Static loop parallelization decision using template metaprogramming

Alexis Pereda, David R.C. Hill, Claude Mazel, Bruno Bachelet
Université Clermont Auvergne, CNRS, LIMOS
Clermont-Ferrand, France
{alexis.pereda, david.hill, claude.mazel, bruno.bachelet}@uca.fr

Abstract

This article proposes to use C++ template metaprogramming techniques to decide at compile-time which parts of a code sequence in a loop can be parallelized. The approach focuses on characterizing the way a variable is accessed in a loop (reading or writing), first to decide how the loop should be split to enable the analysis for parallelization on each part, and then to decide if the iterations inside each loop are independent so that they can be run in parallel. The conditions that enable the parallelization of a loop are first explained to justify the proposed decision algorithm exposed. Then, a C++ library-based solution is presented that uses expression templates to get the relevant information necessary for the parallelization decision of a loop, and metaprograms to decide whether to parallelize the loop and generate a parallel code.

Keywords – code parallelization; metaprogramming; C++ expression templates; parallelization decision

1 Introduction

Today's computers are capable of executing multiple instructions in parallel, and developers must adapt their code to get benefits from that. There are now many tools to make this task easier, allowing developers to indicate which pieces of their code will be executed in parallel. Among these tools, there are MPI, POSIX Threads, OpenMP, Intel Threading Building Blocks (TBB). With such tools, developers still have to decide whether the code can actually be parallelized or not. This decision depends mainly on which data is accessed and how it is accessed (reading and/or writing), which would require a dependence analysis. However, a wrong decision from the developer will lead to incorrect code, meaning a parallel code that produces a result different from its sequential counterpart. In this article, we focus on detecting if a loop can be parallelized (if parallelized, does the loop produce the expected result?), but we do not consider the performance aspect (if parallelized, does the loop run faster?).

The data dependency information can be processed by some code analyser that must be executed before the compilation (optionally requiring annotations to make the processing easier) by the way of meta-compilation or compiler plugin. The former is rather complex to implement and implies redoing nearly the compiler work. With a compiler plugin, users will be tied to the compilers for which it has been designed.

Our research is focused on identifying whether program segments are parallelizable or not with C++ Template Metaprogramming (TMP) independently from the underlying C++ compiler. TMP is part of the C++ compilation process, thus it is standard and is available independently of the compiler, and it permits code re-writing in a syntax-checked way by allowing us to intervene in the compilation process (without being a plugin). The solution presented in this article uses Expression Templates (ET) [16] to retrieve relevant information for parallelization from the code sequence of a loop, and common TMP techniques [1] to design metaprograms to analyse this information to possibly split the loop, to decide the parallelization and to produce the parallel code. This solution, that operates mainly at compile-time, attempts to avoid, as possible, any execution time overhead.

Section 2 introduces related work. In section 3, we present the conditions required to detect parallelizable loops and to run them without altering the results. From these conditions, we present a two-step decision algorithm that first identifies the parts of the code sequence inside the loop that are independent, in order to possibly split the loop, and then, on each part of the loop, detects if the iterations are independent so the part can be parallelized. Then, section 4 shows how we can implement this detection at language level with C++ TMP, based on ET for information retrieval and metaprograms for analysis and parallel code generation. Examples are also presented to demonstrate the automatic parallelization of loops with our library-based solution. Section 5 presents performance results of our proposed design.

2 Related work

Automatic parallelization tools are studied since many years [12, 8]. They usually work as meta-compilers [20, 2], producing a parallel source code from a sequential source code, or as compilers [6], producing directly an executable. More recently, we can find complete infrastructure dedicated to automatic parallelization [15] and also advanced graph and operation research tools enabling a better parallelization [7]. If we focus on recent C++ oriented solutions, there are dynamic and static propositions.

A tool proposed by Bauer et al., Legion [4], helps developing parallel programs by defining logical regions. Those regions make it possible for Legion to do run-time dependence analysis. Then dynamic task scheduling is done, considering all acquired information about data dependencies. However, we think running all dependence analysis during run-time generates an overhead that could possibly be reduced by doing some of the work during compile-time.

Barve's sequential to parallel C++ code converter [3] is an example of metaprogramming used to enable parallelism from a sequential code. It runs dependence analysis, based of Banerjee's test [19], and generates a suitable parallelized version of the sequential input code. This solution demonstrates the possibility to process data dependencies statically. It requires a first pass on a sequential C++ code before compiling it.

MetaPASS [10] is a library allowing implicit task-based parallelism from a sequential C++ code. Its dependence properties are acquired statically using C++ TMP. It then relies on a run-time dependence analyser, making that tool both static and dynamic.

Our solution makes use of C++ TMP for the full dependence analysis and parallel code generation. This way, we can aim to very little execution time overhead, like it has already been achieved in similar contexts [13].

3 Conditions for parallel loops

In this section, we will see what conditions allow us to execute a loop in parallel, meaning running iterations of the loop in parallel, without changing the final result of the code. As parallelizing a loop induces performing the iterations in an undefined order, we focus here on producing a parallel code only if its correctness can be preserved. The performance aspects are not discussed here.

In the listing 1, as it is, the iterations of the program segment at lines 3 to 6 cannot be executed in parallel because of line 4. This instruction requires each iteration of the loop to be executed in the correct order. The instruction at line 4 accesses the same array element of c as in the instruction at line 6, first for writing and then for reading, so we should not separate them. However the two other lines, 3 and 5, are dependent on each other but not with lines 4 and 6. That means the program in listing 2 is equivalent.

```
1 /* a, b, c are arrays */
2 for(int i = 0; i < n; ++i) {
3   a[i] = a[i] * b[i];
4   c[i] = c[i+1] + d[i];
5   b[i] = b[i] + 1;
6   d[i] = c[i];
7 }
```

Listing 1: Loop example

```
1 /* a, b, c are arrays */
2 for(int i = 0; i < n; ++i) {
3   a[i] = a[i] * b[i];
4   b[i] = b[i] + 1;
5 }
6 for(int i = 0; i < n; ++i) {
7   c[i] = c[i+1] + d[i];
8   d[i] = c[i];
9 }
```

Listing 2: Example of loop splitting

Now, the second loop still cannot be run in parallel, whereas the first loop can. This section will explain first how to decide whenever a program segment can be split, in order to separate a loop in several loops that will be analysed independently for parallelization. This section will then present how we can determine if a loop can be run in parallel.

3.1 Conditions for permutation

Three conditions are sufficient to allow the permutation of two program segments P_a and P_b, with W_i the set of the output variables of P_i and R_i the set of the input variables of P_i. These are the conditions given by Bernstein [5].

$$W_b \cap R_a = \emptyset \quad (1)$$

$$R_b \cap W_a = \emptyset \quad (2)$$

$$W_a \cap W_b = \emptyset \quad (3)$$

A program segment $P = \{I_1, ..., I_n\}$ can be composed of one or more instructions I_1 to I_n. Algorithm 1 presents a process to identify independent instructions of P, and to group dependent ones. This algorithm relies on Bernstein's conditions. We assume that function BernsteinTest(x, y) returns `true` if the conditions are met for the program segments x and y.

To identify the instructions of P that cannot be permuted, a set G of independent program segments can be progressively built as follows. Starting with an empty set G, each instruction I_k, considered as a program segment $P_a = \{I_k\}$, is tested for Bernstein's conditions with each program segment P_b of G. If P_a and P_b invalidate the conditions, P_b is removed from G and P_b is added into P_a. When no more program segment $P_b \in G$ dependent from P_a can be found, P_a is added to G. At any step of

Algorithm 1 Grouping dependent instructions

```
input: a list of instructions P = {I_1, ..., I_n}
output: sets of dependent instructions
function GroupDependentInstructions(P)
    G ← ∅
    for all I_k ∈ P do
        P_a ← {I_k}
        for all P_b ∈ G do
            if not BernsteinTest(P_a, P_b) then
                G ← G \ {P_b}
                P_a ← P_a ∪ P_b
            end if
        end for
        G ← G ∪ {P_a}
    end for
    return G
end function
```

the process, G contains independent program segments, and after considering all the instructions of segment P, G contains the program segments of P that can be permuted.

3.2 Loop-level parallelism

In loop-level parallelism, we will only consider loop-carried dependencies. These are between two distinct iterations of the loop. Loop iterations can be seen as k consecutive program segments, with k the number of iterations, so we can use Bernstein's conditions to decide if the loop iterations are independent.

To achieve this, we must define how the program segment executed in the loop accesses its variables. There are two types of variables accessed in loops: regular variables and elements of arrays (identified by an index). We can see arrays as multiple regular variables, one per array element: one-dimensional array a is noted as vector $a = (a_j)_{j=0..n_a-1}$, where each element a_j corresponds to a regular variable that is the element of the array a at index j.

For all iterations of the loop, regular variables are accessed as defined in section 3.1, so we can define W^{reg} and R^{reg} the sets of written and read regular variables of the program segment. Arrays require a more complex construction. For each array a, we will note F_a the set of functions $f : \mathbb{N} \to \mathbb{N}$ that are used to calculate actual indices to access the array in the program segment. F_a^W is the subset of F_a that contains functions used to write into a, and F_a^R is the subset of F_a that contains functions used to read a. As an example, for the array `c` in listing 1, $F_c^W = \{i \mapsto i\}$ and $F_c^R = \{i \mapsto i, i \mapsto i+1\}$. Then, if W^{arr} and R^{arr} are the sets of written, resp. read, arrays of the program segment, we can construct W_i^{arr} and R_i^{arr}, the sets of elements written, resp. read, at iteration i of the program segment:

$$W_i^{arr} = \bigcup_{a \in W^{arr}} \{a_j \mid j \in \mathbb{N}, \exists f \in F_a^W, f(i) = j\} \quad (4)$$

$$R_i^{arr} = \bigcup_{a \in R^{arr}} \{a_j \mid j \in \mathbb{N}, \exists f \in F_a^R, f(i) = j\} \quad (5)$$

Using that, we can define the full sets of written and read variables for each iteration i, W_i and R_i:

$$W_i = W^{reg} \cup W_i^{arr}$$
$$R_i = R^{reg} \cup R_i^{arr}$$

In order to know if iterations are independent, we can use Bernstein's conditions over all iterations:

$$\forall i = 1..k, \quad W_i \cap (\bigcup_{j \neq i} R_j) = \emptyset \quad (6)$$

$$\forall i = 1..k, \quad R_i \cap (\bigcup_{j \neq i} W_j) = \emptyset \quad (7)$$

$$\bigcap_{i=1}^{k} W_i = \emptyset \quad (8)$$

For regular variables, the implications are immediate:

- reading does not break any condition
- writing always breaks the condition (8)

For arrays, we need to focus on indices. We can convert the conditions (6), (7) and (8) for each array a to:

$$\forall i = 1..k, \quad \nexists f \in F_a^W / \exists j \neq i, g \in F_a^R, f(i) = g(j) \quad (6')$$

$$\forall i = 1..k, \quad \nexists f \in F_a^R / \exists j \neq i, g \in F_a^W, f(i) = g(j) \quad (7')$$

$$\forall i = 1..k, \quad \nexists f \in F_a^W / \exists j \neq i, g \in F_a^W, f(i) = g(j) \quad (8')$$

To meet condition (8′), an array must not be written at a same index in different iterations. To meet conditions (6′) and (7′), if an array is written at an index n at iteration i, it must not be read at the same index n in any distinct iteration j. Expressed this way, one would need to enumerate all iterations to check Bernstein's conditions. This approach is not possible at compile-time if the loop range is dynamic (set at run-time), so we propose a simplified condition that can detect only a subset of possible parallelization cases. Future work will be to define better conditions to detect more possible parallelization cases.

Our proposed condition is, for each array a in the loop:

$$F_a^W = \emptyset \vee |F_a| = 1 \quad (9)$$

If the first part of condition (9) ($F_a^W = \emptyset$) is met, there are only reading accesses to the array so it will meet all three Bernstein's conditions. If the second part of condition (9) ($|F_a| = 1$) is met, there is only one function to calculate the effective index to access the array, so, provided that this function is injective, even if accesses are readings and writings, there is no overlap between two distinct iterations.

This condition is always satisfied in the cases where indices calculations are linear. If not we will consider here that Bernstein's conditions are not satisfied.

For a loop with a program segment, its body, we can first use Bernstein's conditions to separate it in multiple independent program sub-segments. There are two advantages in doing that. First, these sub-segments could be executed in parallel. But, more important, we can decide for each sub-segment independently if its loop can be parallelized. To do that, we can use the rules determined previously in the section, which require that, in a sub-segment, a regular variable must only be read, not written, and an array must be accessed with indices such that an index is used in two distinct iterations for either at most one write, or only read access.

4 Detection with Metaprogramming

4.1 Expression Templates

We propose here to use ET to get the Abstract Syntax Tree (AST) of the program segment of a loop, first to identify the dependent sub-segments and possibly split the loop, and then, to identify the dependency between loop iterations and decide to make it parallel or not. We use common template metaprogramming techniques to analyse the AST at compile-time, run the decision algorithms, and finally produce or not a parallel code. Template metaprogramming, notably formalized through the notions of metafunctions [1] enables running full programs (C++ templates are considered Turing-Complete [18]) at compile-time.

Expression templates are a technique introduced by Veldhuizen [16] and Vandevoorde [14] to represent an expression as an object, using templates to build the type of this object. The first goal of expression templates was to tackle performance problems that may occur with operator overloading: instead of performing the effective operation, an operator builds an object representing the intended operation, the purpose being to delay the evaluation of intermediate operations to avoid unnecessary temporary objects, and evaluate the expression at once [17]. When an expression is executed, we get an object that represents the structure of this expression by a recursive composition of types that models its AST: an expression is an operation on operands that are expressions [16].

With our library, ET are enabled when an object is marked as "operand" in an expression. For the detection of parallelism in loops, we focus on arrays that must be declared to be operands as shown in the listing 3 below.

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

Listing 3: C++ example of operand creation

The first line simply defines an array of integers that contains the values 1, 2, 3, 4, 5. The second line encapsulates the array as an operand, which will enable the ET mechanism for all the expressions that contain the operand. The `Operand` generic class takes two parameters: the underlying type and a unique identifier. The purpose of the unique identifier is to differentiate two operands statically (to avoid the aliasing of variables in ET [9]). Information on the underlying data could be used for that (like its memory address), but only dynamically as it is unknown at compile-time.

The operands support C++ arithmetic operations (thanks to operator overloading) and dedicated functions (meaning functions with operands as arguments, in order to make it extendable to potentially any kind of operation). These operations construct an expression template that the metaprogram will use to analyse the program segment structure at data access level. The result of any operation between instances of the `Operand` class is an instance of the `Expression` generic class. The `Expression` generic class handles the same operations as the `Operand` generic class, and represents a complete instruction. The listing 4 shows a simple expression.

```
1 int aData[] = {1, 2, 3, 4, 5},
2     bData[] = {5, 4, 3, 2, 1};
3 Operand<int[5], 1> a(aData);
4 Operand<int[5], 2> b(bData);
5 auto e = a + b;
```

Listing 4: C++ example of ET

In this example, `e` is an instance of the `Expression` generic class. Since C++11, the `auto` keyword lets the compiler deduce the type of a variable based on the value assigned to it during its definition. Thus, the deduced type of `e` is, in a simplified form, `Expression<OAddition, Operand<1>, Operand<2>>`, where `OAddition` is a type used to keep the information of which operation must be done between two operands. We can represent the instance `e` with the tree in figure 1.

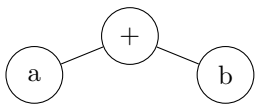


Figure 1: Tree generated from expression "a + b"

The underlying type of operands is not relevant to determine whether it is possible or not to execute it in parallel. The relevant information here is which variables are accessed, and how (read or write). The first information is provided by the unique identifier of the operand. The second one is deduced from the semantics of the operators.

In further examples, operand definition will be eluded in order to minimize code snippets. We will now consider a more complex example in listing 5. In C++, the comma

operator is overloadable, thus we can use it to build expression templates, whereas it is not possible with the semicolon punctuator.

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

Listing 5: 2^{nd} C++ example of ET

The tree in figure 2, generated from listing 5, can be explored at compile-time to determine for each instruction which variables are read and which are written. For example, in assignment operations, we know that left operands are written and right operands are read.

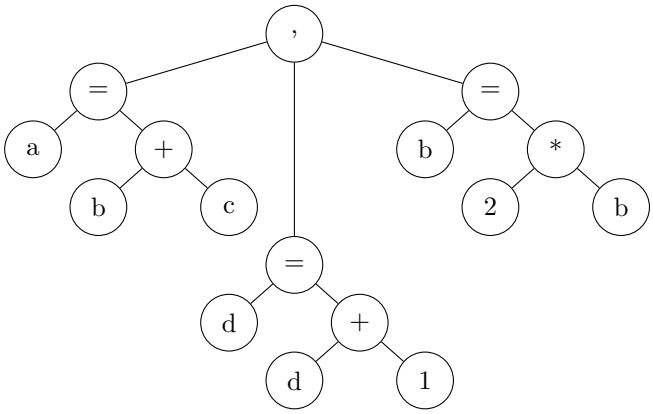


Figure 2: Tree generated from expression of listing 5

Eventually, we will see a last example that contains array accesses in listing 6. Note here that expressions concerning indices are also captured using ET: the `Iterator` generic class has the same function as the `Operand` generic class, which is to enable ET construction. Then, metaprogramming will allow for an array a to get its set of functions F_a.

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

Listing 6: C++ example of ET with arrays

With this last tree, in figure 3, generated from listing 6, we can execute the algorithm 1 presented in section 3 to split the program segment into independent sub-segments, and then over each sub-segment we can check condition (9) to decide whether it must be executed in parallel or not. For this specific tree, we will have two independent sub-segments, each one containing only one instruction.

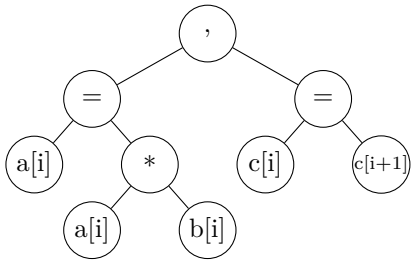


Figure 3: Tree generated from expression in listing 6

The first instruction (left branch of the tree) can be executed in parallel: two arrays are accessed, a and b. For each one, we have only one function on the iterators, the identity, so we have $|F_a| = 1$ and $|F_b| = 1$.

However, the second instruction (right branch of the tree) accesses the array c with two distinct functions (the identity, and $i \mapsto i + 1$), so $|F_c| = 2$, plus we have $F_c^W \neq \emptyset$, so this instruction will be considered not parallelizable (in this case, the instruction really cannot be executed in parallel).

4.2 Working examples

This section presents examples using the `parallel_for` function of our library. This function acts as an index-based `for` loop: it will iterate from a begin index to an end index. The following examples will call the `parallel_for` function, whose prototype is shown in listing 7. We use `@n` to hide small parts of C++ code that are too technical to be presented here.

```
template<
  typename F,
  typename E = @1,
  typename = @2>
std::size_t parallel_for(
  std::size_t first,
  std::size_t last,
  F &&f
);
```

Listing 7: `parallel_for` prototype

This function takes three arguments, a first index and a past last index that indicate the range of the loop, and a function (precisely a "callable", i.e., a function pointer, a functor or a lambda) returning the expression template of the program segment inside the loop to execute. `F` is the type of the function, it is used by expression `@1` to infer the type `E` of the expression returned by the function. The last and unnamed type is necessary to block the function instantiation if `E` is an invalid expression type thanks to expression `@2` (which relies on SFINAE [11]).

The current implementation of `parallel_for` is to split the given expression into independent sub-expressions, then to test each sub-expression for parallelization. If a sub-expression is parallelizable, it produces a code using OpenMP to run it in parallel; else it produces a sequential code. The example in listing 8 demonstrates the automatic parallelization of a simple loop with one instruction.

```
1 /* a, b, c are operands
2   based on arrays of N values */
3 parallel_for(0, N,
4   [&](Iterator i) {
5     return(
6       a[i] = b[i] * c[i]
7     );
8   }
9 );
```

Listing 8: Basic `parallel_for` example

Listing 8 uses a C++ lambda for the last argument of `parallel_for` to simplify the code. This lambda captures declared operands (the `[&]` part). Its argument (`i`) that represents the value of the index at a given iteration is in fact an iterator (as defined in section 4.1) for arrays in the lambda body.

In this example, the expression given to `parallel_for` contains only one parallelizable instruction. The generated source code will be as in listing 9.

```
1 #pragma omp parallel for
2 for(int i = 0; i < N; ++i)
3   a[i] = b[i] * c[i];
```

Listing 9: Code generated by listing 8

The example in listing 10 demonstrates the same point with a more complex program segment, the example of listing 1 from section 3. It is quite similar to listing 8. In line 2 of the program segment, `ct<1>` is used to make value 1 a compile-time data that can be manipulated by metaprograms.

```
1 /* a, b, c, d are operands
2   based on arrays of N values */
3 parallel_for(0, N,
4   [&](Iterator i) {
5     return(
6       a[i] = a[i] * b[i],
7       c[i] = c[i+ct<1>] + d[i],
8       b[i] = b[i] + 1,
9       d[i] = c[i]
10    );
11  }
12 );
```

Listing 10: Advanced `parallel_for` example

As explained before, `parallel_for` function first groups dependent instructions together to get independent program segments, then, for each program segment, checks condition (9). The result is the compile-time generated source code shown in listing 11.

```
1 #pragma omp parallel for
2 for(int i = 0; i < N; ++i) {
3   a[i] = a[i] * b[i];
4   b[i] = b[i] + 1;
5 }
6 // not parallelized:
7 for(int i = 0; i < N; ++i) {
8   c[i] = c[i+1] + d[i];
9   d[i] = c[i];
10 }
```

Listing 11: Code generated by listing 10

In the code snippets above (listings 3 and 11) we have presented the functioning of our library-based solution. The codes explain the syntax used and also give the code generated by the metaprogram. The latter splits the loop into independent chunks of code and has setup parallel directives for code sections identified as parallelizable.

5 Performance results

While the TMP techniques used in our solution operate mainly at compile-time, there is still some additional code that is generated for execution at run-time. We present here performance tests to evaluate the execution time overhead induced with our solution.

Two scenarios are considered: (i) a loop that is not parallelizable (no instruction is parallelizable); (ii) a loop that is partially parallelizable (with two independent instruction sets, one is parallelizable, the other is not, like example in listing 10). For each scenario, the instructions of the loop are arithmetical operations on arrays of integers. The tests have been performed on an Intel Xeon CPU E7-8890 v3, using g++ 6.3.0 with the optimization `O2` flag activated.

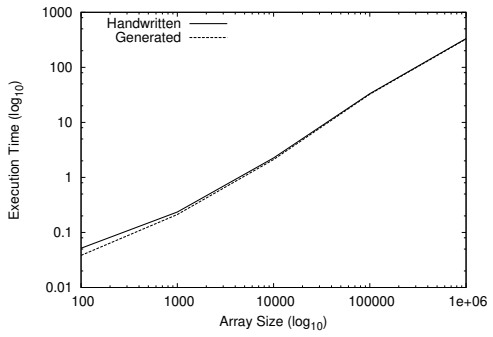
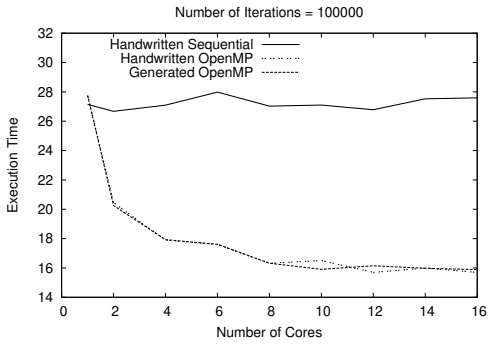
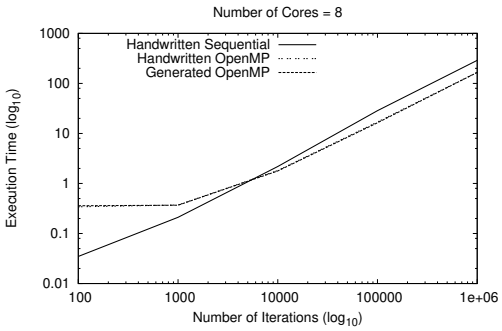


Figure 4: TMP overhead in a purely sequential situation



(a) impact of the number of cores



(b) impact of the number of iterations

Figure 5: TMP overhead in a mixed parallel-sequential situation

Figure 4 shows scenario (i) where the number of iterations of the loop varies form 10^2 to 10^6. It compares the execution times of the handwritten sequential code and the sequential code generated by our solution. No significant overhead is noticeable. Slight differences could be explained by the possible reordering of instructions induced by our solution.

Figures 5a and 5b show scenario (ii) where the number of physical cores used to parallelize the loop varies from 2 to 16 and the range of the loop varies from 10^2 to 10^6 iterations, respectively. In this scenario, the loop must be split into a sequential and a parallel loop. This code written by hand is compared with the equivalent code generated by our solution. Again, no significant overhead is noticeable between the parallel versions.

In those tests, OpenMP is used for parallelization, but any other library can be used, as long as the user provides the recipe for a parallel loop. Our solution has also been tested with standard threads of C++ and provides similar results.

6 Conclusion

In this paper, we show how the detection of parallelization can be achieved within a high-level programming language. The proof of concept is designed with C++ Template Metaprogramming (TMP), that gives us the hand on the underlying compiler to enable code re-writing in a syntax-checked way. We have used Expression Templates (ET) to get relevant information for parallelization on the code sequence of a loop. TMP enables the execution of algorithms at compile-time, in our case to analyse the information retrieved with ET, and we have presented in this article the conditions needed to decide which parts of a code sequence in a loop are parallelizable. After a theoretical explanation, we have presented an algorithm able to decide whether program segments are parallelizable or not. This was followed by the presentation of our C++ library-based solution, with code snippets showing the use of TMP to detect data dependencies between instructions at compile-time. Our library decides whether to parallelize a loop before generating a parallel code (after splitting the loop if necessary). One can choose to generate the parallel code with any regular parallel library (OpenMP, MPI, POSIX Threads, ...). This proof of concept enables the automatic writing of parallel programs even if we do not have a C++ parallel compiler at our disposal. Indeed, with TMP, we are able to lay out code that the programming system executes to generate new code that implements the desired parallelism at instruction level. Performance tests show that our proposal with TMP induces a negligible execution time overhead.

References

[1] David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. 5. printing. The C++ In-Depth Series. Boston, Mass: Addison-Wesley, 2004. ISBN: 978-0-321-22725-6 (cit. on pp. 1, 4).

[2] Ishfaq Ahmad, Yu-Kwong Kwok, Min-You Wu, and Wei Shu. "Automatic Parallelization and Scheduling of Programs on Multiprocessors Using CASCH". In: *Proceedings of the International Conference on Parallel Processing*. ICPP '97. Washington, DC, USA: IEEE Computer Society, 1997, pp. 288–291. ISBN: 978-0-8186-8108-0. DOI: 10.1109/ICPP.1997.622657 (cit. on p. 2).

[3] Amit Barve, Sneha Khomane, Bhagyashree Kulkarni, Shubhangi Katare, and Sonali Ghadage. "A Serial to Parallel C++ Code Converter for Multi-Core Machines". In: *2016 International Conference on ICT in Business Industry Government (ICTBIG)*. 2016, pp. 1–5. ISBN: 978-1-5090-5515-9. DOI: 10.1109/ICTBIG.2016.7892700 (cit. on p. 2).

[4] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. "Legion: Expressing Locality and Independence with Logical Regions". In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC '12. Los Alamitos, CA, USA: IEEE Computer Soci-

ety Press, 2012, 66:1–66:11. ISBN: 978-1-4673-0804-5 (cit. on p. 2).

[5] A. J. Bernstein. "Analysis of Programs for Parallel Processing". In: *IEEE Transactions on Electronic Computers* EC-15.5 (Oct. 1966), pp. 757–763. ISSN: 0367-7508. DOI: 10.1109/PGEC.1966.264565 (cit. on p. 2).

[6] William Blume, Rudolf Eigenmann, Keith Faigin, John Grout, Jay Hoeflinger, David Padua, Paul Petersen, William Pottenger, Lawrence Rauchwerger, Peng Tu, and Stephen Weatherford. "Effective Automatic Parallelization with Polaris". In: *International Journal of Parallel Programming* (1995) (cit. on p. 2).

[7] Daniel Cordes, Peter Marwedel, and Arindam Mallik. "Automatic Parallelization of Embedded Software Using Hierarchical Task Graphs and Integer Linear Programming". In: *International Conference on Hardware Software Codesign*. 2010, pp. 267–276 (cit. on p. 2).

[8] Michael Frumkin, Michelle Hribar, Haoqiang Jin, Abdul Waheed, and Jerry Yan. "A Comparison of Automatic Parallelization Tools/Compilers on the SGI Origin 2000". In: *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*. SC '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 1–22. ISBN: 978-0-89791-984-5 (cit. on p. 2).

[9] Jochen Härdtlein, Alexander Linke, and Christoph Pflaum. "Fast Expression Templates: Object-Oriented High Performance Computing". In: *Lecture Notes in Computer Science*. Springer-Verlag. 2005, pp. 1055–1063 (cit. on p. 4).

[10] David S. Hollman, Janine C. Bennett, Hemanth Kolla, Jonathan Lifflander, Nicole Slattengren, and Jeremiah Wilke. "Metaprogramming-Enabled Parallel Execution of Apparently Sequential C++ Code". In: *Proceedings of the Second International Workshop on Extreme Scale Programming Models and Middleware*. ESPM2. Piscataway, NJ, USA: IEEE Press, 2016, pp. 24–31. ISBN: 978-1-5090-3858-9. DOI: 10.1109/ESPM2.2016.8 (cit. on p. 2).

[11] Jaakko Järvi, Jeremiah Willcock, and Andrew Lumsdaine. "Concept-Controlled Polymorphism". In: *Proceedings of the 2nd International Conference on Generative Programming and Component Engineering*. GPCE '03. New York, NY, USA: Springer-Verlag New York, Inc., 2003, pp. 228–244. ISBN: 978-3-540-20102-1 (cit. on p. 5).

[12] Ulrich Kremer, Heinz-J Bast, Michael Gerndt, and Hans P. Zima. "Advanced Tools and Techniques for Automatic Parallelization". In: *Parallel Computing* 7.3 (Sept. 1988), pp. 387–393. ISSN: 0167-8191. DOI: 10.1016/0167-8191(88)90057-9 (cit. on p. 2).

[13] Antoine Tran Tan, Joel Falcou, Daniel Etiemble, and Hartmut Kaiser. "Automatic Task-Based Code Generation for High Performance Domain Specific Embedded Language". In: *Int. J. Parallel Program.* 44.3 (June 2016), pp. 449–465. ISSN: 0885-7458. DOI: 10.1007/s10766-015-0354-9 (cit. on p. 2).

[14] David Vandevoorde and Nicolai M. Josuttis. *C++ Templates: The Complete Guide*. 12. printing. Boston, Mass.: Addison-Wesley, 2010. ISBN: 978-0-201-73484-3 (cit. on p. 4).

[15] H. Vandierendonck, S. Rul, and K. De Bosschere. "The Paralax Infrastructure: Automatic Parallelization with a Helping Hand". In: *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. Sept. 2010, pp. 389–399 (cit. on p. 2).

[16] Todd Veldhuizen. "Expression Templates". In: *C++ Report* 7 (1995), pp. 26–31 (cit. on pp. 1, 4).

[17] Todd L. Veldhuizen. "Arrays in Blitz++". In: *Lecture Notes in Computer Science*. Vol. 1505. Springer-Verlag, 1998, pp. 223–230 (cit. on p. 4).

[18] Todd L. Veldhuizen. *C++ Templates Are Turing Complete*. Tech. rep. Indiana University Computer Science, 2003 (cit. on p. 4).

[19] Michael Wolfe and Utpal Banerjee. "Data Dependence and Its Application to Parallel Processing". In: *International Journal of Parallel Programming* 16.2 (Apr. 1987), pp. 137–178. ISSN: 0885-7458. DOI: 10.1007/BF01379099 (cit. on p. 2).

[20] Hans P Zima, Heinz-J Bast, and Michael Gerndt. "SUPERB: A Tool for Semi-Automatic MIMD/SIMD Parallelization". In: *Parallel Computing* 6.1 (Jan. 1988), pp. 1–18. ISSN: 0167-8191. DOI: 10.1016/0167-8191(88)90002-6 (cit. on p. 2).