre que représente le squelette. En utilisant cette information, il lui est possible de ne pas affecter plusieurs *threads* à une tâche si celle-ci ne peut pas en tirer profit. La répartition des valeurs T_i^r peut ainsi être adaptée en tenant compte de ce critère. En utilisant en complément une information partiellement dynamique comme le nombre de tâches exécutées par une branche de l'arbre, l'exécuteur peut équilibrer encore davantage les valeurs T_i^r.

En pratique, il n'est pas toujours judicieux d'utiliser une politique d'exécution parallélisant plusieurs niveaux. Prenons par exemple un cas où les deux premiers niveaux sont séparés par un os séquentiel répétant l'exécution de sa tâche principale. Si cet os séquentiel possède un très grand nombre d'itérations, le deuxième niveau parallélisable sera instancié autant de fois, et causera, selon la politique d'exécution, la création de très nombreux *threads*, la surcharge d'une *thread pool*, ... Il n'existe pas une politique d'exécution idéale qui pourrait être utilisée en toutes circonstances.

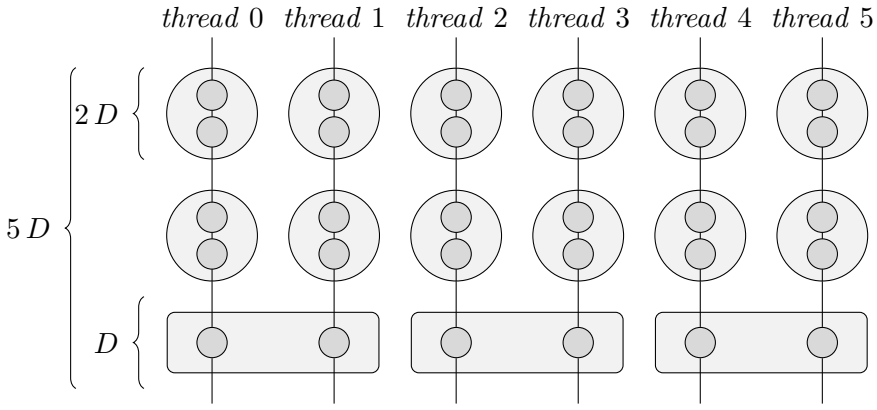


FIGURE 5.23 – Exécution parallèle améliorée de 15 tâches exécutant 2 tâches internes sur 6 *threads*

5.5.4 Implémentation d'un exécuteur

Nous allons aborder ici le cas de deux politiques d'exécution : celui du *thread pool* et celui de l'orchestration statique tenant compte de multiples niveaux de parallélisme. Ces deux manières de procéder permettent d'explorer les deux aspects opposés d'un exécuteur : le traitement effectué durant la compilation et la capacité à utiliser des informations dynamiques, connues uniquement lors de l'exécution du programme.

5.5.4.1 *Thread pool*

Le *thread pool* repose sur une file de tâches partagée par un ensemble de *threads* dont le rôle est de traiter les tâches. Tous les accès à cette file doivent être protégés par un *mutex* (voir la section 1.3.3.2). Un exécuteur doit donc pouvoir accéder à un état partagé, lequel comporte la file de tâches et implémente le *thread pool* en ayant une interface permettant, principalement, de soumettre des tâches (figure 5.24).

Au sein du *thread pool*, dès que l'un des *threads* du *pool* est disponible (parce qu'il n'a pas encore traité de tâche ou qu'il en a terminé une), il essaie d'acquérir le *mutex* pour utiliser la file. Si la file n'est pas vide, le *thread* retire la plus ancienne tâche pour l'exécuter. Sinon, il se met en attente et sera réveillé lorsque la file recevra une nouvelle tâche, après que les éventuels autres *threads* auparavant en attente auront eux-mêmes été débloqués.

Ce fonctionnement permet au *thread pool* d'équilibrer automatiquement et dynamiquement la charge des *threads*. En revanche, l'accès à la file des tâches, à laquelle accèdent tous les *threads*, cause des sections critiques lors de l'exécution si de multiples *threads* deviennent disponibles ou que des tâches doivent être soumises concomitamment. Cela signifie que de très nombreuses tâches relativement courtes causeront un goulot d'étranglement, affectant les performances et l'accélération qui peut être obtenue.

À cela s'ajoute la perte de certaines optimisations due au fonctionnement interne d'un *thread pool*. Pour pouvoir conserver et utiliser les tâches, qui sont des fonctionoïdes, il faut que celles-ci soient de même type car le conteneur (la file) est homogène. Cela passe par l'effacement du type réel du fonctionoïde, ce qui induit une indirection dynamique de l'appel.

Certains exécuteurs peuvent avoir besoin d'un état partagé (pour le *thread pool*, il s'agit de la file des tâches et de l'ensemble des *threads*). Cet état est défini par une spécialisation d'un template de classe propre à chaque exécuteur. Une instance de ce type est ensuite rendue

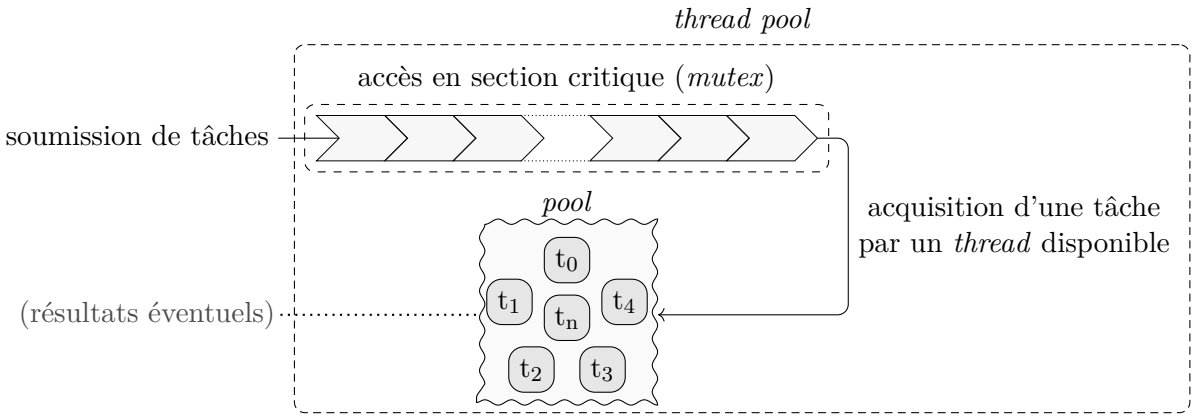


FIGURE 5.24 – Schéma d'un *thread pool*

accessible aux fonctions qui doivent être implémentées par l'exécuteur (précisément, une référence sur une instance unique). Cette séparation entre l'exécuteur lui-même et son état permet à la bibliothèque d'éviter l'existence d'une référence commune de l'exécuteur et simplifie certains détails d'implémentation internes.

L'une de ces fonctions membres de la classe de l'exécuteur est `executeParallel` et correspond à l'exécution d'une tâche n fois en parallèle. Une définition simplifiée est donnée dans l'extrait de code 5.18. On peut y observer l'accès à l'état de l'exécuteur (`state`). Les « futures » (voir la section 1.5.1) sont créées par la fonction `run` servant à soumettre une nouvelle tâche à exécuter. Ces « futures » permettent de synchroniser la terminaison des tâches pour cette exécution.

```
template<
  typename Task,
  typename Impl, typename Muscle, typename Params
>
void executeParallel(
  Impl& impl, Muscle& muscle, Params const& params, std::size_t n
) {
  std::vector<std::future<void>> futures(n);

  typename Impl::State& state = impl.state;

  for(std::size_t i = 0; i < n; ++i) {
    futures[i] = state.executor.run([&]{
      Task::execute(impl, task, i, params);
    });
  }

  state.executor.wait(futures);
}
```

(⩾ C++14)

Extrait de code 5.18 – Implémentation rudimentaire de la fonction membre `executeParallel` (par *thread pool*)

Pour cela, il faut attendre la réalisation de chaque « future ». Cependant, dans un contexte général où plusieurs niveaux de parallélisation peuvent exister, une tâche peut elle-même ajouter un ou plusieurs autres éléments à la file des tâches. Sans action particulière pour gérer ce cas, cela engendre un blocage du système. Pour cela, si le *pool* possède T *threads*, il suffit par exemple que T tâches A^i, $i \in [\![0, T[\![$ aient à exécuter elles-mêmes plus d'une (mettons k) tâche a^i_j, $j \in [\![0, k[\![$. Les T tâches A^i pouvant s'exécuter en même temps, il est possible qu'elles occupent déjà les T *threads* du *pool*. Chacun des T *threads* exécute une tâche A^i et ajoute à la file des tâches ses a^i_j tâches puis se met en attente de leur terminaison. Or, sans un mécanisme de préemption, puisqu'aucun *thread* n'est alors disponible, aucune des tâches a^i_j ne va s'accomplir et le système sera bloqué. Pour cette raison, l'attente des « futures » est effectué par une fonction *ad hoc* du *thread pool* (`state.executor.wait`) qui résout ce problème en faisant travailler le *thread* lorsque cela est nécessaire avant d'attendre les « futures ». Il s'agit de déterminer, avant de se mettre en attente, si la file de tâches ne contient aucune tâche dont dépendrait une des « futures » à attendre, ce qui peut être vérifié en pratique en s'assurant que la file des tâches est vide. Si elle ne l'est pas, le *thread* courant doit alors étendre sa durée de vie au sein du *pool* et traiter une nouvelle tâche avant d'effectuer à nouveau le test susmentionné.

5.5.4.2 Multi-niveau statique

Par « statique », pour une politique d'exécution, il est entendu que chaque tâche est associée à un *thread* (ou l'équivalent s'il ne s'agit pas de parallélisation par *threads*) de manière stable et dès la compilation. Ceci offre différents avantages, notamment lorsqu'il s'agira de s'intéresser à la répétabilité dans la section 5.6.

En restant sur le concept de *pool*, un ensemble de *threads* peut être créé à l'avance et une projection de l'identifiant de chaque tâche (voir la section 5.5.2) sur un numéro de *thread* permet de faire une orchestration statique. Alternativement, il est possible de produire dynamiquement le numéro de *thread* au sein de l'exécuteur (la répartition reste définie durant la compilation). Ces deux méthodes offrent différents avantages sur les caractéristiques de répartition des tâches, ce qui nous sera à nouveau utile pour la répétabilité.

Les *threads* peuvent également être créés au besoin. Pour aller plus loin, il serait possible de créer les *threads* en définissant dès la compilation leur déroulement au moment où l'on conçoit la fonction qu'ils exécuteront. Celle-ci comporterait les appels de chaque tâche devant être traitée par un *thread* donné en suivant une orchestration définie (par exemple celle représentée par la figure 5.23) et en utilisant des barrières de synchronisation lorsque nécessaire.

5.6 Répétabilité

Le fonctionnement de la recherche scientifique, du moins dans la forme actuelle de la méthode scientifique expérimentale, repose sur plusieurs principes : la réfutabilité, la non-contradiction et la reproductibilité [Popper 2005]. Une expérience mène à des résultats, et ceux-ci doivent être interprétés pour obtenir une conclusion. En effectuant cela une seule fois, la conclusion est dépendante de nombreux paramètres : variations dues à des effets aléatoires ou non considérés ; erreurs de manipulation, de mesure ; biais auxquels peuvent être sujets les chercheurs ; fraude... En faisant reproduire l'expérience, dans des contextes différents, par des personnes différentes, il est possible de réduire les effets de ces paramètres et donc de valider la qualité du travail si la même conclusion scientifique est atteinte.

Ce critère de reproductibilité n'est cependant pas respecté systématiquement, et, dans certains domaines, il a même été observé qu'un grand nombre de publications présentent des expériences ne pouvant pas être reproduites [Ioannidis 2005], au point que le problème est connu sous le nom de « crise de la reproductibilité ». Le domaine de la recherche en informatique n'est pas épargné, mais celui-ci est inquiété par un problème supplémentaire : le manque de répétabilité.

Les deux concepts, reproductibilité et répétabilité, sont quelquefois confondus [Drummond 2009] mais il est nécessaire de bien les distinguer. Alors que la reproductibilité ne s'intéresse qu'à l'obtention d'une même conclusion à partir de plusieurs itérations d'une expérience, celle-ci pouvant donc être faite dans des conditions différentes, la répétabilité signifie l'obtention d'un même résultat lorsque les conditions sont identiques. En informatique, ces conditions représentent le contexte d'exécution. Elles incluent le matériel, le système d'exploitation ou encore les bibliothèques logicielles utilisées.

La répétabilité est indispensable afin de pouvoir raisonner sur le comportement d'un programme lors de son exécution : sans cela, il n'est par exemple même plus possible de déboguer et c'est tout l'intérêt de l'automatisation informatique qui est remis en cause. Dans le cas de programmes séquentiels, elle est en général assez facile à garantir, mais la parallélisation a

introduit de nouveaux problèmes. Par exemple, l'exécution dynamique (voir la section 1.4.1.2) peut causer des variations dans les résultats numériques en nombres flottants à cause de la non associativité des opérations associées, pouvant donc rendre l'exécution d'un programme non répétable. D'autres optimisations matérielles peuvent aboutir à cette non répétabilité, mais celles-ci ne seront pas traitées dans cette thèse, qui propose des solutions logicielles au problème de la répétabilité induite par l'introduction de la parallélisation.

Les nombres aléatoires sont utilisés dans de nombreux domaines dont celui de la RO. Plus exactement, ce sont des nombres pseudo-aléatoires qui sont généralement utilisés : cela signifie qu'ils sont générés par un algorithme déterministe, et qu'il est donc possible d'exécuter plusieurs fois un programme avec la même séquence de nombres. Pour cette raison, l'usage de nombre pseudo-aléatoires ne pose généralement pas de problème de répétabilité. Cependant, ce n'est plus le cas lorsqu'ils sont utilisés au sein d'un programme exécuté en parallèle [D. R. C. Hill et al. 2013]. La figure 5.25 présente ce qu'il peut arriver lors de l'utilisation d'un flux de nombres pseudo-aléatoires partagé entre deux *threads* : selon l'exécution, les nombres obtenus par un *thread* ne sont pas les mêmes.

Il est possible d'assurer la répétabilité de plusieurs manières. Celle-ci pourrait être limitée à l'obtention d'un résultat identique pour un degré de parallélisme donné qui ferait alors partie du contexte d'exécution. Cela ne suffirait cependant pas pour obtenir un programme scientifiquement valide, ce qui requiert une répétabilité par rapport à une exécution séquentielle [D. R. C. Hill 2015]. Pour cela, il est possible de considérer le degré de parallélisme comme ne faisant pas partie du contexte d'exécution, la répétabilité devant donc en être indépendante.

Cette section présente une solution, intégrée à notre bibliothèque de squelettes algorithmiques.

5.6.1 Garantir la répétabilité

L'accès à des nombres pseudo-aléatoires au sein de chaque tâche peut être fait :

1. en partageant un flux de nombres pseudo-aléatoires ;
2. en créant un flux par *thread* ;
3. en créant un flux par tâche exécutée en parallèle.

La première solution requiert un mécanisme de *mutex* et causera de manière pratiquement certaine le résultat présenté dans la figure 5.25. En effet, même protégé par un *mutex* (qui, au passage, ralentit l'exécution du programme en créant une section critique, qui diminue l'accélération qu'il est possible d'obtenir), l'accès au flux de nombres pseudo-aléatoires n'est soumis à aucune restriction d'ordre. Ainsi, un *thread* peut accéder en premier au flux durant une exécution mais pas durant une autre exécution.

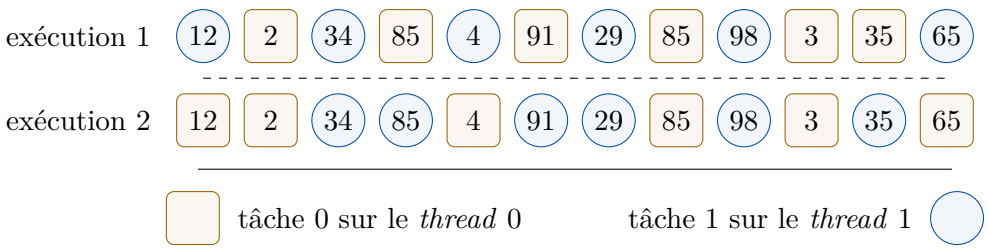


FIGURE 5.25 – Deux partages possibles par deux *threads* des nombres issus d'une séquence pseudo-aléatoire

La deuxième solution, en revanche, permet, sous certaines conditions, une répétabilité pour un degré de parallélisme défini. Ces conditions ne concernent que les tâches faisant usage de nombres pseudo-aléatoires (et par extension celles dont elles sont dépendantes, mais ce n'est pas le propos ici). Le terme de « tâche » dans le paragraphe suivant est restreint à cette définition.

D'une part, il faut que les tâches soient, d'une exécution à une autre, toujours exécutées par le même *thread* (en identifiant un *thread* par le flux de nombres pseudo-aléatoires qui lui est associé). Ceci est difficile à garantir lorsque l'on utilise certains mécanismes d'orchestration comme le *thread pool* [Schmidt 1998] où l'affectation d'une tâche à un *thread* est décidée durant l'exécution du programme en fonction de l'ordre d'arrivée des tâches et de la disponibilité des *threads*. D'autre part, il faut que les tâches exécutées par un *thread* le soient toujours dans le même ordre. Sans cela, il est évident que les nombres obtenus vont différer entre les exécutions.

De par le fonctionnement de notre bibliothèque, il est possible de conserver une affectation aux *threads* et un ordre d'exécution précis pour chaque tâche (en particulier celles utilisant des nombres pseudo-aléatoires), ce qui valide ces pré-requis lorsqu'une politique d'exécution compatible est utilisée. Cependant, comme nous visons une répétabilité indépendante du degré de parallélisme, cette solution ne convient pas, puisqu'il s'agit d'associer un flux à un *thread*. En utilisant un nombre variable de *threads*, certaines tâches qui auraient obtenu des nombres depuis un flux obtiendraient des nombres différents depuis un autre flux, comme le montre l'exemple de la figure 5.26 avec 1 et 2 *threads*.

En fournissant un flux de nombres pseudo-aléatoires à chaque tâche, dernière des trois solutions, le nombre de *threads* utilisés n'est plus pris en compte, rendant la répétabilité valable pour un nombre quelconque de *threads*. Pour la validité scientifique des résultats produits (et non seulement la répétabilité), il est en plus nécessaire de construire des flux indépendants [D. R. C. Hill et al. 2013].

Garantir la répétabilité, au niveau logiciel, dans le cadre de l'utilisation de nombres pseudo-aléatoires au sein d'un programme exécuté en parallèle peut donc être accompli en créant un nombre suffisant (c'est-à-dire au moins égal au nombre de tâches) de flux indépendants et en les distribuant de sorte à ce qu'une tâche donnée accède toujours au même flux. Le premier point est trivial dans notre cadre, puisque le nombre de tâches est connu, sans quoi une limite haute devrait être définie. La distribution des flux requiert deux mécanismes : l'identification des tâches et un moyen de transmettre des informations à celles-ci.

L'identification des tâches est déjà accomplie par la bibliothèque en interne (voir la sec-

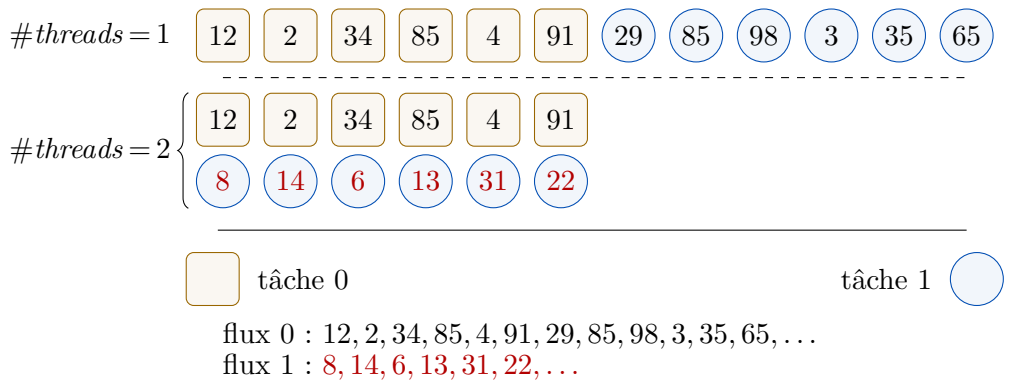


FIGURE 5.26 – Différence entre 1 et 2 *threads* ayant chacun leur séquence de nombres pseudo-aléatoires

tion 5.5.2). Cette identification garantit que deux tâches ayant un même identifiant ont une dépendance et sont donc nécessairement exécutées l'une après l'autre. Grâce à cela, nous réduisons le nombre de flux nécessaire au nombre de tâches « parallèles » plutôt qu'au nombre de tâches total comme indiqué ci-avant.

Le second mécanisme nécessaire est la transmission de données aux tâches. Notre bibliothèque propose un système de liens (voir la section 5.4.2) décrivant les flux de données et fournissant deux templates de types particuliers permettant de représenter les paramètres et les résultats des fonctions. Ainsi, `P<i>` où `i` est un entier naturel désigne le paramètre de la fonction appelante d'indice `i` et `R<i>` indique le résultat du muscle d'indice `i`. Un troisième template s'ajoute à ceux-ci et permet la substitution de données contextuelles, dépendantes de l'identifiant du muscle. Concrètement, il s'agit d'un template nommé `C` et qui fonctionne de manière similaire : `C<i>` correspond à la donnée contextuelle d'indice `i`.

Le principe de cette nouvelle catégorie est de fournir un ou plusieurs éléments d'un n-uplet de données aux tâches le requérant. Le n-uplet est instancié autant de fois qu'il y a de tâches ($a \times b$ dans l'exemple du GRASP×ELS), afin que chacune puisse avoir accès à une instance propre. Ce n-uplet peut être défini par le développeur, ce qui permet donc son utilisation pour des besoins plus étendus que le support de la répétabilité tel que nous le proposons.

Le n-uplet fourni par défaut comporte ainsi deux informations : l'identifiant de la tâche et un PRNG. Ainsi, `C<0>` permet d'accéder à l'identifiant de la tâche tandis que `C<1>` permet d'obtenir une instance propre de PRNG.

Cependant, si une valeur numérique est adaptée pour la sélection d'une donnée dans un n-uplet, et qu'elle est également compréhensible pour les deux templates `P` et `R`, elle n'est en revanche pas efficace pour dénoter ce à quoi elle correspond dans ce cas. En effet, si au sein du n-uplet par défaut, 1 correspond à l'indice du PRNG, ce n'est pas nécessairement toujours sa position s'il existe ce paramètre au sein d'un autre n-uplet possible. Dans tous les cas, il s'agit d'une notation très peu explicite, propice à l'introduction d'erreurs et nécessitant donc probablement des commentaires à chaque utilisation.

Plusieurs solutions sont possibles, par exemple en créant des valeurs nommées (`constexpr unsigned RNGId;`) pour les utiliser comme arguments du template `C`. Une seconde solution est de créer des alias pour des instances de ce template. Ces deux solutions ne s'excluent pas, et nous avons fait le choix de proposer par défaut la seconde. Ainsi, `RNG` permet de signifier qu'un PRNG est attendu comme paramètre.

L'extrait de code 5.6 (illustré par la figure 5.14), présenté comme code utilisant le système des liens lors de son introduction, correspond à la première solution : un unique PRNG partagé par toutes les tâches. Le paramètre de type PRNG provient de la fonction appelante et est transmis aux muscles. Il est d'une part nécessaire d'utiliser un *mutex* au sein des muscles, augmentant le couplage entre le code utilisateur et la parallélisation, mais d'autre part, comme nous l'avons vu, cela induit une absence de répétabilité.

À la place, en utilisant les paramètres contextuels du système de liens (dans ce cas `RNG`), on obtient l'extrait de code 5.19 (illustré par la figure 5.27). La différence principale à observer dans ce nouveau code est que seuls les muscles utilisant réellement les nombres pseudo-aléatoires ont le paramètre `RNG`. Chaque instance de ces muscles recevra automatiquement, grâce au système interne de la bibliothèque, un flux de nombres pseudo-aléatoires adapté à sa situation.

```
template<typename Problem, typename Solution>
using GraspLinks =
L<FarmSel, Solution(Problem const&),
  L<Serial, R<1>(P<0>),
    Solution(P<0>, RNG),
    Solution(P<0>, R<0>, RNG)
  >,
  Solution(Solution const&, Solution const&)
>;
```

(⩾ C++14)

Extrait de code 5.19 – Liens du squelette d'un GRASP avec un paramètre contextuel

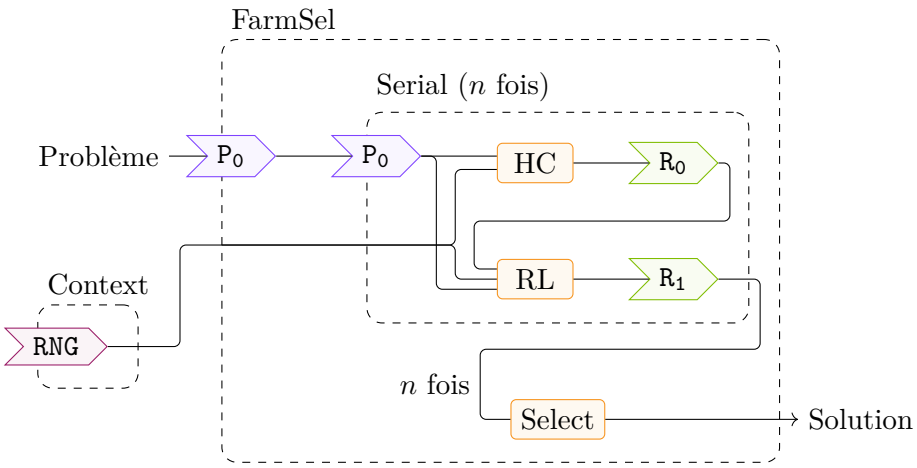


FIGURE 5.27 – Transmission de données au sein d'un GRASP en utilisant les paramètres contextuels

5.6.2 Réduire le nombre de PRNG nécessaires

5.6.2.1 Principe général

Créer un flux de nombres pseudo-aléatoires pour chaque tâche pouvant être exécutée en parallèle d'une autre est une solution fonctionnelle. Il peut s'agir, selon la politique d'exécution sélectionnée, de la seule envisageable pour garantir la répétabilité indépendamment du nombre de *threads*.

Cependant, en considérant les capacités réelles des machines sur lesquelles un programme est exécuté, il est possible de faire mieux. En effet, le nombre de cœurs des machines croît mais n'est pas infini. De plus, en pratique, un programme possède une durée de vie limitée [Tamai et Torimitsu 1992]. Ainsi, il est logique de supposer une limite maximale du nombre de cœurs auxquels aura accès une application lors de son développement. Si le programme vient à être maintenu suffisamment longtemps pour que cette limite s'avère trop basse, il reste possible de la mettre à jour, au coût d'une nouvelle compilation.

Grâce à cette hypothèse, l'utilisateur peut inférer, à partir de la limite théorique de cœurs, un nombre maximal T de *threads* que le programme pourra utiliser. C'est jusqu'à cette limite que notre bibliothèque devra assurer la répétabilité. Lorsqu'un nombre arbitrairement grand de *threads* est possible, toute tâche parallélisable doit être considérée comme exécutée en parallèle de toutes les autres. Ceci contraint à associer à chacune un PRNG propre.

En revanche, si le nombre de *threads* est limité il peut arriver que certaines tâches soient, selon la politique d'exécution, toujours exécutées par un même *thread*. Dans ce cas, plutôt que d'associer à chaque tâche parallélisable un PRNG, il s'agit de déterminer les groupes de tâches qui sont toujours exécutées ensemble afin d'associer un PRNG à chaque groupe.

Le cas général consiste alors à déterminer pour chaque quantité possible de *thread* t (typiquement dans l'intervalle $[\![1, T]\!]$) les ensembles de tâches exécutées en séquence pour finalement les combiner et en déduire les ensembles de tâches qui seront toujours exécutées par le même *thread* quelque soit le nombre de *threads*.

Afin d'expliquer le principe, considérons l'exemple suivant : 9 tâches m_0 à m_8 seront réparties sur 1 à 4 *threads*. La politique d'exécution utilisée pour cet exemple a pour objectif d'équilibrer la charge des *threads* en leur affectant autant que possible le même nombre de tâches. Ainsi, lorsque le nombre de *threads* alloués est de 2, un ensemble S_2 de deux ensembles de tâches est déduit et contient l'ensemble des tâches de m_0 à m_4 et l'ensemble des tâches de m_5 à m_8. L'équation (5.2) décrit les ensembles S_i, où i, le nombre de *threads* associés, prend ses valeurs dans l'intervalle $[\![1, 4]\!]$.

$$\begin{aligned} S_1 &= \{\{m_0, m_1, m_2, m_3, m_4, m_5, m_6, m_7, m_8\}\} \\ S_2 &= \{\{m_0, m_1, m_2, m_3, m_4\}, \{m_5, m_6, m_7, m_8\}\} \\ S_3 &= \{\{m_0, m_1, m_2\}, \{m_3, m_4, m_5\}, \{m_6, m_7, m_8\}\} \\ S_4 &= \{\{m_0, m_1, m_2\}, \{m_3, m_4\}, \{m_5, m_6\}, \{m_7, m_8\}\} \end{aligned} \tag{5.2}$$

En identifiant quelles tâches sont toujours exécutées ensemble, c'est-à-dire par un *thread* commun et donc de manière séquentielle, quelle que soit la quantité de *threads* alloués, on obtient l'ensemble S de l'équation (5.3).

$$S = \{\{m_0, m_1, m_2\}, \{m_3, m_4\}, \{m_5\}, \{m_6\}, \{m_7, m_8\}\} \tag{5.3}$$

Chacun des ensembles contenus dans cet ensemble S contient donc des tâches qui ne seront jamais exécutées en parallèle. Dans cet exemple, il est donc possible de créer en tout 5 PRNG g_0 à g_4 pour les affecter à chacun des 5 sous-ensembles. C'est-à-dire que g_0 sera utilisé par les tâches m_0, m_1 et m_2 ; g_1 par les tâches m_3 et m_4 ; g_2 par la tâche m_5 ; g_3 par la tâche m_6 ; et enfin g_4 par les deux dernières tâches m_7 et m_8. Ainsi, quelque que soit le nombre de *threads* $t \in [\![1, 4]\!]$, un même PRNG ne pourra jamais être utilisé par deux tâches qui s'exécutent en parallèle.

Une méthode efficace pour déterminer S est illustrée par la figure 5.28. Les quatre premières lignes correspondent à une représentation alternative des ensembles S_1 à S_4 (équation (5.2)) et la

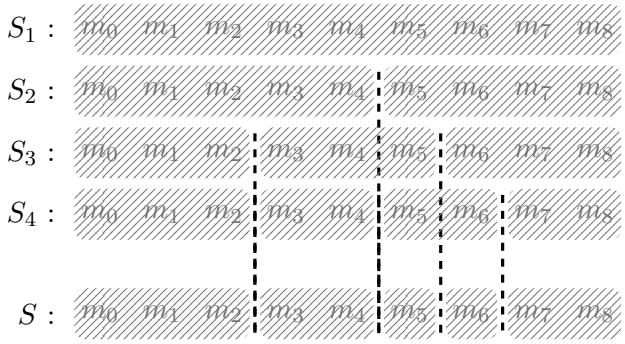


FIGURE 5.28 – Représentation de la fusion d'ensembles de tâches exécutées en séquence

dernière ligne représente le résultat de leur fusion, S (équation (5.3)). Tels que sont construits les ensembles, il semble être suffisant de cumuler les « séparations » de chacun des ensembles S_1 à S_4 pour obtenir S. En construisant les ensembles de ces séparations, leur union décrit alors l'ensemble des séparations de S. Dans cet exemple, ces ensembles de séparations, de S'_1 à S'_4, ressembleraient à $S'_1 = \emptyset$, $S'_2 = \{(m_3, m_4)\}$, $S'_3 = \{(m_1, m_2), (m_4, m_5)\}$ et $S'_4 = \{(m_1, m_2), (m_3, m_4), (m_5, m_6)\}$ (un couple (m_i, m_{i+1}) indiquant donc qu'une séparation existe entre les tâches i et $i+1$) et leur union donne $S' = \{(m_1, m_2), (m_3, m_4), (m_4, m_5), (m_5, m_6)\}$.

5.6.2.2 Démonstration

Afin de démontrer ceci, notons N un entier non nul représentant le nombre de tâches. Notons alors $I = [\![0, N[\![$ et t le nombre de *threads* qu'il est possible d'utiliser.

Définition : on appellera subdivision de I en t parties tout $(t+1)$-uplet $s = (c_0, \dots, c_t)$ tel que

$$c_0 = 0 < c_1 < \cdots < c_t = N.$$

Remarque : en notant $I_k = [\![a_k, a_{k+1}[\![$, on observe que la famille $(I_k)_{0 \leqslant k \leqslant t-1}$ est une partition de I, c'est-à-dire

$$\bigcup_{k=0}^{t-1} I_k = I$$
$$\forall i, j \in [\![0, t[\![, i \neq j \implies I_i \cap I_j = \emptyset.$$

Remarque : on utilisera néanmoins le terme de subdivision et non celui de partition pour exprimer le fait que les I_k sont des intervalles. Notons Σ l'ensemble des subdivisions de I en un nombre quelconque de parties, e l'application de Σ dans $P(I)$ (l'ensemble des parties de I) qui à $s = (a_0, \dots, a_t) \in \Sigma$ associe $e(s) = \{a_0, \dots, a_t\} \in P(I)$. Cette application, qui est clairement bijective, permet d'induire sur Σ une relation d'ordre partielle définie par :

$$\forall (s, s') \in \Sigma^2, s \leqslant s' \iff e(s) \subset e(s').$$

Remarque : les autres relations de comparaison ($<$, $\geqslant$, $>$) se déduisent naturellement de la définition précédente (par exemple $\forall (s, s') \in \Sigma^2, s < s' \iff e(s) \not\subseteq e(s')$).

Remarque : si $s_1 > s_0$, on dit que s_1 est plus fine que s_0.

Définition : on dira que m est un majorant de $S \subset \Sigma$, un ensemble de subdivisions d'un intervalle entier, si et seulement si $\forall s \in S, s \leqslant m$.

Exemple : en prenant l'intervalle entier $I = [\![0, 6]\!]$ et $S = \{(0, 3, 4, 6), (0, 2, 3, 6)\}$, les majorants de S sont $(0, 1, 2, 3, 4, 5, 6)$, $(0, 2, 3, 4, 5, 6)$, $(0, 1, 2, 3, 4, 6)$ et $(0, 2, 3, 4, 6)$.

Définition : on appellera borne supérieure d'un ensemble de subdivisions le plus petit de ses majorants.

Exemple : en poursuivant avec l'exemple précédent, $(0, 2, 3, 4, 6)$ est la borne supérieure de S.

Définissons maintenant dans Σ une loi de composition interne $\vee$ définie par

$$\forall (s, s') \in \Sigma^2, s \vee s' = e^{-1}(e(s) \cup e(s')).$$

Remarque : en termes simples, $s \vee s'$ s'obtient en ordonnant la liste réunissant les termes de s et s'.

Exemple : ainsi, $(0, 2, 6) \vee (0, 2, 3, 4) = (0, 2, 3, 4, 6)$.

Lemme : $\vee$ est associative.

Démonstration : l'associativité de $\vee$ découle de l'associativité de l'union et de la bijectivité de e.

Théorème : soit $S = \{s_1, \ldots, s_n\} \subset \Sigma$ un ensemble de subdivisions d'un intervalle entier. La borne supérieure de S est $b = \bigvee_{i=1}^{n} s_i$.

Démonstration : par définition, on a $e(b) = \bigcup_{i=1}^{n} e(s_i)$. Donc, $\forall j \in [\![1, n]\!], e(s_j) \subset e(b)$. Il en découle que $\forall j \in [\![1, n]\!], s_j \leqslant b$, et donc que b est un majorant de S.

Supposons alors qu'il existe $b' \in \Sigma$, tel que b' soit un majorant de S, avec $b' < b$. Pour tout $s_i \in S$, notons plus précisément $s_i = (c_{i,0}, \ldots, c_{i,t_i})$ et $b' = (c'_0, \ldots, c'_{p'})$. Puisque $b' < b$, on a $e(b') \nsubseteq e(b)$, donc il existe $s_k \in S$ et $c_{k,l} \in e(s_k)$ tel que $c_{k,l} \in e(b)$ et $c_{k,l} \notin e(b')$.

Par conséquent, $\forall j \in [\![1, p']\!]$, $c'_j \neq c_{k,l}$, et on ne peut donc pas avoir $s_k \leqslant b'$, ce qui est une contradiction avec le fait que b' soit un majorant de S.

L'hypothèse $b' < b$ est donc absurde et, en conclusion, b est le plus petit majorant de S, c'est-à-dire sa borne supérieure.

5.6.2.3 Conclusion

La conclusion de cette démonstration est qu'une subdivision en sous-ensembles de tâches qui, quel que soit le nombre de *threads* utilisés, seront exécutées par un même *thread*, peut être trouvée en appliquant et en généralisant la méthode algorithmique expliquée précédemment et schématisée dans la figure 5.28. Si la subdivision trouvée, notée s, comporte p parties, alors il faudra associer à chaque identifiant de tâche id un identifiant de contexte $idx \in [\![0, p-1]\!]$, toutes les tâches ayant le même identifiant de contexte idx étant exécutées par un même *thread*.

La conversion d'un identifiant id d'une tâche en un identifiant de contexte idx est une opération qui nécessite de trouver le premier élément de s supérieur à id, idx étant alors égal à sa position dans s moins un. Par exemple, si $s = (0, 2, 4, 5, 6, 9)$ (cela correspond à l'équation (5.3)), le tableau 5.1 liste les conversions d'indice. En terme de code, s peut être conservé sous la forme d'un `std::set` (un ensemble ordonné) et la conversion peut être effectuée par la fonction définie dans l'extrait de code 5.20 (complexité globale linéaire en la taille de s).

Table 5.1 – Conversions d'identifiants de tâche en identifiants de contexte

identifiant de tâche	borne supérieure	position	identifiant de contexte
0	2	1	0
1	2	1	0
2	2	1	0
3	4	2	1
4	4	2	1
5	5	3	2
6	6	4	3
7	9	5	4
8	9	5	4

```cpp
std::size_t contextIdFromTaskId(std::size_t id) {
  // s: std::set<std::size_t>, une subdivision
  auto it = std::upper_bound(std::begin(s), std::end(s), id);
  return std::distance(std::begin(s), it) - 1;
}
```

(⩾ C++11)

Extrait de code 5.20 – Conversion d'un identifiant de tâche en identifiant de contexte

Tout exécuteur, devant fournir cette fonction, doit construire la subdivision s correspondant à ce qu'il devra exécuter et aux contraintes données par l'utilisateur quant aux nombres de *threads* devant être supportés pour la répétabilité. Cette subdivision découle de $\bigvee_{t\in\mathbf{T}} s_t$ (notation introduite dans la section 5.6.2.2), où $\mathbf{T}$ est l'ensemble des nombres de *threads* possibles et les s_t sont les subdivisions obtenues pour chaque valeur $t \in \mathbf{T}$. En C++, l'opération $\vee$ est implémentée par une simple union des `std::set`. En pratique, le résultat de l'union est automatiquement obtenu en insérant au fur et à mesure les éléments de la subdivision : le fonctionnement de `std::set` (qui représente la notion d'ensemble) fait que si l'élément est déjà présent, il n'est pas ajouté à nouveau.

5.6.3 Évaluation du nombre de PRNG créés

Grâce à la technique expliquée précédemment, le nombre de PRNG en particulier, mais plus généralement le nombre de n-uplets pour le contexte, qu'il est nécessaire d'instancier est réduit. La figure 5.29 montre les quantités nécessaires sans et avec optimisation selon différentes politiques d'exécution pour une valeur de $T = 64$ et pour N variant entre 1 et 1000. Les deux politiques d'exécution utilisées ont le même objectif (parallélisation multi-niveau du squelette algorithmique) mais diffèrent sur leur manière de répartir les tâches sur les *threads* à leur disposition, l'une correspondant globalement à la méthode décrite dans la section 5.5.3, l'autre optant pour une alternative « gloutonne » donnant théoriquement des résultats similaires lorsque la durée des tâches est elle aussi similaire. On observe que le choix d'une politique d'exécution peut donc également être motivé par le nombre de PRNG que celle-ci implique.

L'ensemble des nombres de *threads* pour lesquels il faut garantir la répétabilité est de définition libre : l'utilisateur peut indiquer par exemple spécifiquement $\{3, 111\}$. En pratique, l'ensemble des valeurs de 1 jusqu'à une borne supérieure semble assez commun. Cependant, un autre motif est fréquent : l'ensemble des puissances de 2 jusqu'à une limite. En utilisant de tels ensembles, le nombre d'instances du n-uplet contextuel décroît encore. La figure 5.30 montre les quantités nécessaires de ce n-uplet sans et avec optimisation pour une même politique d'exécution pour une valeur de $N \in [\![1, 1000]\!]$ et $T = 64$. Dans un des deux cas, $\mathbf{T} = [\![1, 64]\!]$, tandis que dans l'autre cas, $\mathbf{T} = 2^{[\![0,6]\!]}$. Avec la seconde définition de $\mathbf{T}$, on observe que le nombre de PRNG semble posséder une valeur maximale indépendante de N.

Il est encore possible d'améliorer ces résultats. Notamment, lorsque les différents paramètres spécifiques au contexte (`C<i>`) ne sont pas utilisés du tout, il semble évident qu'aucune instance du n-uplet contextuel n'est nécessaire. Or, grâce au système de liens, dont l'information est connue durant la compilation, la détection des conditions de ce critère est justement permise. Plus généralement, il est possible d'analyser l'arbre du squelette pour ignorer dans le calcul toute branche dont aucune feuille (les muscles) ne fait usage d'un paramètre contextuel.

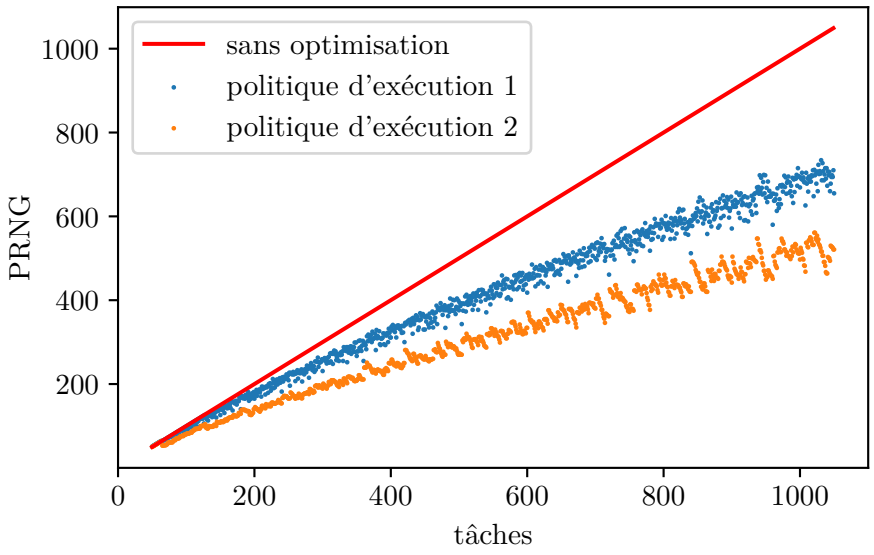


FIGURE 5.29 – Nombre de PRNG pour deux politiques d'exécution avec 1 à 1000 tâches et pour un nombre de *threads* dans $[\![1, 64]\!]$

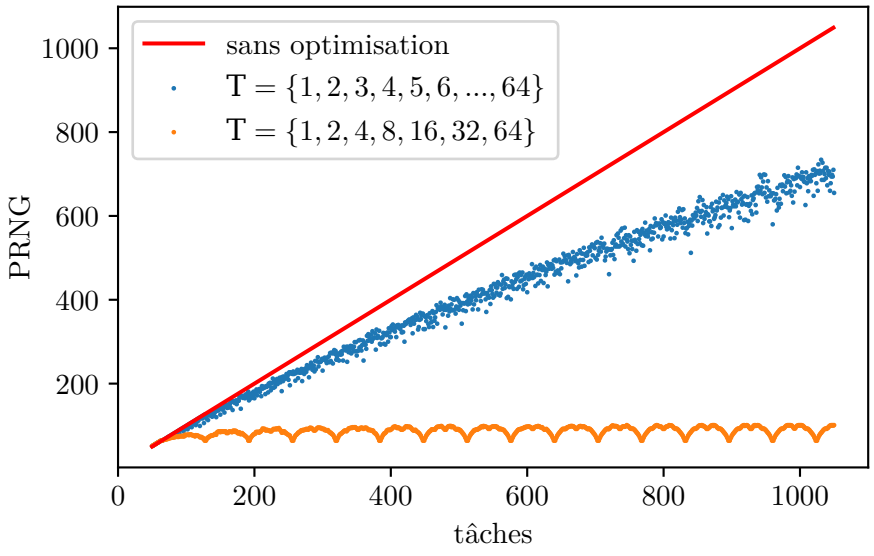


FIGURE 5.30 – Nombre de PRNG pour une politique d'exécution avec 1 à 1000 tâches et pour un nombre de *threads* dans $[\![1, 64]\!]$ et $2^{[\![0,6]\!]}$

5.7 Utilisation

5.7.1 Langage dédié

L'interface proposée par notre bibliothèque, telle que présentée jusqu'ici, requiert la définition dans un premier temps, d'une part, de la structure du squelette, et d'autre part des liens qui seront ensuite combinés à la structure pour obtenir un squelette. Dans un second temps, les muscles sont ajoutés pour former un corps qui pourra être implémenté selon une politique d'exécution et enfin exécuté. Cette manière de faire offre l'avantage de séparer naturellement la structure et les liens, de sorte que leur réutilisation au sein d'autres squelettes est directe.

Cette section introduit un EDSL, proposé par la bibliothèque, comme alternative à cette première interface. Le principe général de ce langage dédié est de construire les corps à partir des muscles, la structure du squelette étant déterminée par la manière de combiner ces muscles au moyen de fonctions dédiées.

Sans cet EDSL, pour construire un GRASP il faut définir sa structure (voir l'extrait de code 5.3) et ses liens (voir l'extrait de code 5.6) pour les combiner (voir l'extrait de code 5.7) et enfin associer à ce squelette des muscles (voir l'extrait de code 5.10) pour former un corps. En utilisant cet EDSL, toutes ces étapes se traduisent par ce qui est fait dans l'extrait de code 5.21 Les types `Init`, `LS` et `Select` correspondent à des muscles.

```
auto init   = link<Solution(P<0>, P<1>), Init>();
auto ls     = link<Solution(R<0>, P<1>), LS>();
auto select = link<Solution(Solution, Solution), Select>();

auto grasp = link<Solution(Problem const&, PRNG&)>(
  (*link<R<1>(P<0>, P<1>)>(init, ls)) ->* select
);
```

(⩾ C++14)

Extrait de code 5.21 – Définition d'un GRASP utilisant un EDSL

Cette section a pour objectif d'expliquer les différents éléments de cet EDSL (le template `link` ou encore les opérateurs utilisés, par exemple). Au sein de ce langage, les opérandes (dans le code : `init`, `ls` et `select`) représenteront les muscles qui seront utilisés (respectivement `Init`, `LS` et `Select`). Puisqu'il ne s'agit pas directement de ces muscles mais de variables les représentant au sein du langage, le terme de pseudo-muscle sera employé pour les désigner. Ces opérandes seront utilisés dans des expressions qui permettront de : les annoter afin de définir les liens (ce à quoi sert le template `link`) ; les combiner pour obtenir la représentation du futur squelette ; ... De la même manière que pour le cas des pseudo-muscles, le terme de pseudo-corps correspondra à la représentation d'un corps dans l'EDSL. Plus généralement, le préfixe « pseudo- » sera ajouté, dans cette section, devant un terme pour indiquer qu'il s'agit d'une représentation au sein du langage dédié. Dans l'extrait de code 5.21, on définit donc précisément un pseudo-corps de GRASP.

Les opérandes terminaux (pseudo-muscles) sont créés à partir d'un muscle. L'extrait de code 5.22 présente la syntaxe la plus simple créant un opérande à partir du muscle `LocalSearch`.

```
auto localSearch = makeOperand<LocalSearch>();
```

(⩾ C++14)

Extrait de code 5.22 – Définition d'un opérande à partir d'un muscle

Dès lors que des pseudo-muscles sont définis, ils peuvent être utilisés pour créer des pseudo-corps. À noter que ce qui est généré est bien directement un pseudo-corps, l'accès au squelette (précisément à la structure et aux liens) se fait au travers de ce pseudo-corps, sujet abordé ultérieurement.

Pour chaque os est fournie une fonction spécifique et éventuellement un ensemble d'opérateurs propres pour la construction de pseudo-corps à partir d'opérandes du langage (c'est-à-dire soit un pseudo-muscle, soit un pseudo-corps). Par exemple, l'os `Serial` fournit la fonction `serial` et l'opérateur équivalent `operator,` tandis que l'os `Farm` met à disposition la fonction `farm` et l'opérateur unaire équivalent `operator*`. L'extrait de code 5.23 présente la définition de pseudo-corps à un seul os pour quelques structures.

```
auto taskSeq    = (task0, task1); // or serial(task0, task1);
auto taskFarm   = *task;          // or farm(task);
auto taskWhile  = while_(cond).do_(task);
```

(⩾ C++14)

Extrait de code 5.23 – Définition de pseudo-corps à un seul os pour différentes structures

Certains os n'étant capables de produire que des comportements séquentiels, ils peuvent être construits à partir de leur homologue parallélisable au moyen d'une fonction indiquant qu'il s'agit de la version séquentielle qui doit être produite. Ainsi, l'os correspondant à n exécutions en séquence d'une tâche (`Loop`) s'obtient, comme montré dans l'extrait de code 5.24, à partir d'un pseudo-corps (utilisant l'os `Farm`), lui-même créé de la même manière qu'à la ligne 2 de l'extrait de code 5.23.

```
auto taskLoop = seq(*task); // or loop(task)
```

(⩾ C++14)

Extrait de code 5.24 – Définition d'un pseudo-corps utilisant l'os `Loop`

Lors de la création de pseudo-corps, il est possible d'associer des liens à un opérande (à nouveau, pseudo-muscle ou pseudo-corps). Pour cela, plusieurs outils sont fournis : un pour associer les liens dès la création d'un pseudo-muscle et un autre pour associer les liens *a posteriori* qui peut donc s'appliquer sur un pseudo-corps quelconque. Les signatures qui sont utilisées comme arguments fonctionnent à l'identique de ce qui est expliqué dans la section 5.4.2. Par exemple, pour définir, dans ce langage, les mêmes liens pour la tâche de recherche locale à la ligne 6 de l'extrait de code 5.6, il est possible d'utiliser les syntaxes présentées par les extraits de code 5.25 et 5.26.

```
auto direct   = makeOperand<Solution(P<0>, P<1>), LocalSearch>();
auto alt      = link<Solution(P<0>, P<1>), LocalSearch>();
```

(⩾ C++14)

Extrait de code 5.25 – Affectation directe de liens à un pseudo-corps

```
auto localSearch = makeOperand<LocalSearch>();
auto indirect = link<Solution(P<0>, P<1>)>(localSearch);
```

(⩾ C++14)

Extrait de code 5.26 – Affectation indirecte de liens à un pseudo-corps

L'affectation directe, soit par la syntaxe de construction d'opérande (comme pour `direct`),

soit par la syntaxe alternative (comme pour `alt`), peut être utilisée lorsque l'on sait à l'avance comment le pseudo-corps produit sera utilisé.

La syntaxe indirecte permet de définir localement la signature à employer pour un pseudo-corps existant (et ayant éventuellement déjà une signature associée) : dans l'extrait de code 5.26, on affecte à `localSearch` de nouveaux liens (la variable `indirect` conservant le resultat de cette opération).

Certaines circonstances permettent la déduction d'une partie de la signature d'un pseudo-corps. C'est le cas lorsque l'on produit celui d'un `FarmSel` comme dans l'extrait de code 5.27. Ce mécanisme permet de réduire le nombre de cas où l'application explicite de liens est nécessaire.

```
auto select = link<Solution(Solution const&, Solution const&), Select>();
auto farmSel = link<Auto(Problem const&)>(*task) ->* select;
```

(⩾ C++14)

Extrait de code 5.27 – Définition d'un pseudo-corps utilisant l'os `FarmSel`

Pour la création de `farmSel`, un premier pseudo-corps est construit à partir du pseudo-muscle `task`, en utilisant l'`operator*`, et correspond donc à une `Farm` (on aurait d'ailleurs pu écrire `farm(task)` au lieu de `*task`). À ce pseudo-corps est associé la signature `Auto(Problem const&)` : un paramètre doit être attendu (de type `Problem const&`) et le type de retour (`Auto`) sera automatiquement remplacé.

En effet, ce pseudo-corps de `Farm` est ensuite utilisé pour construire un pseudo-corps de `FarmSel` par l'utilisation de l'`operator->*`. Or le squelette `FarmSel` contraint le type de retour : le type de retour global est le même que celui de l'opération de sélection (ici, `select`). Cela permet l'inférence de ce type de retour dans le cas des pseudo-corps, et donc l'utilisation de ce type `Auto`. Une syntaxe alternative à l'`operator->*` est permise par une fonction membre `select` acceptant en paramètre le pseudo-corps de sélection. Dans les deux cas, le pseudo-corps produit est celui d'un `FarmSel` dont la signature est par défaut telle que son type de retour soit celui du pseudo-corps de sélection et ses paramètres ceux du pseudo-corps de `Farm`. Ainsi, si les signatures sont respectivement `FarmRet(FarmArgs...)` et `SelectRet(SelectArgs...)`, la signature résultante sera `SelectRet(FarmArgs...)`. Dans le cas ci-avant, la signature déduite est donc `Solution(Problem const&)`.

Pour construire l'expression d'un GRASP×ELS, c'est-à-dire le pseudo-corps correspondant au corps (pour rappel, squelette et muscles) de l'extrait de code 5.10, nous utilisons donc les différents outils présentés jusqu'ici (voir l'extrait de code 5.28). Bien qu'il soit possible de le faire en une seule expression, notamment pour des raisons de lisibilité, nous définissons dans un premier temps l'expression d'un ELS pour l'utiliser ensuite au sein d'un GRASP.

Le pseudo-corps `els` est écrit dans un premier temps pour fonctionner tel quel : les liens établis (`R<1>(Solution)`) permettent son utilisation en l'état. Cependant, il est ensuite utilisé comme pseudo-muscle de `graspels`, contexte nécessitant un changement de liens (`R<1>(R<0>)`) puisque la solution à améliorer provient alors du muscle précédent. Dans ce cas précis, il n'est donc pas utile de spécifier les liens initiaux de `els`, cependant le faire permet plus de flexibilité quant à son utilisation en général.

La fonction `box` permet d'empêcher certaines optimisations automatiques prématurées de l'arbre construit (ici la suppression du `Serial` correspondant à l'ELS). Dans cet exemple, cela permet d'éviter de devoir faire soi-même l'analyse des liens corrects à employer. Notamment, le retour `R<1>` devrait être `R<2>` avec l'aplatissement de l'arbre. Cette optimisation est ainsi

```
// all pseudo-muscles are already defined and correctly linked
auto els = link<R<1>(Solution)>(
  initLocalSearch,
  link<Solution(R<0>)>(
    iter(link<Solution(Solution)>(
      farm(link<R<1>(P<0>)>(mutate, ls)).select(innerSelect)
    )).select(outerSelect)
  )
);

auto graspels = link<Solution(Problem)>(
  farm(link<R<1>(P<0>)>(
    constructiveHeuristic,
    box(link<R<1>(R<0>)>(els))
  ).select(select)
);
```

(⩾ C++14)

Extrait de code 5.28 – Définition d'un pseudo-corps de GRASP×ELS

reportée, ce qui permet au système de procéder lui-même à la correction des liens puisqu'il possèdera alors une représentation complète et non seulement locale.

Bien que l'on puisse entièrement se reposer sur ce seul EDSL pour utiliser notre bibliothèque, la conception de squelettes algorithmiques réutilisables (typiquement pour les mettre à disposition d'autres développeurs) requiert de définir des types ou des templates plutôt que des variables. C'est-à-dire qu'il faut pour cela utiliser des squelettes et non des pseudo-corps, composés de structures et de liens, tels que présentés initialement. Pour extraire le squelette d'un pseudo-corps, il faut l'assembler à partir de la structure et des liens qui sont produits par ce pseudo-corps. À cet effet, une fonction utilitaire est fournie (voir l'extrait de code 5.29). Si l'on dispose déjà d'un pseudo-corps comme celui de l'extrait de code 5.28, cette fonction peut être employée ainsi : `decltype(getSkeleton(graspels))`.

```
template<typename Expression>
constexpr auto getSkeleton(Expression)
-> BuildSkeletonT<typename Expression::Struct, typename Expression::Links>;
```

(⩾ C++11)

Extrait de code 5.29 – Fonction `getSkeleton` permettant de convertir un pseudo-corps en squelette

Par TAD (*Template Argument Deduction*) (voir la section 2.5), le type `Expression` est déduit lors de l'appel à partir de l'argument qui est passé à la fonction. Si ce type possède effectivement les types membres `Struct` et `Links` (c'est le cas des pseudo-corps), le type de retour de la fonction est déterminé comme étant le squelette construit à partir de ces deux éléments. La fonction n'est jamais définie : son utilisation doit être faite dans un contexte non évalué, c'est-à-dire que l'appel ne sera utilisé par le compilateur que pour déterminer le type de retour de la fonction. Un exemple de tel contexte est l'argument d'un `decltype` (voir la section 2.11).

Les squelettes pour le GRASP et l'ELS sont respectivement implémentés par l'extrait de code 5.30 et l'extrait de code 5.31. Pour chaque cas, cela inclut donc la définition de la structure, des liens et la construction du squelette les utilisant (pour le GRASP, cela correspond, en n'utilisant pas l'EDSL, respectivement à l'extrait de code 5.3, l'extrait de code 5.6 et l'extrait de code 5.7).

```cpp
template<
  typename Problem, typename Solution, typename PRNG,
  typename Init, typename LS, typename Select
>
using SkelGrasp = decltype(getSkeleton([]{
  constexpr auto init   = link<Solution(P<0>, P<1>), Init>();
  constexpr auto ls     = link<Solution(R<0>, P<1>), LS>();
  constexpr auto select = link<Solution(Solution, Solution), Select>();

  return link<Solution(Problem const&, PRNG&)>(
    (*link<R<1>(P<0>, P<1>)>(init, ls)) ->* select
  );
}()));
```

(⩾ C++20)

Extrait de code 5.30 – Squelette d'un GRASP à partir d'un pseudo-corps

```cpp
template<
  typename Solution, typename PRNG,
  typename InitLS, typename Mutate, typename LS,
  typename InnerSelect, typename OuterSelect
>
using SkelEls = decltype(getSkeleton([]{
  constexpr auto initLS       = link<Solution(P<0>, P<1>), InitLS>();
  constexpr auto mutate       = link<Solution(P<0>, P<1>), Mutate>();
  constexpr auto ls           = link<Solution(R<0>), LS>();
  constexpr auto innerSelect  = link<Solution(Solution, Solution), InnerSelect>();
  constexpr auto outerSelect  = link<Solution(Solution, Solution), OuterSelect>();

  return link<R<1>(Solution, PRNG&)>((
    initLS,
    link<Solution(R<0>, P<1>)>(
      seq(*link<Solution(Solution, P<1>)>(
        *link<R<1>(P<0>, P<1>)>(mutate, ls) ->* innerSelect
      )) ->* outerSelect
    )
  ));
}()));
```

(⩾ C++20)

Extrait de code 5.31 – Squelette d'un ELS à partir d'un pseudo-corps

Cette syntaxe permet une plus grande expressivité et une définition plus dense d'un squelette par rapport aux relativement nombreuses étapes nécessaires lorsque l'on passe par la structure et les liens, puis enfin le squelette. L'utilisation d'une lambda permet de créer des éléments intermédiaires et donc de rendre plus lisible l'expression du pseudo-corps, ainsi que d'alléger l'écriture en s'autorisant l'utilisation de `using namespace` pour les espaces de noms de la bibliothèque en en limitant efficacement la portée lexicale. Les fonctions lambda ne peuvent être utilisées dans un contexte non évalué que depuis le standard C++20. Sans celui-ci il est possible, bien que cela nécessite davantage de code, d'utiliser à la place une fonction nommée.

5.7.2 Implémentation de l'algorithme représenté par un corps

L'implémentation de l'algorithme que représente un squelette instancié (un corps) est la pénultième étape avant son exécution. Celle-ci consiste en la création d'un fonctionoïde dont

l'interface principale est l'`operator()` acceptant comme arguments ceux décrits par les liens de l'os le plus externe du squelette concerné. Cette opération se déroule donc en partie durant la compilation.

Pour obtenir une implémentation de l'algorithme, il faut fournir au moins le squelette instancié y correspondant et la méthode d'exécution souhaitée (séquentielle, parallèle, ...) (voir la section 5.5). L'extrait de code 5.32 montre comment l'implémentation est obtenue. Le premier argument de la fonction `implement`, dans ce cas `DynamicPool`, correspond au template d'une politique d'exécution que l'on veut utiliser : on aurait pu mettre `Sequential`, `StaticPool`, ...

```
auto graspEls = implement<DynamicPool, GRASPxELS>();
```

(⩾ C++14)

Extrait de code 5.32 – Implémentation de GRASP×ELS à partir d'un corps

La variable `graspEls` correspond alors à un fonctionoïde et l'appeler exécutera l'algorithme décrit par le squelette algorithmique correspondant. Cependant, en l'état, il manque les informations purement dynamiques telles que le nombre d'itérations que doivent effectuer les os comme `FarmSel` ou encore le nombre de *threads* qui peuvent être utilisés.

Ces données peuvent être renseignées comme cela est fait dans l'extrait de code 5.33. La ligne 1 configure le nombre d'itérations de la boucle du GRASP. Les paramètres du squelette sont tous contenus dans le membre `skeleton`, et les noms sont définis par les os eux-mêmes.

```
1  graspEls.skeleton.n = 10;
2  graspEls.skeleton.task.task<1>().task<1>().n = 15;
3  graspEls.skeleton.task.task<1>().task<1>().task.n = 20;
4
5  graspEls.executor.cores = 16;
6  graspEls.executor.repeatability.expUpTo(64);
```

(⩾ C++14)

Extrait de code 5.33 – Configuration du GRASP×ELS instancié (extrait de code 5.32)

La ligne 2 est un peu plus difficile à lire. Le premier os étant un `FarmSel`, il possède plusieurs membres, notamment `n` que l'on a déjà vu, mais aussi `task` qui réfère au muscle exécuté dans la boucle. Celui-ci étant, dans ce cas, un os `Serial`, il nous permet d'accéder aux différentes tâches qu'il exécute : la deuxième (d'indice `1`) correspond ici au cœur de cet algorithme, la recherche locale implémentée par un ELS. Cet ELS étant implémenté par une séquence de deux tâches, la seconde correspondant à un `IterSel`, cette ligne en modifie donc le nombre d'itérations.

La suivante, la ligne 3, suit le même principe pour atteindre le `FarmSel` au sein de l'ELS afin d'en définir son nombre d'itérations.

La ligne 5 ne concerne pas le squelette mais l'exécuteur. Elle indique le nombre de cœurs disponibles (par défaut, la bibliothèque essaie de détecter automatiquement le nombre de cœurs de la machine et utilise cette valeur).

Enfin, la ligne 6 permet de définir un ensemble de valeurs pour lesquelles il faut garantir la répétabilité. Le membre `repeatability` comporte un ensemble de valeurs qui peut être rempli directement ou au moyen de fonctions qui permettent de le remplir en suivant un certain motif. La fonction `upTo` insère toute valeur comprise entre deux bornes en avançant par incréments successifs d'un certain pas (par défaut, de 1 à un maximum par pas de 1). La fonction utilisée dans l'exemple, `expUpTo`, agit semblablement mais utilise son pas de manière multiplicative. Dans

cet exemple, le pas par défaut étant de 2 et la valeur initiale 1, l'ensemble des nombres insérés sera $\{1, 2, 4, 8, 16, 32, 64\}$. Il est important de ne pas faire dépendre cet ensemble de la variable `graspEls.executor.cores` : celle-ci est possiblement amenée à changer d'une exécution à une autre, la répétabilité doit alors être conservée.

En utilisant le langage dédié introduit précédemment, les paramètres propres au squelette peuvent être indiqués en amont de l'implémentation. Dans l'extrait de code 5.34, l'extrait de code 5.28 est repris en ajoutant les valeurs utilisées [5] pour configurer le GRASP×ELS dans l'extrait de code 5.33. Il est ensuite implémenté : la syntaxe varie légèrement puisque le langage dédié produit une variable et non un type. La configuration du nombre de cœurs et de la répétabilité se font alors exactement comme dans le premier cas.

```
// all pseudo-muscles are already defined and correctly linked
auto elsBody = link<R<1>(Solution)>(
  initLocalSearch,
  link<Solution(R<0>)>(
    seq(15 * link<Solution(Solution)>(
      (20 * link<R<1>(P<0>)>(mutate, ls)) ->* innerSelect
    )) ->* (outerSelect)
  )
);

auto graspElsBody = link<Solution(Problem)>(
  (10 * link<R<1>(P<0>)>(
    constructiveHeuristic,
    box(link<R<1>(R<0>)>(elsBody))
  )) ->* select
);

auto graspEls = implement<DynamicPool>(graspElsBody);
```

(⩾ C++14)

Extrait de code 5.34 – Utilisation de l'EDSL pour configurer le GRASP×ELS

À partir de là, `graspEls` peut être appelé comme le montre l'extrait de code 5.35. Il n'y a rien de particulier à relever ici.

```
tsp::Solution solution = graspEls(problem);
```

(⩾ C++14)

Extrait de code 5.35 – Appel du fonctionoïde `graspEls` généré par un squelette algorithmique

5.8 Performances

Cette section présente des mesures de temps d'exécution de programmes utilisant la bibliothèque présentée dans ce chapitre. Toutes les mesures ont été effectuées sur une machine dotée d'un Intel Xeon E7-8890 v3, cadencé à 2,5 GHz et ayant 72 cœurs physiques (18 processeurs ayant chacun 4 cœurs physiques). Les programmes sont compilés en utilisant GCC (`g++`) 8.2.0 avec notamment le pack d'optimisations `O2`. Les valeurs données dans ce document sont obtenues par une moyenne sur 20 exécutions en utilisant des états initiaux différents pour les PRNG (les mêmes 20 états sont utilisés à chaque fois). Lorsque cela est pertinent (notamment, suffisamment

5. Remarque : les deux syntaxes de l'EDSL permettent de le faire, que ce soit `n * task` ou `farm(task, n)`.

visible), l'intervalle de confiance à 99 % est affiché. Pour ce qui est des exécutions, nous avons réglé l'affinité du processus de sorte que pour chaque processeur, au maximum un cœur soit utilisé. Cela limite donc à 18 cœurs pour protéger d'un biais de mesure potentiel dû à l'utilisation simultanée de plusieurs cœurs d'un même processeur.

Un objectif important que devait atteindre cette bibliothèque est d'être compétitive, en termes de temps d'exécution par rapport à une solution écrite directement par un développeur. En effet, une abstraction est moins intéressante si elle implique un fort surcoût. Pour cette raison, nous avons utilisé le langage C++ et la métaprogrammation template qui permettent, lorsqu'ils sont correctement utilisés, de s'approcher d'une implémentation manuelle en ce qui concerne le temps d'exécution.

Pour évaluer cela, nous avons comparé les temps d'exécution de programmes écrits en utilisant la bibliothèque (et en utilisant différentes politiques d'exécution) par rapport aux temps d'exécution de programmes équivalents écrits sans utiliser cette bibliothèque. Ces programmes implémentent un GRASP×ELS dans le but de résoudre des instances de TSP de 38 sommets et de 194 sommets, ils utilisent donc des nombres pseudo-aléatoires. Afin de s'assurer de la pertinence des temps mesurés, nous avons donc fait en sorte de garantir la répétabilité des programmes écrits sans utiliser la bibliothèque. Pour les versions l'utilisant, la répétabilité est obtenue sans effort.

Dans cette section, nous utiliserons trois variables N, O et I qui correspondront respectivement au nombre d'itérations (parallélisables) du GRASP, au nombre d'itérations (non parallélisables) de la boucle extérieure de l'ELS et au nombre d'itérations (parallélisables) de la boucle intérieure de l'ELS.

Toutes les mesures sont effectuées pour quatre politiques d'exécutions différentes lorsque la bibliothèque est utilisée :

— « firstlevel », qui ne parallélise que le premier niveau pouvant être exécuté en parallèle ;
— « staticpool », qui utilise un *thread pool* en associant à une tâche un *thread* en fonction de son identifiant et en équilibrant les multiples niveaux parallèles comme décrit dans la section 5.5 ;
— « dynamicpool », qui utilise un *thread pool* utilisé de manière classique ;
— « thread », qui applique l'équilibrage des multiples niveaux parallèles de la section 5.5 en créant des *threads* dynamiquement.

Bien que l'objectif principal de cette bibliothèque soit d'aider à l'écriture de programmes parallèles, elle peut également être utilisée pour produire des programmes séquentiels en choisissant une politique d'exécution appropriée. Cela peut par exemple être utile à des fins de débugage, mais aussi pour effectuer des mesures de performance dans certains domaines scientifiques où les comparaisons sont généralement effectuées sur des programmes séquentiels. Nous avons donc dans un premier temps mesuré les temps d'exécution de programmes séquentiels.

Les figures 5.31 et 5.32 montrent les temps d'exécution de programmes séquentiels avec $N = 24$, $O = 20$ et $I = 20$. Les étiquettes préfixées de « hw » correspondent aux programmes écrits sans utiliser la bibliothèque tandis que celles préfixées de « sk » indiquent ceux l'utilisant. L'étiquette « hw_seq » est associée au programme écrit pour une exécution séquentielle. Quant à « hw_par », il s'agit d'un programme écrit pour une exécution parallèle, bien que l'exécution sera effectivement séquentielle puisqu'un seul cœur est affecté.

Quelle que soit la politique d'exécutions utilisée pour les versions employant les squelettes algorithmiques, l'exécution sera également obligatoirement séquentielle en pratique puisqu'un

seul cœur est affecté comme pour « hw_par ».

La figure 5.31 correspond aux temps d'exécution pour une instance de TSP ayant 38 sommets et il s'agit d'une instance ayant 194 sommets pour la figure 5.32. Pour une petite instance (38 sommets), on observe de légères variations, cependant la courte durée d'exécution ne permet pas de conclure clairement, sinon que l'utilisation de la bibliothèque n'engendre pas de surcoût significatif même pour des tâches courtes.

Les temps mesurés pour la seconde instance (194 sommets) sont particulièrement proches pour les programmes écrits sans et avec la bibliothèque à l'exception des politiques d'exécution « staticpool » et « thread ». Ainsi, à nouveau, on observe que la politique d'exécution séquentielle permet d'obtenir des exécutions de durées quasiment identiques à ce que l'on peut atteindre sans utiliser la bibliothèque. À l'inverse, l'utilisation d'une autre politique d'exécution dans un contexte séquentiel (parce qu'un seul cœur est disponible) peut en revanche être coûteux. En effet, celles-ci peuvent avoir besoin de mettre en place des mécanismes, lesquels ne sont généralement pas optimisés pour le cas d'une exécution séquentielle.

Afin de garantir la répétabilité y compris lorsqu'un seul *thread* est possible, l'implémentation des politiques d'exécution doit dans ce cas également produire un comportement cohérent, privant celle-ci d'éventuelles optimisations. Cela explique par ailleurs pourquoi la bibliothèque n'utilise pas automatiquement la politique d'exécution séquentielle lorsque le nombre de cœurs alloués est de 1. Néanmoins, dans le cas où la répétabilité n'a pas besoin d'être assurée au point d'avoir le même déroulement entre une exécution séquentielle et une exécution parallèle, alors il est avisé de la part du développeur d'utiliser la politique d'exécution séquentielle lorsqu'il n'y a qu'un cœur.

Les figures 5.33 et 5.34 présentent les temps d'exécution de programmes parallèles avec $N = 24$, $O = 20$ et $I = 20$. La première étiquette, « hw_par », correspond au programme écrit sans utiliser la bibliothèque pour une exécution parallèle. Les étiquettes préfixées par « sk » correspondent à des programmes écrits en utilisant la bibliothèque et en utilisant différentes politiques d'exécution. La seconde partie de chaque étiquette indique quelle politique d'exécution est utilisée.

Pour une instance de TSP ayant 38 sommets, on obtient les résultats présentés par la figure 5.33. On observe globalement des performances similaires et surtout une décroissance stable du temps d'exécution avec l'augmentation du nombre de cœurs. La politique d'exécution « staticpool » semble particulièrement efficace. Cela se justifie par l'absence de sections critiques (contrairement à ce que l'on a avec « dynamicpool ») et la réutilisation des *threads* créés (contrairement à ce que fait « thread »). La répartition des tâches est également un peu meilleure que celle de « firstlevel » pour les dernières itérations de la boucle principale du GRASP lorsqu'il y a un reste à la division de N par le nombre de cœurs disponibles.

Lorsque l'on travaille sur une instance de TSP avec 194 sommets, on observe les résultats présentés dans la figure 5.34. Ils sont dans l'ensemble équivalents aux précédents, ce qui confirme une certaine indépendance des performances obtenues par rapport à la taille des données traitées.

En faisant varier N de 4 à 20 par pas de 4, on obtient les courbes de la figure 5.35. En particulier pour $N = 4$ (figure 5.35a), $N = 8$ (figure 5.35b) et $N = 12$ (figure 5.35c) pour lesquels c'est très visible, on observe une limite dans l'accélération obtenue pour « firstlevel ». Ce résultat est logique puisque la politique d'exécution utilisée dans ce cas ne parallélisant que le premier niveau possible, si celui-ci correspond à une boucle de k itérations l'accélération maximale que l'on peut obtenir est de k. Les autres politiques d'exécution se comportent de la même manière,

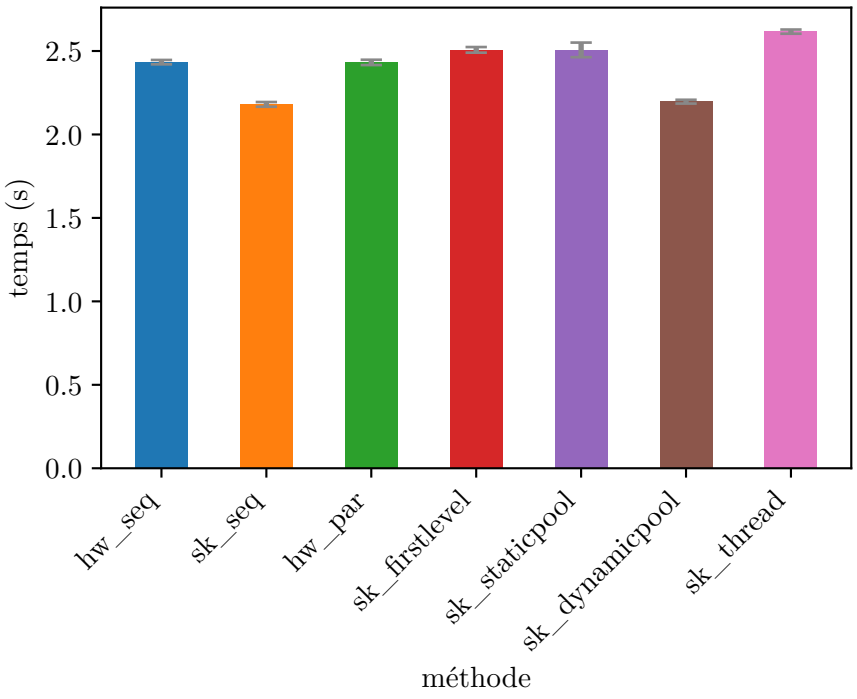


FIGURE 5.31 – Temps d'exécution séquentielle d'un GRASP×ELS pour une instance de TSP de 38 sommets

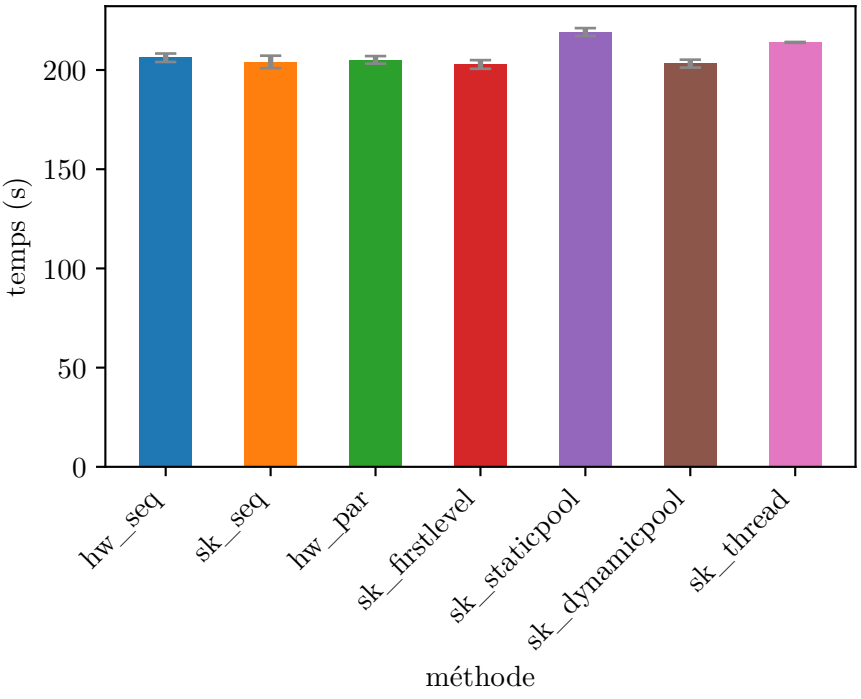


FIGURE 5.32 – Temps d'exécution séquentielle d'un GRASP×ELS pour une instance de TSP de 194 sommets

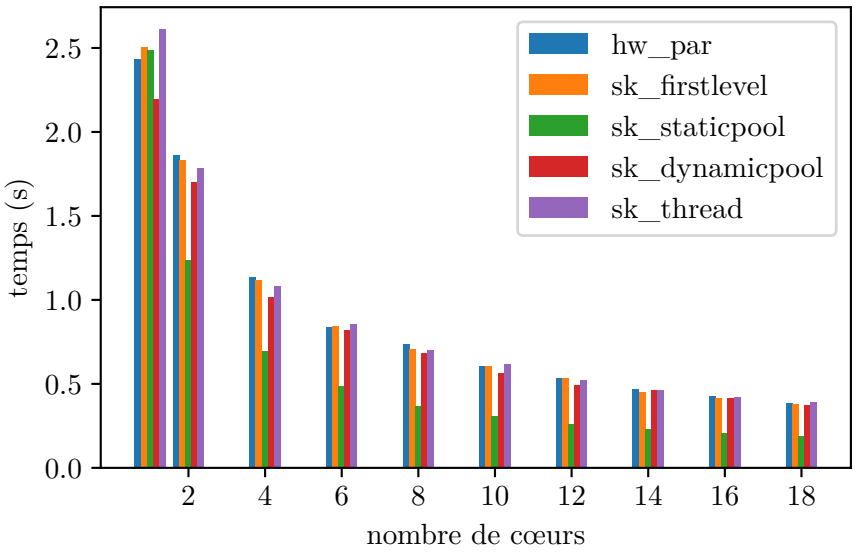


FIGURE 5.33 – Temps d'exécution parallèle d'un GRASP×ELS pour une instance de TSP de 38 sommets

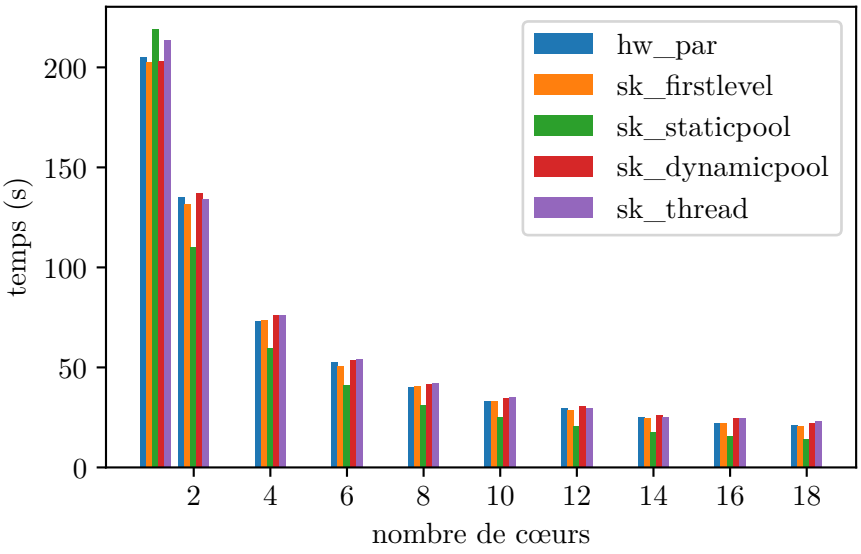


FIGURE 5.34 – Temps d'exécution parallèle d'un GRASP×ELS pour une instance de TSP de 194 sommets

indépendamment du nombre d'itérations du GRASP, ce qui est attendu puisque toutes les trois sont capables de paralléliser de multiples niveaux et donc de tirer profit des itérations parallélisables de l'ELS (au nombre fixe de $I = 20$).

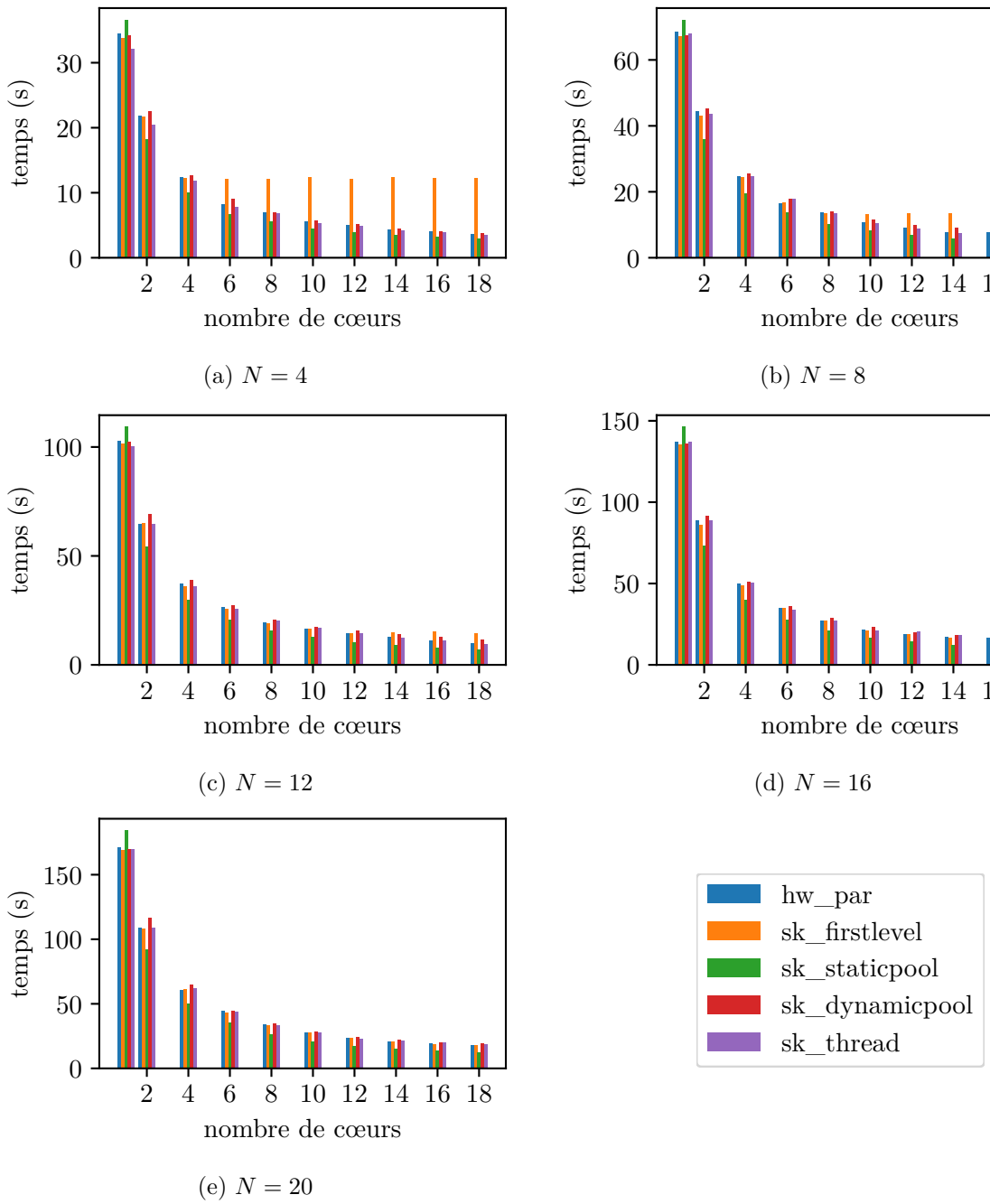
Nous avons voulu vérifier l'effet de la variation du nombre d'itérations de la boucle centrale non parallélisable (la boucle extérieure de l'ELS dont le nombre d'itérations est O). Les courbes présentées dans la figure 5.36 correspondent à des mesures de temps d'exécution pour $N = 4$ et O variant entre 1 et 50. Ces courbes permettent d'observer que d'une manière globale, le comportement reste similaire.

Enfin, sur la base des données précédentes, les figures 5.37 et 5.38 présentent l'accélération obtenue pour les différentes politiques d'exécution (ainsi que pour une parallélisation sans utiliser la bibliothèque) en fonction du nombre de cœurs alloués. La figure 5.37 correspond à l'exécution d'un GRASP×ELS pour lequel $N = 4$, c'est-à-dire que la boucle principale du GRASP effectue 4 itérations qui peuvent être parallélisées. Les valeurs de O et I sont conservées à la valeur par défaut utilisée durant cette section, à savoir 20 pour les deux. Lorsque $N = 4$, on observe en particulier que la parallélisation en utilisant une politique d'exécution ne traitant qu'un niveau est limitée en accélération à la valeur atteinte lorsque le nombre de cœurs alloués égale N, ce qui n'est pas surprenant. Par ailleurs, l'accélération obtenue en utilisant la politique d'exécution par *thread pool* « statique » semble être généralement meilleure que les autres. Dans tous les cas, on observe que l'accélération croît effectivement lorsque le nombre de cœurs alloués croît.

La figure 5.38 correspond quant à elle à l'exécution d'un GRASP×ELS avec $N = 20$. Le comportement global est conservé et l'accélération obtenue est proportionnelle au nombre de cœurs alloués. Pour la politique d'exécution « sk_firstlevel », l'accélération obtenue n'atteint logiquement plus un plafond puisque le nombre de cœurs alloués n'atteint pas, durant ces mesures, la valeur de N.

Les figures 5.39 et 5.40 montrent l'accélération obtenue en fonction de la valeur de N pour un nombre de cœurs alloués défini, respectivement 1 et 18. Dans la figure 5.39, on observe donc le comportement des politiques d'exécution lorsqu'elles sont utilisées pour une exécution séquentielle. Cela confirme les remarques précédentes à propos de la politique « sk_staticpool » qui apparaît comme étant la moins efficace dans ce contexte précis.

En revanche, dans la figure 5.40 on vérifie que pour toute valeur de N cette politique d'exécution est meilleure. Cela semble indiquer que le fait d'effectuer une partie du travail durant la compilation et de déterminer à l'avance à quel *thread* affecter chaque tâche affecte favorablement les performances du programme.



(a) $N = 4$

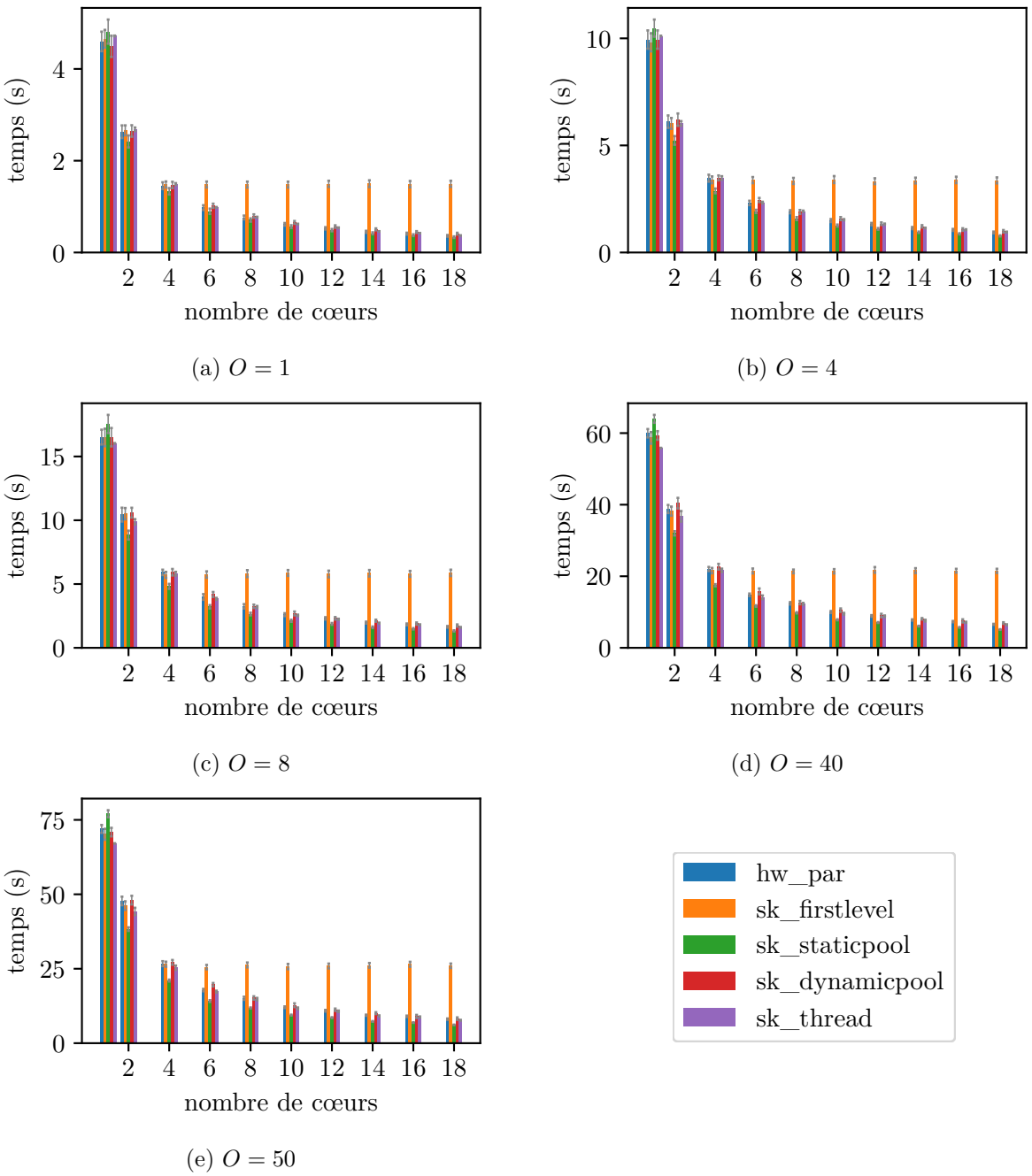
(b) $N = 8$

(c) $N = 12$

(d) $N = 16$

(e) $N = 20$

FIGURE 5.35 – Temps d'exécution parallèle d'un GRASP×ELS pour une instance de TSP de 194 sommets selon le nombre d'itérations du GRASP



(a) $O = 1$

(b) $O = 4$

(c) $O = 8$

(d) $O = 40$

(e) $O = 50$

FIGURE 5.36 – Temps d'exécution parallèle d'un GRASP×ELS pour une instance de TSP de 194 sommets selon le nombre d'itérations non parallélisables de l'ELS

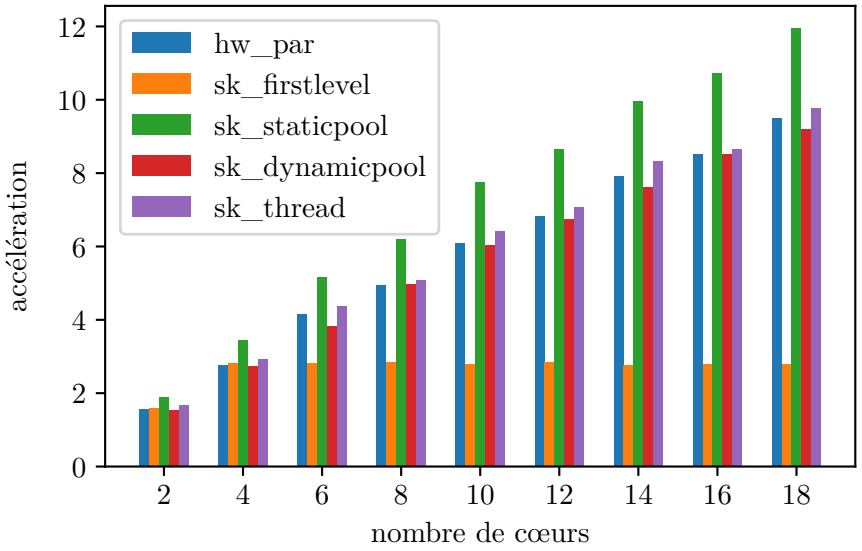


FIGURE 5.37 – Accélération en fonction du nombre de cœurs alloués (GRASP×ELS avec $N = 4$, TSP de 194 sommets)

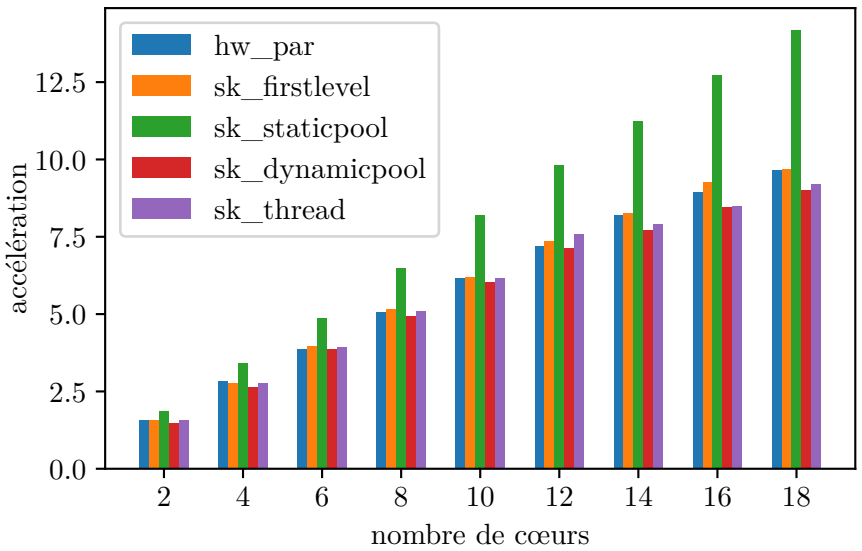


FIGURE 5.38 – Accélération en fonction du nombre de cœurs alloués (GRASP×ELS avec $N = 20$, TSP de 194 sommets)

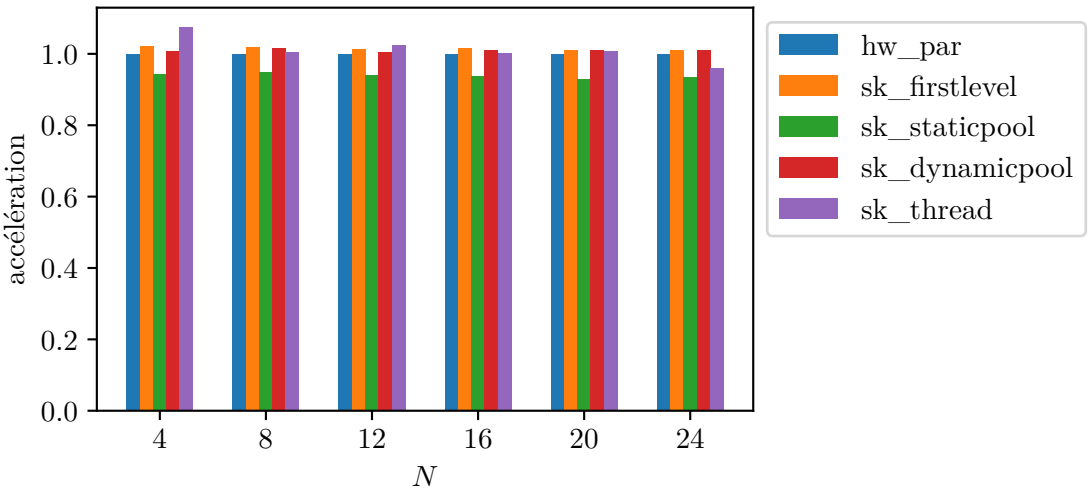


FIGURE 5.39 – Accélération en fonction de N pour 1 cœur alloué (GRASP×ELS, TSP de 194 sommets)

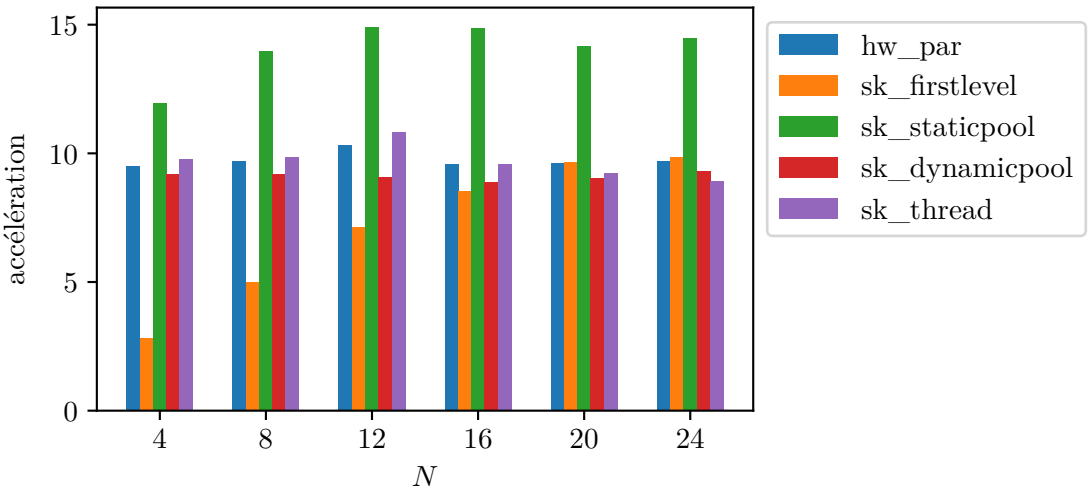


FIGURE 5.40 – Accélération en fonction de N pour 18 cœurs alloués (GRASP×ELS, TSP de 194 sommets)

5.9 Conclusion

Ce chapitre nous a permis d'explorer le concept de squelette algorithmique. Ceux-ci permettent de représenter un algorithme pour qu'un programme puisse en faire usage afin de produire un exécutable dont l'exécution exploite mieux le matériel à sa disposition. Un atout des squelettes algorithmiques est d'isoler l'ensemble des détails de la parallélisation du code de l'utilisateur.

Nous avons dans un premier temps introduit un problème sur lequel appliquer des squelettes algorithmiques, sélectionné dans le domaine de la RO. Celui-ci permet d'illustrer les caractéristiques des différents éléments composant les squelettes algorithmiques. En particulier, la métaheuristique GRASP×ELS exhibe trois couches de boucles dont deux pour lesquelles la parallélisation est possible.

Nous avons proposé une bibliothèque tirant profit de la métaprogrammation du langage C++ depuis l'acquisition du squelette algorithmique jusqu'à la génération du programme. Nous présentons d'abord les concepts fondamentaux de cette bibliothèque : la structure du squelette et les liens notamment, mais aussi la notion d'os, de muscles et de corps. La structure d'un squelette décrit l'algorithme par une composition de motifs d'exécution (les os). Les liens décrivent les flux de données entre les muscles. Ce système fait en grande partie la spécificité de notre bibliothèque.

L'exécution d'un algorithme est conditionnée par une politique d'exécution. Ceci permet de séparer logiquement les motifs d'exécution de la manière dont le programme doit être exécuté : sur CPU (*Central Processing Unit*) ou bien GPU (*Graphics Processing Unit*) ; avec une quantité de *threads* définie ; en orchestrant statiquement ou dynamiquement la répartition des tâches ; ... Différentes politiques d'exécution ont alors été exposées pour en illustrer le fonctionnement.

Grâce au mécanisme des liens et au fonctionnement interne de la bibliothèque (qui permet par exemple l'attribution d'un identifiant unique à toute tâche parallèle), nous avons proposé un moyen de garantir la répétabilité de l'exécution d'un programme stochastique. Cette répétabilité est assurée pour un nombre quelconque de *threads*, y compris par rapport à une exécution séquentielle, ce qui est précieux pour la vérification des résultats. L'intérêt de cette solution initiale réside dans son autonomie : elle consiste en l'affectation à toute tâche parallèle de son contexte propre. Ceci est faisable à la main, mais fastidieux et source d'erreur, pour un nombre de *threads* spécifique. Si ce dernier change, l'affectation doit être revue.

De plus, nous avons étudié différents moyens permettant d'améliorer cette solution. En effet, en supposant une parallélisation infinie, toute tâche parallèle doit posséder un contexte unique. Or, en tenant compte des limites techniques du matériel, nous avons montré comment réduire significativement le nombre de contextes distincts requis, par exemple en n'assurant la répétabilité que jusqu'à une limite supérieure du nombre de cœurs.

Afin de fournir une abstraction supplémentaire aux concepts internes de la bibliothèque, nous proposons un EDSL. Celui-ci a pour vocation principale de montrer qu'il est possible de masquer la complexité de l'interface brute. Cette itération permet la définition de squelettes et de corps et permet aussi de configurer à la volée les paramètres de l'algorithme.

L'application à la RO est utilisée au long de ce chapitre : cela offre un exemple dont les caractéristiques nous ont permis d'introduire, notamment, les problématiques de la parallélisation sur plusieurs niveaux et le traitement des nombres pseudo-aléatoires. Notre bibliothèque a finalement été employée pour résoudre des instances de TSP afin de comparer les performances durant l'exécution du programme entre une version écrite « à la main » et une version générée automatiquement par un squelette algorithmique.

Cette bibliothèque offre différents points d'entrée. Au niveau le plus interne, un développeur peut concevoir de nouveaux os (donc de nouveaux motifs d'exécution). Un expert en parallélisation peut implémenter une nouvelle politique d'exécution. Du point de vue des utilisateurs de cette bibliothèque, deux catégories peuvent être distinguées : ceux qui vont décrire un ensemble de squelettes prêts à être utilisés au sein d'une nouvelle bibliothèque et les utilisateurs finaux qui ajoutent les muscles à ces squelettes pour pouvoir les exécuter.

La répétabilité est garantie par la bibliothèque sous certaines conditions (principalement l'utilisation correcte du système de liens). L'utilisateur peut fournir lui-même les états qui seront utilisés pour la génération de nombres aléatoires, et il reste ainsi à sa charge de valider la qualité scientifique de ceux-ci.

Conclusion

La programmation parallèle induit de nombreuses difficultés par rapport à la programmation séquentielle classique. En premier lieu, il faut identifier les portions de code source qui peuvent être exécutées en parallèle sans invalider leur comportement, c'est-à-dire en conservant le même déroulement lors de leur exécution jusqu'à l'obtention du même résultat. Lorsque ces portions de code sont identifiées, la mise en place d'une stratégie de parallélisation demande un nouvel effort. De plus, une évolution du programme, même mineure, peut nécessiter une maintenance coûteuse puisqu'il faut évaluer à nouveau s'il est possible de l'exécuter en parallèle et mettre à jour le code en conséquence. Un programme efficacement parallélisé, s'il utilise par exemple des nombres pseudo-aléatoires, peut ne plus être répétable d'une exécution à l'autre. D'autre part, en matière de vérification, il est précieux de pouvoir tester les résultats d'un programme parallèle par rapport à son équivalent exécuté séquentiellement. Cette thèse présente des outils conçus pour répondre à différents problèmes introduits par la programmation parallèle, dont la répétabilité de codes stochastiques faisant usage de nombres pseudo-aléatoires.

Afin de pouvoir expliquer la théorie et le fonctionnement des propositions faites dans cette thèse, celle-ci explique les rudiments de la programmation parallèle ainsi que les différentes abstractions et automatisations qui existent. Deux chapitres sont ensuite dédiés à la généricité, notamment en C++, et à la métaprogrammation, notamment celle dite template du C++. Ils détaillent en particulier les fonctionnalités permises par ces paradigmes et qui servent ensuite dans l'implémentation des bibliothèques actives que sont les outils proposés.

La première bibliothèque, présentée dans le chapitre 4, propose une interface pour automatiser la parallélisation d'instructions au sein d'une boucle. Celle-ci emploie les patrons d'expressions pour acquérir une représentation des instructions de la boucle sous la forme d'un ASA (arbre syntaxique abstrait). Cet ASA détaille jusqu'aux opérations sur les indices d'accès aux tableaux – appelées fonctions d'indice – dont les opérandes sont alors connus dès la compilation. Grâce à cela, la bibliothèque vérifie si tous les accès aux données permettent une exécution parallèle, et ce selon les caractéristiques des fonctions d'indice :

1. si toutes sont affines, la bibliothèque répond, avec une spécificité et une sensibilité parfaite, en calculant l'existence de solutions à un ensemble d'équations diophantiennes ;
2. sinon, si toutes sont injectives, elle répond, avec une sensibilité imparfaite, en utilisant un test simplifié ;
3. sinon la bibliothèque suppose par défaut que les instructions ne peuvent être parallélisées.

Cette séquence de tests garantit une spécificité parfaite et empêche donc la parallélisation d'un code qui ne peut l'être sainement.

Ces tests sont appliqués à des groupes d'instructions qui sont formés de sorte que deux instructions quelconques venant de groupes différents sont indépendantes quant aux données qu'elles

utilisent. De cette manière, les instructions qui ne doivent être exécutées en parallèle peuvent être maintenues séparées de celles pouvant être exécutées en parallèle si elles sont indépendantes. Ainsi, la non-parallélisabilité de ces premières instructions ne gêne pas la parallélisabilité des dernières, alors qu'une analyse sur l'ensemble complet aurait logiquement rapporté une réponse négative quant à la possibilité d'exécuter les instructions en parallèle.

À partir des fonctions d'indices seules, la bibliothèque détermine automatiquement si celles-ci sont affines. Quant aux autres propriétés telles que l'injectivité, elles peuvent être indiquées voire déduites. C'est par exemple le cas de l'injectivité si la fonction d'indice est strictement croissante ou strictement décroissante.

Une fois que les ensembles d'instructions parallélisables et non parallélisables sont ainsi définis, la bibliothèque reproduit le code représenté par l'ASA en intégrant ce qui est nécessaire pour que soient exécutées en parallèles les instructions concernées. Pour cela, il est possible d'utiliser différents générateurs. La bibliothèque propose une parallélisation avec OpenMP ou en utilisant des *threads* POSIX (*Portable Operating System Interface*) ou encore une génération des instructions en utilisant la technique du déroulement de boucle.

La bibliothèque a été testée en comparant ses performances avec des programmes équivalents écrits dans les meilleures règles de l'art, de façon « artisanale » (et donc sans l'utiliser). Les temps de compilation, bien qu'il y ait certainement encore des améliorations possibles sur cet aspect, sont raisonnables et permettent une utilisation au sein de projets. Les temps d'exécution montrent que l'abstraction apportée par la bibliothèque n'implique que des surcoûts minimes.

La seconde proposition faite dans cette thèse est une autre bibliothèque active ayant pour objectif la parallélisation assistée par les squelettes algorithmiques. Contrairement à la première proposition, les portions du programme qui peuvent être exécutées en parallèle sont intrinsèquement liées au squelette algorithmique que définit le développeur.

Cette bibliothèque dispose de sa propre manière de concevoir des squelettes algorithmiques. Celle-ci repose sur une séparation initiale de deux concepts : la structure et les liens. La structure du squelette est une composition d'autres structures et d'os, les éléments structurels atomiques introduits dans cette thèse. Chaque os correspond à un motif d'exécution, par exemple l'exécution séquentielle de plusieurs tâches ou l'exécution parallèle d'une même tâche répétée, suivie d'une autre pour sélectionner le meilleur résultat produit. Quant aux liens, il s'agit d'une description des transferts de données entre les différentes tâches à exécuter. Cela se définit en utilisant des paramètres spéciaux, lesquels indiquent à la bibliothèque ce par quoi ils doivent être remplacés (un paramètre de la tâche appelante, la valeur de retour d'une autre, ...).

La structure du squelette permet de savoir quels éléments peuvent être exécutés en parallèle. Il est donc par exemple possible de déterminer le nombre de niveaux de parallélisation dont on dispose pour un squelette donné. Pour cela, une bibliothèque annexe d'outils pour la métaprogrammation template est utilisée. Celle-ci comporte des algorithmes pour parcourir des listes et des arbres de types, et un squelette algorithmique peut être transformé en arbre (et un arbre en liste si nécessaire).

En utilisant, notamment, l'information du nombre de niveaux parallélisables, la bibliothèque permet l'optimisation de la répartition des tâches sur les différents *threads* selon plusieurs politiques d'exécution : *thread pool* ; répartition équilibrée ; répartition équilibrée n'utilisant que le premier niveau ; ... Dans le cas du *thread pool*, l'équilibrage de la charge entre les différents *threads* est automatique et dynamique, au coût d'une synchronisation à effectuer pour l'accès aux tâches à exécuter. La répartition équilibrée proposée suppose une durée similaire dans l'exécution

des différentes tâches et les distribue aux *threads* de manière à ce que chacun en ait, à une près, le même nombre. Cette hypothèse semble raisonnable en particulier pour des os répétant une même tâche plusieurs fois, os fréquemment utilisés dans l'implémentation de métaheuristiques.

Grâce aux connaissances apportées par le choix d'une politique d'exécution et au contrôle permis par les liens, cette thèse propose une solution au problème de la perte de répétabilité lors de l'exécution parallèle d'un programme utilisant des nombres pseudo-aléatoires. Cette solution permet plus généralement le maintien de la répétabilité lors de l'utilisation de données qui doivent n'être utilisées que par un unique *thread*. Ces données doivent donc être associées aux tâches qui partagent le *thread* qui les exécute. Une solution triviale consiste à fournir à chaque tâche ses propres données. Lorsqu'il s'agit de nombres pseudo-aléatoires, cela signifie qu'il faut déterminer une séquence indépendante pour chaque tâche, ce qui devient coûteux, entre autres en mémoire, lorsque le nombre de tâches augmente et que le statut initial du générateur est conséquent (proche de 2,5 Kio par exemple pour un état initial de Mersenne Twister MT19937). En tenant compte de la distribution des tâches sur les *threads*, et ce pour différentes quantités de *threads*, on détermine quelles tâches sont toujours exécutées par un *thread* commun (indépendamment du nombre de *threads*) et on leur associe une séquence de nombres pseudo-aléatoires commune (état initial partagé). Ceci permet de garantir la répétabilité non seulement d'une exécution à l'autre, mais également lorsque le nombre de *threads* varie. Les tâches ayant besoin de nombres pseudo-aléatoires peuvent alors en recevoir un automatiquement au moyen d'un paramètre spécial que les liens permettent d'utiliser.

Cette bibliothèque a été utilisée pour résoudre des instances de TSP (*Travelling Salesman Problem*) par la métaheuristique GRASP×ELS. Les temps d'exécution sont comparés à ceux obtenus avec une implémentation manuelle d'un GRASP×ELS et correspondent à des exécutions parallèles et à des exécutions séquentielles (afin de valider que, même dans ce cas, la bibliothèque n'induit pas de surcoût). Dans tous les cas, les temps obtenus sont analogues.

Limites et perspectives

Nous avons bien conscience que notre première proposition, orientée sur l'analyse et la parallélisation automatique de boucles, est incomplète dans la mesure où de multiples boucles imbriquées ne sont pas détectées. Pour le moment, l'utilisation de multiples boucles imbriquées signifie qu'une seule sera traitée pour la parallélisation. Ce fait n'est pas gênant si l'on considère les architectures matérielles multi-cœurs actuelles. La bibliothèque ne propose pas d'outil pour mettre en place une parallélisation automatique sur plusieurs niveaux car elle ne permet pas, à ce stade, l'utilisation de l'indice d'un niveau à un niveau inférieur. C'est une piste de recherche que nous envisageons d'explorer à l'avenir.

De plus, il sera intéressant de tester différentes méthodes pour appliquer la parallélisation. Notamment l'utilisation d'un *thread pool* afin de voir si l'introduction de sections critiques induit un coût négligeable par rapport au gain supposé apporté par le fait d'éviter la recréation de *threads* à chaque nouvelle boucle parallélisée.

Au niveau de l'analyse et particulièrement des tests permettant de déterminer la parallélisabilité des instructions, l'implémentation du modèle polyédral est également une piste. Celle-ci peut notamment être utile au support de multiples boucles imbriquées, mais peut éventuellement également servir à augmenter la sensibilité du test pour des boucles à un seul niveau lorsque les fonctions d'indice ne sont ni affines ni injectives.

Une autre piste d'amélioration consiste en l'application de transformations sur le code source traité. Actuellement, celui-ci est analysé pour sa parallélisabilité, et s'il ne passe pas le test, il est exécuté séquentiellement. Il est possible, en modifiant astucieusement les instructions, de conserver le comportement du programme en éliminant des dépendances, augmentant donc potentiellement sa parallélisabilité.

Par rapport à la seconde proposition, axée sur la parallélisation assistée par l'utilisation de squelettes algorithmiques, le travail effectué était principalement centré sur l'abstraction apportée par la bibliothèque de squelettes algorithmiques, au détriment de l'optimisation des différentes politiques d'exécution fournies. Ces politiques d'exécution ne sont donc pas implémentées de manière optimale. Une personne dont la parallélisation est le domaine d'expertise pourrait améliorer cela. En outre, il serait très intéressant de tester la piste évoquée dans la section 5.5.4.2, à savoir la préparation, dès la compilation, de la séquence complète des tâches qu'exécutera chaque *thread* afin d'éviter un défaut actuel de la politique d'exécution répartissant les tâches de manière équilibrée : les *threads* des niveaux parallèles, à l'exception du tout premier, sont créés de multiples fois.

Une autre idée est de permettre l'ajustement du poids des tâches à exécuter afin de ne plus supposer un temps d'exécution homogène. En utilisant ces poids, une distribution équilibrée des tâches sur les différents *threads* peut, peut-être, être aussi efficace en termes d'équilibrage de charge que ce que peut accomplir dynamiquement un *thread pool*. De plus, une exécution,

partielle ou sur des données réduites, du programme permettrait éventuellement la génération automatique de ces poids.

Les os proposés au sein de cette thèse répondent aux besoins levés par l'applicatif en RO (Recherche Opérationnelle) que nous avons utilisé. Néanmoins, des motifs tels que le *pipeline*, sont manquants pour une utilisation réellement générale. Certains de ces nouveaux os pourraient nécessiter l'ajout de primitives au sein des exécuteurs.

Le système de liens apporte des avantages intéressants, mais également une syntaxe plus lourde pour le développeur qui, en utilisant d'autres bibliothèques de squelettes algorithmiques, pourrait préférer une valeur par défaut. L'EDSL (*Embedded Domain Specific Language*) introduit en partie cette possibilité de valeur par défaut et même de déduction de liens, mais cela manque aux couches plus basses.

Ce système de liens permet par ailleurs de connaître exactement quelles tâches nécessitent l'utilisation, par exemple, de nombres pseudo-aléatoires. En utilisant cette information, il est possible de réduire encore le nombre de PRNG (*Pseudorandom Number Generator*) devant être créés pour garantir la répétabilité du programme.

Sigles

API *Application Programming Interface* 30, 33, 34, 39, 45, 48

ASA arbre syntaxique abstrait 9, 106, 111–115, 117–119, 122, 201, 202

AST *Abstract Syntax Tree* 75, 76, 91, 111

AVX *Advanced Vector eXtensions* 33, 45

BLP *Bit-Level Parallelism* 39

C++ AMP C++ *Accelerated Massive Parallelism* 32

CPP *C PreProcessor* 5, 17, 51, 53–55, 75, 98

CPU *Central Processing Unit* 29, 30, 32, 33, 45, 198

CTAD *Class Template Argument Deduction* 58, 112

CUDA *Compute Unified Device Architecture* 30, 33

DSL *Domain Specific Language* 96, 98, 142, 143

DSP *Digital Signal Processor* 39

EBO *Empty Base Optimization* 78

EDSL *Embedded Domain Specific Language* 20, 96, 103, 106, 112, 140, 143, 182, 185, 188, 198, 206

ELS *Evolutionary Local Search* 10, 11, 15, 20, 144, 148–150, 152–154, 156, 158, 165, 167, 175, 184–189, 191–198, 203

ET *Expression Templates* 91–94, 96, 98, 100, 101, 106

FIFO *First In, First Out* 36

FPGA *Field-Programmable Gate Array* 33, 39

GPGPU *General-Purpose computing on GPU (Graphics Processing Unit)* 32

GPU *Graphics Processing Unit* 30–33, 198, 207

GRASP *Greedy Randomized Adaptive Search Procedure* 10, 11, 15, 20, 144, 146–148, 150–156, 158, 165, 167, 175, 176, 182, 184–198, 203

HPC *High Performance Computing* 28

ID *Instruction Decode* 41

IEEE *Institute of Electrical and Electronics Engineers* 31

IF *Instruction Fetch* 41

ILP *Instruction-Level Parallelism* 39

ILS *Iterative Local Search* 10, 15, 144, 146–149, 154

IPC *Inter-Process Communication* 28

MA *Memory Access* 41

MIMD *Multiple-Instruction stream – Multiple-Data stream* 9, 31, 33

MISD *Multiple-Instruction stream – Single-Data stream* 31

MMX *multimedia extensions* 33, 45

MPI *Message Passing Interface* 30

NUMA *Non-Uniform Memory Access* 9, 28–30

OpenCL *Open Computing Language* 30

OpenMP *Open Multi-Processing* 30, 48

PEPS Premier Entré, Premier Sorti 36

PGCD plus grand commun diviseur 124, 125

POO Programmation Orientée Objet 67

POSIX *Portable Operating System Interface* 17, 31, 33, 36, 37, 39, 45, 46, 48, 202

PRNG *Pseudorandom Number Generator* 7, 10, 139, 154, 155, 175–177, 180, 181, 188, 206

RAII *Resource Acquisition Is Initialisation* 60

RAW *Read After Write* 15, 43, 108

RO Recherche Opérationnelle 23, 140, 143, 144, 148, 173, 198, 206

RRID *Resource Release Is Destruction* 60

SFINAE *Substitution Failure Is Not An Error* 17, 62–65, 67, 97, 119

SIMD *Single-Instruction stream – Multiple-Data stream* 9, 31–33, 45, 48, 208

SISD *Single-Instruction stream – Single-Data stream* 9, 31

SMP *Symmetric MultiProcessing* 28

SSE *Streaming SIMD (Single-Instruction stream – Multiple-Data stream) Extension* 33, 45

TAD *Template Argument Deduction* 17, 56–58, 185

TBB *Threading Building Blocks* 49, 142

TMP *template metaprogramming* 78

TSP *Travelling Salesman Problem* 10, 11, 20, 23, 143–146, 158, 189–192, 194–198, 203

UAL Unité Arithmétique et Logique 40

UMA *Uniform Memory Access* 9, 28–30

UVF Unité de calcul en Virgule Flottante 40

WAR *Write After Read* 15, 43, 44, 108

WAW *Write After Write* 15, 43, 44, 108

WB *WriteBack* 41

Bibliographie

Ahmad, Ishfaq, Kwok, Yu-Kwong, Wu, Min-You et Shu, Wei (1997). « Automatic Parallelization and Scheduling of Programs on Multiprocessors Using CASCH ». Dans : *Proceedings of the International Conference on Parallel Processing*. ICPP '97. Washington, DC, USA : IEEE Computer Society, p. 288-291. isbn : 978-0-8186-8108-0. doi : 10.1109/ICPP.1997.622657 (pages 21, 106).

Aho, Alfred V, Sethi, Ravi et Ullman, Jeffrey D (1986). *Compilers: Principles, Techniques, and Tools*. India : Pearson Education. isbn : 978-81-7808-046-8 (page 74).

Aldinucci, Marco, Danelutto, Marco, Meneghin, Massimiliano, Torquati, Massimo et Kilpatrick, Peter (2009). « Efficient Streaming Applications on Multi-Core with FastFlow: The Biosequence Alignment Test-Bed ». Dans : *ParCo 2009: Parallel Computing*. T. 19, p. 273-280. doi : 10.3233/978-1-60750-530-3-273 (page 143).

Alexandrescu, Andrei (2001). *Modern C++ Design: Generic Programming and Design Patterns Applied*. 2. print. The C++ In-Depth Series. Boston, Mass. : Addison-Wesley. 352 p. isbn : 0-201-70431-5 (page 86).

Amdahl, Gene M. (1967). « Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities ». Dans : *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference on - AFIPS '67 (Spring)*. The April 18-20, 1967, Spring Joint Computer Conference. Atlantic City, New Jersey : ACM Press, p. 483. doi : 10.1145/1465482.1465560 (page 26).

Arabnejad, Hamid, Bispo, João, Barbosa, Jorge G. et Cardoso, João M.P. (2018). « AutoPar-Clava: An Automatic Parallelization Source-to-Source Tool for C Code Applications ». Dans : *Proceedings of the 9th Workshop and 7th Workshop on Parallel Programming and RunTime Management Techniques for Manycore Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms - PARMA-DITAM '18*. The 9th Workshop and 7th Workshop. Manchester, United Kingdom : ACM Press, p. 13-19. isbn : 978-1-4503-6444-7. doi : 10.1145/3183767.3183770 (page 42).

Artigas, Pedro V., Gupta, Manish, Midkiff, Samuel P. et Moreira, José E. (2000). « Automatic Loop Transformations and Parallelization for Java ». Dans : *Proceedings of the 14th International Conference on Supercomputing - ICS '00*. The 14th International Conference. Santa Fe, New Mexico, United States : ACM Press, p. 1-10. isbn : 978-1-58113-270-0. doi : 10.1145/335231.335232 (pages 44, 107).

Asai, Kenichi (2014). « Compiling a Reflective Language Using MetaOCaml ». Dans : *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences - GPCE 2014*. The 2014 International Conference. ACM Press, p. 113-122. isbn : 978-1-4503-3161-6. doi : 10.1145/2658761.2658775 (page 76).

Bachelet, Bruno et Yon, Loïc (2017). « Designing Expression Templates with Concepts ». Dans : *Software: Practice and Experience* 47.11, p. 1521-1537. ISSN : 00380644. DOI : 10.1002/spe.2483 (page 64).

Barney, Blaise (2009). « POSIX Threads Programming ». Dans : *Lawrence Livermore National Laborator*, p. 26 (page 30).

Batcher, Kenneth (1980). « Design of a Massively Parallel Processor ». Dans : *IEEE Transactions on Computers* C-29.9, p. 836-840. ISSN : 0018-9340. DOI : 10.1109/TC.1980.1675684 (page 39).

Batlle, J (2002). « A New FPGA/DSP-Based Parallel Architecture for Real-Time Image Processing ». Dans : *Real-Time Imaging* 8.5, p. 345-356. ISSN : 10772014. DOI : 10.1006/rtim.2001.0273 (page 39).

Benoit, Anne et Cole, Murray (2005). « Two Fundamental Concepts in Skeletal Parallel Programming ». Dans : *Computational Science – ICCS 2005*. T. 3515. Berlin, Heidelberg : Springer Berlin Heidelberg, p. 764-771. ISBN : 978-3-540-26043-1. DOI : 10.1007/11428848_98 (pages 49, 140).

Bernstein, A. J. (1966). « Analysis of Programs for Parallel Processing ». Dans : *IEEE Transactions on Electronic Computers* EC-15.5, p. 757-763. ISSN : 0367-7508. DOI : 10.1109/PGEC.1966.264565 (page 108).

Binkley, David (2007). « Source Code Analysis: A Road Map ». Dans : *Future of Software Engineering (FOSE '07)*. Future of Software Engineering. Minneapolis, MN, USA : IEEE, p. 104-119. ISBN : 978-0-7695-2829-8. DOI : 10.1109/FOSE.2007.27 (page 74).

Blume, William, Eigenmann, Rudolf, Faigin, Keith, Grout, John, Hoeflinger, Jay, Padua, David, Petersen, Paul, Pottenger, William, Rauchwerger, Lawrence, Tu, Peng et Weatherford, Stephen (1995). « Effective Automatic Parallelization with Polaris ». Dans : *International Journal of Parallel Programming* (pages 21, 106).

Blumofe, Robert D., Joerg, Christopher F., Kuszmaul, Bradley C., Leiserson, Charles E., Randall, Keith H. et Zhou, Yuli (1996). « Cilk: An Efficient Multithreaded Runtime System ». Dans : *Journal of Parallel and Distributed Computing* 37.1, p. 55-69. ISSN : 07437315. DOI : 10.1006/jpdc.1996.0107 (page 48).

Bondhugula, Uday, Hartono, Albert, Ramanujam, J. et Sadayappan, P. (2008). « A Practical Automatic Polyhedral Parallelizer and Locality Optimizer ». Dans : *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI '08*. The 2008 ACM SIGPLAN Conference. Tucson, AZ, USA : ACM Press, p. 101. ISBN : 978-1-59593-860-2. DOI : 10.1145/1375581.1375595 (page 107).

Bringmann, Roger A., Mahlke, Scott A., Hank, Richard E., Gyllenhaal, John C. et Hwu, Wen-mei W. (1993). « Speculative Execution Exception Recovery Using Write-Back Suppression ». Dans : *Proceedings of the 26th Annual International Symposium on Microarchitecture*. Proceedings of 26th Annual International Symposium on Microarchitecture (Cat. No.93TH0602-3). Austin, TX, USA : IEEE, p. 214-223. ISBN : 978-0-8186-5280-6. DOI : 10.1109/MICRO.1993.282757 (page 42).

Butenhof, David R. (1997). *Programming with POSIX Threads*. Addison-Wesley Professional Computing Series. Reading, Mass : Addison-Wesley. 381 p. ISBN : 978-0-201-63392-4 (page 33).

Campbell, Duncan K. G. (1996). *Towards the Classification of Algorithmic Skeletons*. YCS 276. University of York department of computer science YCS (page 141).

Chamberlain, B.L., Callahan, D. et Zima, H.P. (2007). « Parallel Programmability and the Chapel Language ». Dans : *The International Journal of High Performance Computing Applications* 21.3, p. 291-312. ISSN : 1094-3420, 1741-2846. DOI : 10.1177/1094342007078442 (pages 21, 106).

Chan, Bryan et Abdelrahman, Tarek S. (2004). « Run-Time Support for the Automatic Parallelization of Java Programs ». Dans : *The Journal of Supercomputing* 28.1, p. 91-117. ISSN : 0920-8542. DOI : 10.1023/B:SUPE.0000014804.20789.21 (pages 22, 106).

Ciechanowicz, Philipp, Poldner, Michael et Kuchen, Herbert (2009). « The Münster Skeleton Library Muesli - a Comprehensive Overview ». Dans : (page 143).

Cole, Murray (1989). *Algorithmic Skeletons: Structured Management of Parallel Computation*. Cambridge, MA, USA : MIT Press. ISBN : 978-0-262-53086-6 (pages 49, 140).

Collard, Jean-François (1995). « Automatic Parallelization of While-Loops Using Speculative Execution ». Dans : *International Journal of Parallel Programming* 23.2, p. 191-219. ISSN : 0885-7458, 1573-7640. DOI : 10.1007/BF02577789 (pages 44, 107).

Courtois, P. J., Heymans, F. et Parnas, D. L. (1971). « Concurrent Control with "Readers" and "Writers" ». Dans : *Communications of the ACM* 14.10, p. 667-668. ISSN : 00010782. DOI : 10.1145/362759.362813 (page 36).

cppreference (2011). *Reference Declaration*. URL : https://en.cppreference.com/w/cpp/language/reference#Reference_collapsing (visité le 06/11/2019) (pages 18, 69).

Cray, Seymour R (1978). « Computer Vector Register Processing ». Brev. 4128880 (pages 31, 40).

Dagum, L. et Menon, R. (1998). « OpenMP: An Industry Standard API for Shared-Memory Programming ». Dans : *IEEE Computational Science and Engineering* 5.1, p. 46-55. ISSN : 1070-9924. DOI : 10.1109/99.660313 (pages 30, 48).

Dahl, Ole-Johan et Nygaard, Kristen (1966). « SIMULA: An ALGOL-Based Simulation Language ». Dans : *Communications of the ACM* 9.9, p. 671-678. ISSN : 00010782. DOI : 10.1145/365813.365819 (page 28).

Dantzig, George, Fulkerson, Ray et Johnson, Selmer (1954). « Solution of a Large-Scale Traveling-Salesman Problem ». Dans : *Journal of the operations research society of America*, p. 393-410 (page 143).

Davidson, J.W. et Jinturkar, S. (1995). « Improving Instruction-Level Parallelism by Loop Unrolling and Dynamic Memory Disambiguation ». Dans : *Proceedings of the 28th Annual International Symposium on Microarchitecture*. Proceedings of MICRO'95 : 28th Annual IEEE/ACM International Symposium on Microarchitecture. Ann Arbor, MI, USA : IEEE, p. 125-132. ISBN : 978-0-8186-7349-8. DOI : 10.1109/MICRO.1995.476820 (page 128).

Dijkstra, Edsger W. (1968). « Cooperating Sequential Processes ». Dans : *The Origin of Concurrent Programming*. Sous la dir. de Per Brinch Hansen. New York, NY : Springer New York, p. 65-138. ISBN : 978-1-4419-2986-0. DOI : 10.1007/978-1-4757-3472-0_2 (page 35).

Dimov, Peter, Hinnant, Howard E. et Abrahams, David (2002). *The Forwarding Problem: Arguments*. URL : http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2002/n1385.htm (page 69).

Dittamo, Cristian, Cisternino, Antonio et Danelutto, Marco (2007). « Parallelization of C# Programs through Annotations ». Dans : *Computational Science – ICCS 2007*. Sous la dir. d'Yong Shi, Geert Dick van Albada, Jack Dongarra et Peter M. A. Sloot. Réd. par David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern,

John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi et Gerhard Weikum. T. 4488. Lecture Notes in Computer Science. Berlin, Heidelberg : Springer Berlin Heidelberg, p. 585-592. doi : 10.1007/978-3-540-72586-2_86 (page 48).

Drummond, Chris (2009). « Replicability Is Not Reproducibility: Nor Is It Good Science ». Dans : *Proceedings of the Evaluation Methods for Machine Learning Workshop*. 26th International Conference for Machine Learning. Montreal, Quebec, Canada, p. 696-701 (page 172).

Ernstsson, August, Li, Lu et Kessler, Christoph (2018). « SkePU 2: Flexible and Type-Safe Skeleton Programming for Heterogeneous Parallel Systems ». Dans : *International Journal of Parallel Programming* 46.1, p. 62-80. issn : 0885-7458, 1573-7640. doi : 10.1007/s10766-017-0490-5 (pages 22, 49, 143).

Estérie, Pierre, Falcou, Joel, Gaunard, Mathias, Lapresté, Jean-Thierry et Lacassagne, Lionel (2014). « The Numerical Template Toolbox: A Modern C++ Design for Scientific Computing ». Dans : *Journal of Parallel and Distributed Computing* 74.12, p. 3240-3253. issn : 07437315. doi : 10.1016/j.jpdc.2014.07.002 (page 98).

Falcou, J., Sérot, J., Chateau, T. et Lapresté, J. T. (2006). « Quaff: Efficient C++ Design for Parallel Skeletons ». Dans : *Parallel Computing*. Algorithmic Skeletons 32.7, p. 604-615. issn : 0167-8191. doi : 10.1016/j.parco.2006.06.001 (pages 49, 143).

Falcou, Joel, Sérot, Jocelyn, Pech, Lucien et Lapresté, Jean-Thierry (2008). « Meta-Programming Applied to Automatic SMP Parallelization of Linear Algebra Code ». Dans : *Euro-Par 2008 – Parallel Processing*. T. 5168. Lecture Notes in Computer Science. Berlin, Heidelberg : Springer Berlin Heidelberg, p. 729-738. isbn : 978-3-540-85450-0. doi : 10.1007/978-3-540-85451-7_78 (pages 22, 107).

Feo, Thomas A. et Resende, Mauricio G. C. (1989). « A Probabilistic Heuristic for a Computationally Difficult Set Covering Problem ». Dans : *Operations Research Letters* 8.2, p. 67-71. issn : 01676377. doi : 10.1016/0167-6377(89)90002-3 (page 146).

Flynn, Michael J. (1972). « Some Computer Organizations and Their Effectiveness ». Dans : *IEEE Transactions on Computers* C-21.9, p. 948-960. issn : 0018-9340. doi : 10.1109/TC.1972.5009071 (page 31).

Fonseca, Alcides, Cabral, Bruno, Rafael, João et Correia, Ivo (2016). « Automatic Parallelization: Executing Sequential Programs on a Task-Based Parallel Runtime ». Dans : *International Journal of Parallel Programming* 44.6, p. 1337-1358. issn : 0885-7458, 1573-7640. doi : 10.1007/s10766-016-0426-5 (pages 21, 106).

Gamma, Erich, Helm, Richard, Johnson, Ralph et Vlissides, John (1995). *Design Patterns: Elements of Reusable Software Architecture*. Addison-Wesley (pages 48, 52, 77).

Geuns, S J, Bekooij, M J G, Bijlsma, T et Corporaal, H (2011). « Parallelization of While Loops in Nested Loop Programs for Shared-Memory Multiprocessor Systems ». Dans : *2011 Design, Automation & Test in Europe*. 2011 Design, Automation & Test in Europe. Grenoble : IEEE, p. 1-6. isbn : 978-3-9810801-8-6. doi : 10.1109/DATE.2011.5763118 (page 45).

Gingras, Armando R. (1990). « Dining Philosophers Revisited ». Dans : *ACM SIGCSE Bulletin* 22.3, p. 21. issn : 0097-8418. doi : 10.1145/101085.101091 (page 39).

Gordon, Daniel M. (1998). « A Survey of Fast Exponentiation Methods ». Dans : *Journal of Algorithms* 27.1, p. 129-146. issn : 01966774. doi : 10.1006/jagm.1997.0913 (page 83).

Griebl, M., Lengauer, C. et Wetzel, S. (1998). « Code Generation in the Polytope Model ». Dans : *Proceedings. 1998 International Conference on Parallel Architectures and Compilation*

Techniques (Cat. No.98EX192). 1998 International Conference on Parallel Architectures and Compilation Techniques. Paris, France : IEEE Comput. Soc, p. 106-111. ISBN : 978-0-8186-8591-0. DOI : 10.1109/PACT.1998.727179 (page 107).

GRIEBL, Martin et COLLARD, Jean-François (1995). « Generation of Synchronous Code for Automatic Parallelization of While Loops ». Dans : *EURO-PAR '95 Parallel Processing*. Sous la dir. de Seif HARIDI, Khayri ALI et Peter MAGNUSSON. Réd. par Gerhard GOOS, Juris HARTMANIS et Jan van LEEUWEN. T. 966. Lecture Notes in Computer Science. Berlin, Heidelberg : Springer Berlin Heidelberg, p. 313-326. ISBN : 978-3-540-60247-7. DOI : 10.1007/BFb0020474 (page 45).

GUSTAFSON, John L (1988). « The Scaled-Sized Model: A Revision of Amdahl's Law ». Dans : *Supercomputing*. International Conference on Supercomputing. Boston, MA, USA, p. 130-133 (page 27).

HALSTEAD, Robert H. (1985). « MULTILISP: A Language for Concurrent Symbolic Computation ». Dans : *ACM Transactions on Programming Languages and Systems (TOPLAS)* 7.4, p. 501-538. ISSN : 0164-0925, 1558-4593. DOI : 10.1145/4472.4478 (page 47).

HÄRDTLEIN, Jochen, LINKE, Alexander et PFLAUM, Christoph (2005). « Fast Expression Templates: Object-Oriented High Performance Computing ». Dans : *Lecture Notes in Computer Science*. Springer-Verlag, p. 1055-1063 (page 112).

HILL, David R. C. (2015). « Parallel Random Numbers, Simulation, and Reproducible Research ». Dans : *Computing in Science & Engineering* 17.4, p. 66-71. ISSN : 1521-9615. DOI : 10.1109/MCSE.2015.79 (page 173).

HILL, David R. C., MAZEL, Claude, PASSERAT-PALMBACH, Jonathan et TRAORE, Mamadou K. (2013). « Distribution of Random Streams for Simulation Practitioners: CPE HPCS 2010 Special Issue Submission ». Dans : *Concurrency and Computation: Practice and Experience* 25.10, p. 1427-1442. ISSN : 15320626. DOI : 10.1002/cpe.2942 (pages 173, 174).

HILL, J.M.D. et SKILLICORN, D.B. (1998). « Practical Barrier Synchronisation ». Dans : *Proceedings of the Sixth Euromicro Workshop on Parallel and Distributed Processing - PDP '98 -*. Sixth Euromicro Workshop on Parallel and Distributed Processing - PDP '98 -. Madrid, Spain : IEEE Comput. Soc, p. 438-444. ISBN : 978-0-8186-8332-9. DOI : 10.1109/EMPDP.1998.647231 (page 37).

HOARE, C. A. R. (1971). « Proof of a Program: FIND ». Dans : *Communications of the ACM* 14.1, p. 39-45. ISSN : 00010782. DOI : 10.1145/362452.362489 (page 74).

HOBEROCK, Jared, MARATHE, Jaydeep, GARLAND, Michael, GIROUX, Olivier, GROVER, Vinod, LAKSBERG, Artur, SUTTER, Herb et ROBISON, Arch (2013). *A Parallel Algorithms Library*. URL : http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3724.pdf (page 48).

HORWITZ, S., PFEIFFER, P. et REPS, T. (1989). « Dependence Analysis for Pointer Variables ». Dans : *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation - PLDI '89*. The ACM SIGPLAN 1989 Conference. Portland, Oregon, United States : ACM Press, p. 28-40. ISBN : 978-0-89791-306-5. DOI : 10.1145/73141.74821 (page 44).

HOWARD, John H. (1973). « Mixed Solutions for the Deadlock Problem ». Dans : *Communications of the ACM* 16.7, p. 427-430. ISSN : 00010782. DOI : 10.1145/362280.362290 (page 39).

Hwu, W. et Patt, Y. N. (1986). « HPSm, a High Performance Restricted Data Flow Architecture Having Minimal Functionality ». Dans : *ACM SIGARCH Computer Architecture News* 14.2, p. 297-306. ISSN : 01635964. DOI : 10.1145/17356.17391 (page 44).

Iglberger, Klaus, Hager, Georg, Treibig, Jan et Rude, Ulrich (2012). « High Performance Smart Expression Template Math Libraries ». Dans : *2012 International Conference on High Performance Computing & Simulation (HPCS)*. 2012 International Conference on High Performance Computing & Simulation (HPCS). Madrid, Spain : IEEE, p. 367-373. ISBN : 978-1-4673-2362-8. DOI : 10.1109/HPCSim.2012.6266939 (page 92).

Ioannidis, John P. A. (2005). « Why Most Published Research Findings Are False ». Dans : *PLoS Medicine* 2.8, e124. ISSN : 1549-1676. DOI : 10.1371/journal.pmed.0020124 (page 172).

ISO et C++ committee (2011). *Programming Language - C++*. Standard ISO/IEC 14882 :2011. International Organization for Standardization, p. 1338. URL : http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3337.pdf (pages 71, 102).

ISO et C++ committee (2020). *Programming Language - C++*. Standard ISO/IEC CD 14882. International Organization for Standardization, p. 1815 (page 64).

Jackson, D.J., Whiteside, D.M. et Wurtz, L.T. (1992). « Exploiting Bit-Level Parallelism in Boolean Matrix Operations for Graph Analysis ». Dans : *Proceedings IEEE Southeastcon '92*. IEEE Southeastcon '92. Birmingham, AL, USA : IEEE, p. 838-841. ISBN : 978-0-7803-0494-9. DOI : 10.1109/SECON.1992.202252 (page 40).

Järvi, Jaakko et Powell, Gary (2001). « Side Effects and Partial Function Application in C++ ». Dans : *Proceedings of the Multiparadigm Programming with OO Languages Workshop (MPOOL'01)*, p. 17 (page 102).

Järvi, Jaakko et Powell, Gary (2003). *Boost.Lambda*. URL : https://www.boost.org/doc/libs/1_73_0/doc/html/lambda.html (visité le 10/06/2020) (page 100).

Järvi, Jaakko, Powell, Gary et Lumsdaine, Andrew (2003). « The Lambda Library: Unnamed Functions in C++ ». Dans : *Software: Practice and Experience* 33.3, p. 259-291. ISSN : 0038-0644. DOI : 10.1002/spe.504 (page 100).

Jo, Youngjoon, Goldfarb, Michael et Kulkarni, Milind (2013). « Automatic Vectorization of Tree Traversals ». Dans : *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*. 22nd International Conference on Parallel Architectures and Compilation Techniques. Edinburgh, UK : IEEE, p. 12. ISBN : 978-1-4799-1021-2. DOI : 10.1109/PACT.2013.6618832 (page 45).

Keller, Robert M. (1975). « Look-Ahead Processors ». Dans : *ACM Computing Surveys* 7.4, p. 177-195. ISSN : 03600300. DOI : 10.1145/356654.356657 (page 41).

Kennedy, Ken (1994). « Compiler Technology for Machine-Indepenent Parallel Programming ». Dans : *International Journal of Parallel Programming* 22.1, p. 79-98. ISSN : 0885-7458, 1573-7640. DOI : 10.1007/BF02577793 (page 45).

Kernighan, Brian W et Ritchie, Dennis M (1988). *The C Programming Language*. Upper Saddle River, NJ : Prentice Hall. ISBN : 978-0-13-110362-7 (page 75).

Kirby, R.C. (2003). « A New Look at Expression Templates for Maxtrix Computation ». Dans : *Computing in Science & Engineering* 5.3, p. 66-70. ISSN : 1521-9615. DOI : 10.1109/MCISE.2003.1196309 (page 92).

Kish, Laszlo B (2002). « End of Moore's Law: Thermal (Noise) Death of Integration in Micro and Nano Electronics ». Dans : *Physics Letters A* 305.3-4, p. 144-149. ISSN : 03759601. DOI : 10.1016/S0375-9601(02)01365-8 (page 26).

Kuchen, Herbert (2002). « A Skeleton Library ». Dans : *Euro-Par 2002 Parallel Processing*. T. 2400. Berlin, Heidelberg : Springer Berlin Heidelberg, p. 620-629. ISBN : 978-3-540-44049-9. DOI : 10.1007/3-540-45706-2_86 (pages 22, 49, 141).

Kuchen, Herbert et Striegnitz, Jörg (2002). « Higher-Order Functions and Partial Applications for a C++ Skeleton Library ». Dans : *Proceedings of the 2002 Joint ACM-ISCOPE Conference on Java Grande - JGI '02*. The 2002 Joint ACM-ISCOPE Conference. Seattle, Washington, USA : ACM Press, p. 122-130. ISBN : 978-1-58113-599-2. DOI : 10.1145/583810.583824 (page 101).

Landi, William et Ryder, Barbara G. (1991). « Pointer-Induced Aliasing: A Problem Taxonomy ». Dans : *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL '91*. The 18th ACM SIGPLAN-SIGACT Symposium. Orlando, Florida, United States : ACM Press, p. 93-103. ISBN : 978-0-89791-419-2. DOI : 10.1145/99583.99599 (page 112).

Lazarescu, Mihai T. et Lavagno, Luciano (2012). « Dynamic Trace-Based Data Dependency Analysis for Parallelization of C Programs ». Dans : *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*. 2012 12th IEEE Working Conference on Source Code Analysis and Manipulation (SCAM). Riva del Garda, Italy : IEEE, p. 126-131. ISBN : 978-0-7695-4783-1. DOI : 10.1109/SCAM.2012.15 (page 107).

Legaux, Joeffrey, Loulergue, Frédéric et Jubertie, Sylvain (2013). « OSL: An Algorithmic Skeleton Library with Exceptions ». Dans : *Procedia Computer Science* 18, p. 260-269. ISSN : 18770509. DOI : 10.1016/j.procs.2013.05.189 (page 143).

Lengauer, C. et Griebl, M. (1995). « On the Parallelization of Loop Nests Containing While Loops ». Dans : *Proceedings of the First Aizu International Symposium on Parallel Algorithms/Architecture Synthesis*. The First Aizu International Symposium on Parallel Algorithms/Architecture Synthesis. Fukushima, Japan : IEEE Comput. Soc. Press, p. 10-18. ISBN : 978-0-8186-7038-1. DOI : 10.1109/AISPAS.1995.401360 (page 45).

Leyton, Mario et Piquer, José M. (2010). « Skandium: Multi-Core Programming with Algorithmic Skeletons ». Dans : *2010 18th Euromicro Conference on Parallel, Distributed and Network-Based Processing*. 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP 2010). Pisa : IEEE, p. 289-296. ISBN : 978-1-4244-5673-4. DOI : 10.1109/PDP.2010.26 (page 143).

Li, Shigang, Hoefler, Torsten et Snir, Marc (2013). « NUMA-Aware Shared-Memory Collective Communication for MPI ». Dans : *Proceedings of the 22nd International Symposium on High-Performance Parallel and Distributed Computing*. HPDC '13. New York, New York, USA : Association for Computing Machinery, p. 85-96. ISBN : 978-1-4503-1910-2. DOI : 10.1145/2462902.2462903 (page 28).

Lilis, Yannis et Savidis, Anthony (2019). « A Survey of Metaprogramming Languages ». Dans : *ACM Computing Surveys* 52.6, p. 1-39. ISSN : 03600300. DOI : 10.1145/3354584 (page 75).

Liskov, B. et Shrira, L. (1988). « Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems ». Dans : *ACM SIGPLAN Notices* 23.7, p. 260-267. ISSN : 03621340. DOI : 10.1145/960116.54016 (page 47).

Lopez, M. Graham, Bergstrom, Christopher, Li, Ying Wai, Elwasif, Wael et Hernandez, Oscar (2016). « Using C++ AMP to Accelerate HPC Applications on Multiple Platforms ». Dans : *High Performance Computing*. Sous la dir. de Michela Taufer, Bernd Mohr et Julian M. Kunkel. T. 9945. Lecture Notes in Computer Science. Cham : Springer International

Publishing, p. 563-576. ISBN : 978-3-319-46078-9. DOI : 10.1007/978-3-319-46079-6_38 (page 32).

Lourenço, Helena R., Martin, Olivier C. et Stützle, Thomas (2003). « Iterated Local Search ». Dans : *Handbook of Metaheuristics*. Sous la dir. de Fred Glover et Gary A. Kochenberger. International Series in Operations Research & Management Science. Boston, MA : Springer US, p. 320-353. ISBN : 978-0-306-48056-0. DOI : 10.1007/0-306-48056-5_11 (page 146).

Loveman, D. B. (1993). « High Performance Fortran ». Dans : *IEEE Parallel & Distributed Technology: Systems & Applications* 1.1, p. 25-42. ISSN : 1063-6552. DOI : 10.1109/88.219857 (pages 21, 106).

Luebke, David (2008). « CUDA: Scalable Parallel Programming for High-Performance Scientific Computing ». Dans : *2008 5th IEEE International Symposium on Biomedical Imaging: From Nano to Macro*. 2008 5th IEEE International Symposium on Biomedical Imaging (ISBI 2008). Paris, France : IEEE, p. 836-838. ISBN : 978-1-4244-2002-5. DOI : 10.1109/ISBI.2008.4541126 (pages 30, 33).

Mack, Chris A. (2011). « Fifty Years of Moore's Law ». Dans : *IEEE Transactions on Semiconductor Manufacturing* 24.2, p. 202-207. ISSN : 0894-6507. DOI : 10.1109/TSM.2010.2096437 (page 26).

Maddock, John (2002). *A Proposal to Add Type Traits to the Standard Library*. URL : http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2002/n1345.html (visité le 01/09/2020) (page 96).

Marques, Ricardo, Paulino, Hervé, Alexandre, Fernando et Medeiros, Pedro D. (2013). « Algorithmic Skeleton Framework for the Orchestration of GPU Computations ». Dans : *Euro-Par 2013 Parallel Processing*. T. 8097. Berlin, Heidelberg : Springer Berlin Heidelberg, p. 874-885. ISBN : 978-3-642-40046-9. DOI : 10.1007/978-3-642-40047-6_86 (page 49).

Mathews, Manju et Abraham, Jisha P (2016). « Automatic Code Parallelization with OpenMP Task Constructs ». Dans : *2016 International Conference on Information Science (ICIS)*. 2016 International Conference in Information Science (ICIS). Kochi, India : IEEE, p. 233-238. ISBN : 978-1-5090-1987-8. DOI : 10.1109/INFOSCI.2016.7845333 (page 42).

Matsuzaki, Kiminori, Hu, Zhenjiang et Takeichi, Masato (2006). « Towards Automatic Parallelization of Tree Reductions in Dynamic Programming ». Dans : *Proceedings of the Eighteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures - SPAA '06*. The Eighteenth Annual ACM Symposium. Cambridge, Massachusetts, USA : ACM Press, p. 39. ISBN : 978-1-59593-452-9. DOI : 10.1145/1148109.1148116 (page 45).

Mattson, Timothy G., Sanders, Beverly A. et Massingill, Berna (2005). *Patterns for Parallel Programming*. Boston : Addison-Wesley. 355 p. ISBN : 978-0-321-22811-6 (page 48).

McCool, Michael, Reinders, James et Robison, Arch (2012). *Structured Parallel Programming: Patterns for Efficient Computation*. Elsevier. 433 p. ISBN : 978-0-12-391443-9. Google Books : 2hYqeoO8t8IC (page 48).

Meneide, JeanHeyd (2020). *std::embed and #depend*. URL : http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1040r6.html (page 79).

Munshi, Aaftab (2009). « The OpenCL Specification ». Dans : *2009 IEEE Hot Chips 21 Symposium (HCS)*. 2009 IEEE Hot Chips 21 Symposium (HCS). Stanford, CA : IEEE, p. 1-314. ISBN : 978-1-4673-8873-3. DOI : 10.1109/HOTCHIPS.2009.7478342 (pages 30, 33).

Musser, David R. et Stepanov, Alexander A. (1989). « Generic Programming ». Dans : *Symbolic and Algebraic Computation*. Sous la dir. de P. Gianni. Réd. par G. Goos, J. Hartmanis,

D. Barstow, W. Brauer, P. Brinch Hansen, D. Gries, D. Luckham, C. Moler, A. Pnueli, G. Seegmüller, J. Stoer et N. Wirth. T. 358. Berlin, Heidelberg : Springer Berlin Heidelberg, p. 13-25. isbn : 978-3-540-51084-0. doi : 10.1007/3-540-51084-2_2 (page 52).

Nam Sung Kim, Austin, T., Blaauw, D., Mudge, T., Flautner, K., Jie S. Hu, Irwin, M.J., Kandemir, M. et Narayanan, V. (2003). « Leakage Current: Moore's Law Meets Static Power ». Dans : *Computer* 36.12, p. 68-75. issn : 0018-9162. doi : 10.1109/MC.2003.1250885 (page 21).

Neth, Brandon et Strout, Michelle Mills (2019). « Automatic Parallelization of Irregular X86-64 Loops ». Dans : *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). Washington, DC, USA : IEEE, p. 266-266. isbn : 978-1-72811-436-1. doi : 10.1109/CGO.2019.8661167 (pages 44, 107).

Nicolau, A., Potasman, R. et Wang, H. (1992). « Register Allocation, Renaming and Their Impact on Fine-Grain Parallelism ». Dans : *Languages and Compilers for Parallel Computing*. Sous la dir. d'Utpal Banerjee, David Gelernter, Alex Nicolau et David Padua. T. 589. Lecture Notes in Computer Science. Berlin/Heidelberg : Springer-Verlag, p. 218-235. isbn : 978-3-540-55422-6. doi : 10.1007/BFb0038667 (page 44).

Patterson, Jason R. C. (1995). « Accurate Static Branch Prediction by Value Range Propagation ». Dans : *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation - PLDI '95*. The ACM SIGPLAN 1995 Conference. La Jolla, California, United States : ACM Press, p. 67-78. isbn : 978-0-89791-697-4. doi : 10.1145/207110.207117 (page 41).

Peleg, Alex, Wilkie, Sam et Weiser, Uri (1997). « Intel MMX for Multimedia PCs ». Dans : *Communications of the ACM* 40.1, p. 24-38. issn : 00010782. doi : 10.1145/242857.242865 (page 33).

Perach, Ben et Weiss, Shlomo (2018). « SiMT-DSP: A Massively Multithreaded DSP Architecture ». Dans : *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 26.8, p. 1413-1426. issn : 1063-8210, 1557-9999. doi : 10.1109/TVLSI.2018.2817564 (page 39).

Pereda, Alexis, Hill, David R. C., Mazel, Claude et Bachelet, Bruno (2018). « Static Loop Parallelization Decision Using Template Metaprogramming ». Dans : *2018 International Conference on High Performance Computing & Simulation (HPCS)*. Orleans : IEEE, p. 1015-1021. isbn : 978-1-5386-7878-7. doi : 10.1109/HPCS.2018.00159 (page 123).

Pereda, Alexis, Hill, David R. C., Mazel, Claude, Yon, Loïc et Bachelet, Bruno (2020). « Processing Algorithmic Skeletons at Compile-Time ». Dans : *21ème Congrès de La Société Française de Recherche Opérationnelle et d'Aide à La Décision (ROADEF)*. Montpellier, France. url : https://hal.archives-ouvertes.fr/hal-02573660 (page 156).

Philippe, Jolan et Loulergue, Frédéric (2019). « PySke: Algorithmic Skeletons for Python ». Dans : *The 2019 International Conference on High Performance Computing & Simulation (HPCS)*. Dublin, Ireland, p. 40-47 (page 143).

Popper, Karl (2005). *The Logic of Scientific Discovery*. Routledge. 545 p. isbn : 978-1-134-47002-0. Google Books : LWSBAgAAQBAJ (page 172).

Prins, Christian (2009). « A GRASP × Evolutionary Local Search Hybrid for the Vehicle Routing Problem ». Dans : *Bio-Inspired Algorithms for the Vehicle Routing Problem*. Studies

in Computational Intelligence. Berlin, Heidelberg : Springer, p. 35-53. ISBN : 978-3-540-85152-3. DOI : 10.1007/978-3-540-85152-3_2 (page 148).

PROTIC, J., TOMASEVIC, M. et MILUTINOVIC, V. (1996). « Distributed Shared Memory: Concepts and Systems ». Dans : *IEEE Parallel & Distributed Technology: Systems & Applications* 4.2, p. 63-71. ISSN : 10636552. DOI : 10.1109/88.494605 (page 30).

QUINN, Michael Jay (2003). *Parallel Programming in C with MPI and OpenMP*. T. 526. McGraw-Hill. 529 p. ISBN : 978-0-07-123265-4 (page 35).

RAMAN, S.K., PENTKOVSKI, V. et KESHAVA, J. (2000). « Implementing Streaming SIMD Extensions on the Pentium III Processor ». Dans : *IEEE Micro* 20.4, p. 47-57. ISSN : 02721732. DOI : 10.1109/40.865866 (page 33).

RAMON-CORTES, Cristian, AMELA, Ramon, EJARQUE, Jorge, CLAUSS, Philippe et BADIA, Rosa (2018). *AutoParallel: A Python Module for Automatic Parallelization and Distributed Execution of Affine Loop Nests*, p. 13 (pages 44, 107).

RANNS, Nina et VANDEVOORDE, Daveed (2018). *Down with typename!* URL : http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0634r2.html (page 88).

REVZIN, Barry, SMITH, Richard, SUTTON, Andrew et VANDEVOORDE, Daveed (2020). *If Consteval*. URL : http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p1938r2.html (page 115).

RIEGER, Christoph, WREDE, Fabian et KUCHEN, Herbert (2019). « Musket: A Domain-Specific Language for High-Level Parallel Programming with Algorithmic Skeletons ». Dans : *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*. SAC '19 : The 34th ACM/SIGAPP Symposium on Applied Computing. Limassol Cyprus : ACM, p. 1534-1543. ISBN : 978-1-4503-5933-7. DOI : 10.1145/3297280.3297434 (pages 21, 142, 143).

ROBINSON, Julia (1949). *On the Hamiltonian Game (a Traveling Salesman Problem)*. Rand project air force arlington va. (page 143).

ROSCOE, Andrew William et HOARE, Charles Antony Richard (1988). « The Laws of OCCAM Programming ». Dans : *Theoretical Computer Science* 60.2, p. 177-229. ISSN : 0304-3975. DOI : 10.1016/0304-3975(88)90049-7 (pages 21, 106).

SAIDANI, Tarik, FALCOU, Joel, TADONKI, Claude, LACASSAGNE, Lionel et ETIEMBLE, Daniel (2009). « Algorithmic Skeletons within an Embedded Domain Specific Language for the CELL Processor ». Dans : IEEE, p. 67-76. ISBN : 978-0-7695-3771-9. DOI : 10.1109/PACT.2009.21 (page 143).

SCHMIDT, Douglas C. (1998). « Evaluating Architectures for Multithreaded Object Request Brokers ». Dans : *Communications of the ACM* 41.10, p. 54-60. ISSN : 00010782. DOI : 10.1145/286238.286248 (page 174).

SHEARD, Tim (2001). « Accomplishments and Research Challenges in Meta-Programming ». Dans : *Semantics, Applications, and Implementation of Program Generation*. Sous la dir. de Walid TAHA. Réd. par Gerhard GOOS, Juris HARTMANIS et Jan van LEEUWEN. T. 2196. Berlin, Heidelberg : Springer Berlin Heidelberg, p. 2-44. ISBN : 978-3-540-42558-8. DOI : 10.1007/3-540-44806-3_2 (page 74).

SHI, Yuan (1996). *Reevaluating Amdahl's Law and Gustafson's Law*. technical. Computer Sciences Department, Temple University (MS :38-24) (page 27).

SIEK, Jeremy, GREGOR, Douglas, GARCIA, Ronald, WILLCOCK, Jeremiah, JÄRVI, Jaakko et LUMSDAINE, Andrew (2005). *Concepts for C++0x*. URL : http://www.open-std.org/JTC1/SC22/wg21/docs/papers/2005/n1758.pdf (page 64).

Siek, Jeremy et Lumsdaine, Andrew (2000). « Concept Checking: Binding Parametric Polymorphism in C++ ». Dans : First Workshop on C++ Template Programming (page 64).

Smith, James E. (1998). « A Study of Branch Prediction Strategies ». Dans : *25 Years of the International Symposia on Computer Architecture (Selected Papers) - ISCA '98*. 25 Years of the International Symposia. Barcelona, Spain : ACM Press, p. 202-215. isbn : 978-1-58113-058-4. doi : 10.1145/285930.285980 (page 41).

Smith, Richard, Sutton, Andrew et Vandevoorde, Daveed (2018a). *Immediate Functions*. url : http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1073r3.html (page 82).

Smith, Richard, Sutton, Andrew et Vandevoorde, Daveed (2018b). *std::is_constant_ evaluated*. url : http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0595r2.html (page 115).

Stallings, W. (1988). « Reduced Instruction Set Computer Architecture ». Dans : *Proceedings of the IEEE* 76.1, p. 38-55. issn : 00189219. doi : 10.1109/5.3287 (page 41).

Striegnitz, Jörg (2000). « Making C++ Ready for Algorithmic Skeletons ». Dans : p. 10 (page 143).

Stroustrup, Bjarne (1997). *The C++ Programming Language*. 3rd ed. Reading, Mass : Addison-Wesley. 910 p. isbn : 978-0-201-88954-3 (page 60).

Stroustrup, Bjarne (2017). *Concepts: the future of generic programming*. url : http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0557r0.pdf (page 64).

Sutter, Herb (2019). *Metaclasses: Generative C++*. url : http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0707r4.pdf (pages 76, 79).

Tamai, T. et Torimitsu, Y. (1992). « Software Lifetime and Its Evolution Process over Generations ». Dans : *Proceedings Conference on Software Maintenance 1992*. Conference on Software Maintenance 1992. Orlando, FL, USA : IEEE Comput. Soc. Press, p. 63-69. isbn : 978-0-8186-2980-8. doi : 10.1109/ICSM.1992.242557 (page 176).

Tomczak, Lukasz Jaroslaw (2012). « GPU Ray Marching of Distance Fields ». Asmussens Alle, Building 305, DK-2800 Kgs. Lyngby, Denmark : Technical University of Denmark. 79 p. (page 31).

Toussaint, Hélène (2010). « Algorithmique rapide pour les problèmes de tournées et d'ordonnancement ». Clermont-Ferrand 2. url : http://www.theses.fr/2010CLF22053 (visité le 28/05/2018) (page 144).

Turing, A. M. (1937). « Computability and λ-Definability ». Dans : *Journal of Symbolic Logic* 2.4, p. 153-163. issn : 0022-4812, 1943-5886. doi : 10.2307/2268280 (page 100).

Unruh, Erwin (1994). *Prime Number Computation*. ANSI X3J16-94-0075/ISO WG21-462 (pages 77, 78).

Vandevoorde, Daveed et Josuttis, Nicolai M. (2010). *C++ Templates: The Complete Guide*. Addison-Wesley. 528 p. isbn : 978-0-201-73484-3 (pages 52, 62).

Vandevoorde, Daveed, Josuttis, Nicolai M. et Gregor, Douglas (2017). *C++ Templates: The Complete Guide*. Addison-Wesley. 788 p. isbn : 978-0-321-71412-1 (page 52).

Veldhuizen, Todd L. (1995). « Expression Templates ». Dans : *C++ Report* 7, p. 26-31 (pages 91, 111).

Veldhuizen, Todd L. (2000). « Blitz++: The Library That Thinks It Is a Compiler ». Dans : *Advances in Software Tools for Scientific Computing*. T. 10. Berlin, Heidelberg : Springer

Berlin Heidelberg, p. 57-87. ISBN : 978-3-540-66557-1. DOI : 10.1007/978-3-642-57172-5_2 (page 92).

VELDHUIZEN, Todd L. (2003). *C++ Templates Are Turing Complete*. Indiana University Computer Science (page 77).

VELDHUIZEN, Todd L. et GANNON, Dennis (1998). « Active Libraries: Rethinking the Roles of Compilers and Libraries ». Dans : *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-Operable Scientific and Engineering Computing*. SIAM Press, p. 286-295 (pages 22, 78, 143).

VIDEAU, Brice, POUGET, Kevin, GENOVESE, Luigi, DEUTSCH, Thierry, KOMATITSCH, Dimitri, DESPREZ, Frédéric et MÉHAUT, Jean-François (2018). « BOAST: A Metaprogramming Framework to Produce Portable and Efficient Computing Kernels for HPC Applications ». Dans : *The International Journal of High Performance Computing Applications* 32.1, p. 28-44. ISSN : 1094-3420, 1741-2846. DOI : 10.1177/1094342017718068 (pages 22, 107).

VOLLMANN, Detlef (2016). *Why Joining_thread from P0206 Is a Bad Idea*. URL : http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0379r0.html (page 47).

VOUFO, Larisse, ZALEWSKI, Marcin et LUMSDAINE, Andrew (2011). « ConceptClang: An Implementation of C++ Concepts in Clang ». Dans : *Proceedings of the Seventh ACM SIGPLAN Workshop on Generic Programming - WGP '11*. The Seventh ACM SIGPLAN Workshop. Tokyo, Japan : ACM Press, p. 71. ISBN : 978-1-4503-0861-8. DOI : 10.1145/2036918.2036929 (page 64).

VOUTILAINEN, Ville (2016). *A Joining Thread*. URL : http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0206r1.html (page 47).

WALKER, David W et DONGARRA, Jack J (1996). « MPI: A Standard Message Passing Interface ». Dans : *Supercomputer*, p. 15 (page 30).

WILLHALM, Thomas et POPOVICI, Nicolae (2008). « Putting Intel® Threading Building Blocks to Work ». Dans : *Proceedings of the 1st International Workshop on Multicore Software Engineering*. IWMSE '08. New York, NY, USA : ACM, p. 3-4. ISBN : 978-1-60558-031-9. DOI : 10.1145/1370082.1370085 (page 49).

WOLF, Steffen et MERZ, Peter (2007). « Evolutionary Local Search for the Super-Peer Selection Problem and the p-Hub Median Problem ». Dans : *Hybrid Metaheuristics*. Sous la dir. de Thomas BARTZ-BEIELSTEIN, María José BLESA AGUILERA, Christian BLUM, Boris NAUJOKS, Andrea ROLI, Günter RUDOLPH et Michael SAMPELS. Lecture Notes in Computer Science. Berlin, Heidelberg : Springer, p. 1-15. ISBN : 978-3-540-75514-2. DOI : 10.1007/978-3-540-75514-2_1 (page 148).

WREDE, Fabian, RIEGER, Christoph et KUCHEN, Herbert (2020). « Generation of High-Performance Code Based on a Domain-Specific Language for Algorithmic Skeletons ». Dans : *The Journal of Supercomputing* 76.7, p. 5098-5116. DOI : 10.1007/s11227-019-02825-6 (page 142).

ZHANG, Hong, MILLS, Richard T., RUPP, Karl et SMITH, Barry F. (2018). « Vectorized Parallel Sparse Matrix-Vector Multiplication in PETSc Using AVX-512 ». Dans : *Proceedings of the 47th International Conference on Parallel Processing*. ICPP 2018 : 47th International Conference on Parallel Processing. Eugene OR USA : ACM, p. 1-10. ISBN : 978-1-4503-6510-9. DOI : 10.1145/3225058.3225100 (page 33).

ZHANG, Yang, ZHANG, Jingjun et ZHANG, Dongwen (2009). « Implementing and Testing Producer-Consumer Problem Using Aspect-Oriented Programming ». Dans : *2009 Fifth International*

Conference on Information Assurance and Security. 2009 Fifth International Conference on Information Assurance and Security. Xi'An China : IEEE, p. 749-752. ISBN : 978-0-7695-3744-3. DOI : 10.1109/IAS.2009.41 (page 39).

Zima, Hans P, Bast, Heinz-J et Gerndt, Michael (1988). « SUPERB: A Tool for Semi-Automatic MIMD/SIMD Parallelization ». Dans : *Parallel Computing* 6.1, p. 1-18. ISSN : 0167-8191. DOI : 10.1016/0167-8191(88)90002-6 (pages 21, 106).