Automatic Code Generation of Distributed Parallel Tasks

Nelson Lossing, Corinne Ancourt, François Irigoin

Automatic Code Generation of Distributed Parallel Tasks

Nelson Lossing
MINES ParisTech,
PSL Research University
nelson.lossing@mines-paristech.fr

Corinne Ancourt
MINES ParisTech,
PSL Research University
corinne.ancourt@mines-paristech.fr

Francois Irigoin
MINES ParisTech,
PSL Research University
francois.irigoin@mines-paristech.fr

***Abstract*—With the advent of clustered systems, more and more parallel computing is required. However a lot of programming skills is needed to write a parallel codes, especially when you want to benefit from the various parallel architectural resources, with heterogeneous units and complex memory organizations. We present in this paper a method that generates automatically, step by step, a task-parallel distributed code from a sequential program. It has been implemented in an existing source-to-source compiler PIPS. Our approach provides two main advantages 1) all the program transformations are simple and applied on source code, thus are visible by the user, 2) a proof of correctness of the parallelization process can be made. This ensures that we end up with a correct distributed-task program for distributed-memory machines. To our knowledge, it is the first tool that automatically generates a distributed code for task parallelization.**

I. Introduction

Nowadays, modern computers have more and more processors. It became cheaper and easier for computer manufacturers to multiply cores inside chips than to accelerate the processor itself [1]. So developers have to write parallel programs if they want to take full advantage of these new architectures. But parallel programming requires a lot of skills.

It is hard and depends on the target architecture. For instance, shared memory architectures require the programmer to take care of data races and race condition [2] to ensure code correctness, and to detect and avoid false sharing [3] to achieve good performance. Distributed memory architectures lead to data consistency issues for each executed process and often lead to communication bottleneck problems. Accelerator devices, like GPGPU, must execute exactly the same code with the minimum of divergent control flows and must do enough computation to compensate the data transfers between host and accelerators.

Several parallelization paradigms exist. We focus on two main paradigms that are used for both shared and distributed memory. The first one is *loop parallelization*. In *loop parallelization*, the issue is to minimize dependencies between each iteration of the loop nest. For this purpose, many transformations can be applied 1) on loop body variables like *privatization* or *induction variable substitution*, or 2) on loop itself like *loop fission/fusion*, *loop interchange* or 3) by fully rescheduling the loop iteration order. *Loop parallelization* issues are well known and many techniques have been proposed to automatically parallelize loop nests [4]. Tools like Cetus [5][6], Pluto+ [7][8], PPCG [9][10], XFOR [11][12] or dSTEP [13] handle this case of problems for shared or distributed memory architectures. Even some production compilers, like gcc [14], icc [15] or clang [16], make a few automatic optimizations on loops such as vectorization, strip-mining, etc. The second paradigm is the *task parallelization*. In fact, *loop parallelization* can be considered as a special case of *task parallelization*. *Task parallelization* needs to minimize the dependencies between tasks. *Control flow graph* and *data dependence* analyses are used to make this parallelization. Many task scheduling tools exist [17][18], but they are totally black boxes and produce object codes directly. There is no way to check that the executed code is correct as it could be possible from the sources of parallel codes. These task parallel codes use standard library or language like OpenMP [19], TBB [20] or Cilk+ [21] for share memory, or PVM [22][23] or MPI [24] for distributed memory.

Compilers are organized as sets of passes used to map the high-level constructs onto simple target machine features and run-time calls. Most of the current compiler technology targets sequential cores and parallel, shared or distributed processors, are handled via run-time calls. We propose new compiler passes to address distributed targets like sequential ones with few run-time calls. These passes focus to make *task parallelization*. The valid schedule is built by one task scheduler. Because we aim at performance, we assume a static mapping. Indeed, in a distributed memory context, a dynamic mapping would imply a master/slaves mechanism where the master is the scheduler. This latter becomes a bottleneck since it has to send and receive the data for and from each slave but also to send the task to be executed by each of them. Whereas a static mapping can easily allow a peer-to-peer communication between each process. The mapping can also be performed manually if the developer wants to use his/her own mapping. Our new distributed code is generated by a source-to-source compiler, step-by-step. It is useful for the user to know how his/her code is generated, and correctness proofs can be done after each transformation step. The code is generated in MPI which is the most popular library for distributed code.

The paper is organized as follows. Section II introduces the related work on automatic generation of distributed parallel codes. The tool and analyses that we use are presented in Section III. Section IV describes the successive transfor-

mations that we apply to generate the distributed task parallel code from the sequential source. Section V details our experimental results. Finally, the conclusion and some possible future work are presented in Section VI.

II. Related Work

Only few tools generate a distributed parallel code, and very few perform task parallelization distribution. Some generate a distributed code from sequential code only for loop nests such as dSTEP [13], Pluto+ [25] or DMCG [26] and others expect a parallel description as input and target only specific architecture machines such as Hypertool [27], PYRROS [28] or CellSs [29]. In our knowledge, there is no tool that automatically generates a distributed task parallel code from a sequential code.

dSTEP, distributed STEP [30], and DMCG generate directive-based MPI codes and use customized directives to mark the parallel loops to work on. In dSTEP case, two pragmas are introduced: `dstep distribute` is associated with an array to indicate how it is distributed along the process; `dstep gridify` is associated with a loop and describes in which direction, ie. loop iterator, and which schedule it will be processed. DMCG permits to indicate the parallel loops and the partitionings that are applied. In Pluto+, the parts of the program to be parallelized is encapsulated within `scop` and `endscop` directives. Pluto+ reschedules loop iteration using a polyhedron model in order to improve data locality, then generates the distributed code. In these three tools, customized functions, based on MPI, are introduced to make the communications.

Hypertool and PYRROS need a direct acyclic graph (DAG) as input and generate parallel code for specific distributed architectures. For instance, PYRROS generates code for nCube or Intel iPSC/2 machine architectures. PYRROS uses a Dominant Sequence Algorithm (DSC) to automatically schedules and maps tasks. In CellSs, Cell Superscalar framework, annotation on sequential code is done to describe the dependency between the tasks. CellSs generates a source code that can only be compiled for Cell architecture.

III. Context

In this section, we present the tool used to implement our code transformations. The most important analyses and our choices for the code generation are also introduced.

A. PIPS, a Source-to-Source Compiler

We use an existing source-to-source compiler for the code generation, PIPS [31][32]. It offers a wide set of analyses and transformations over Fortran and C codes. Furthermore, PIPS uses an integer polyhedral abstraction to represent the domains of the program variables, which turns out very effective in the parallelization context.

PIPS contains an automatic task scheduler based on the BDSC algorithm [33], which is an improvement of DSC. The BDSC output can be an input of our automatic distribution of sequential code. This scheduler generates a static schedule with a static mapping and this static mapping is not an issue, in case of distributed task parallelization, since we want to achieve some performances. Moreover, to have a static mapping, we consider that the number of processes that execute the parallel code is numerically known in advance.

Among PIPS analyses and transformations, our compilation process uses *data dependence* with *convex array regions* and *dead-code elimination*. They are presented in the following subsection and again in Subsection IV-C.

B. Data Dependencies and Convex Array Regions

One key analysis in parallelization is *data dependence*. Three types of data dependence exist: flow (Read after Write), anti (Write after Read) and output (Write after Write) dependence. To generate distributed code, the important one is flow dependence, because it determines when a communication is needed. For anti and output dependencies, since each process has its own memory space, no conflict occur inside the process execution. Contrary to the case of shared memory, due to the usage of the same memory space, synchronization might be required to avoid data race.

In PIPS, the traditional *data dependence graph*, constructed with the previous dependencies, is not required to show the dependencies. A polyhedral analysis, called *convex array regions* [34], is available to show the set of array elements that are read or written by a statement. This includes also read and written scalar variables. By extension, PIPS gives *out regions* which describe array regions that are written by a statement and used in the continuation. So *out region* depict the Write part of a flow dependence, the part of an array or a scalar variable that need to be communicated for a following Read. The *convex array regions* use a polyhedral representation. Over-approximations of non-polyhedral set are made in some cases. For instance, if two non-consecutive elements of an array are used to make a computation, the *read convex array region* is the convex hull of the two elements instead of just the two elements. This case does not occur often and when it occurs, the analyses are always correct since they are conservative.

C. Code Generation

The code is generated in MPI, a well known library for distributed parallelization. Moreover, MPI is implemented for many platforms and architectures.

The generated code is the most simple possible. A knowledgeable MPI user can easily review the generated code and modify it if desired. The code is generated only for fix number of processors. For different number of processors, many different codes can be generated.

IV. Compilation Process

Our compilation process is composed of four big steps, described in Figure 1. These big steps are themselves broken up into different passes. These passes are as simple as possible to be informative for the user and provable. After each pass, the code is still correct and can be executed.

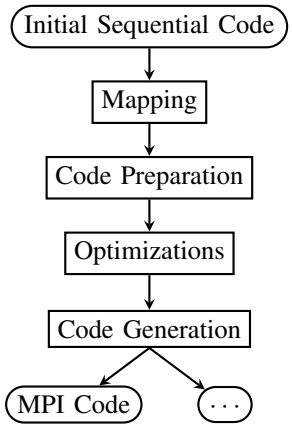

Figure 1: Compilation Process Schema

To illustrate the compilation process, the corner detection algorithm by Harris and Stephens [35] is used. This algorithm applies Sobel and Gauss filters, a matrix multiplication and a coarsity matrix. Listing 1 shows the algorithm code, and Figure 2 highlights dependencies between tasks.

Listing 1: Harris&Stephens algorithm

```
int main(int argc, char **argv) {
  /* Variable declaration/allocation. */
  double in[6000][5900]; /*...*/
    /* ...Initialize array... */
  SobelX(6000, 5900, Gx, in);
  SobelY(6000, 5900, Gy, in);
  Multiply(6000, 5900, Ixx, Gx, Gx);
  Multiply(6000, 5900, Iyy, Gy, Gy);
  Multiply(6000, 5900, Ixy, Gx, Gy);
  Gauss(6000, 5900, Sxx, Ixx);
  Gauss(6000, 5900, Syy, Iyy);
  Gauss(6000, 5900, Sxy, Ixy);
  Coarsity(6000, 5900, out, Sxx, Syy, Sxy);
  /* ...Print result... */
  return 0;
}
```

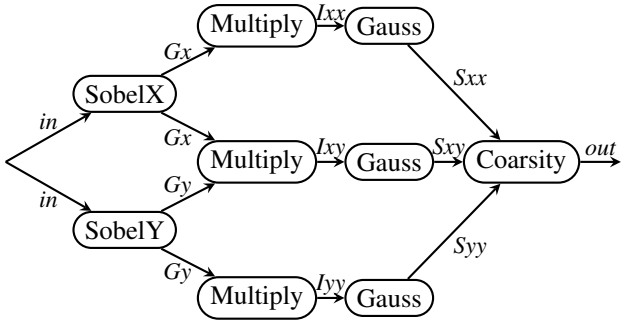

Figure 2: Harris&Stephens task dependence graph

A. Mapping

The task can be mapped automatically from the results of a task scheduler, like BDSC in PIPS. In addition, it can be specified manually with some `pragma`. For this purpose, the `pragma distributed` is added with the parameter `on_cluster` to indicate on which virtual process the task must be executed. This `pragma` has to be only at the outermost scope of a function. So no task is declared inside a test case or a loop. In case of loops, a *loop unrolling* or *loop splitting* can be applied to treat each iteration or a group of iterations as a task.

Unlike some other tools, such as OpenMP 4.0 [36] or OmpSs [37][38], that need the explicit dependencies between each task in order to perform an efficient scheduling, our mapping does not need this information from the user. PIPS automatically computes this information that is also taken into account in the *array region* analyses.

The mapping selected for the corner detection algorithm is presented in Listing 2 with three processes. Process `0` makes the initializations and prints the results. The three processes only work simultaneously during the multiplication and the application of Gauss filters. Two different processes execute Sobel filter.

Listing 2: Mapping for Harris&Stephens algorithm

```
int main(int argc, char **argv) {
  /* Variable declaration/allocation. */
  double in[6000][5900]; /*...*/
#pragma distributed on_cluster = 0
  { /* ...Initialize array... */ }
#pragma distributed on_cluster = 0
  { SobelX(6000, 5900, Gx, in); }
#pragma distributed on_cluster = 1
  { SobelY(6000, 5900, Gy, in); }
#pragma distributed on_cluster = 0
  { Multiply(6000, 5900, Ixx, Gx, Gx); }
#pragma distributed on_cluster = 1
  { Multiply(6000, 5900, Iyy, Gy, Gy); }
#pragma distributed on_cluster = 2
  { Multiply(6000, 5900, Ixy, Gx, Gy); }
#pragma distributed on_cluster = 0
  { Gauss(6000, 5900, Sxx, Ixx); }
#pragma distributed on_cluster = 1
  { Gauss(6000, 5900, Syy, Iyy); }
#pragma distributed on_cluster = 2
  { Gauss(6000, 5900, Sxy, Ixy); }
#pragma distributed on_cluster = 0
  { Coarsity(6000, 5900, out, Sxx, Syy, Sxy); }
#pragma distributed on_cluster = 0
  { /* ...Print result... */ }
  return 0;
}
```

B. Code Preparation

The Preparation Step formats the code to be parallelized for our code generation. For this purpose, we chose to allocate to each process a dedicated copy for each variable of the initial code. The representation of a communication is done by a copy statement to the appropriated variable of the target process. For instance, a communication of variable `x` from process `i` to process `j` is represented with a copy of `x_i` to `x_j`, ie. `x_j=x_i`. This step is carried out by a sequence of different passes.

The first pass declares the dedicated variables in each process. For instance, for `n` processes and for variable `x`, the dedicated variables `x_0` to `x_{n-1}` are declared by this pass.

The second pass generates the copy statements that represent the communications. These copies are useful to guaranty the memory coherency. They are generated at the end of each task. This can be done in three different ways. The first

one is the naive approach. It copies every variable for each process. The memory consistency is easily obtained this way. Some optimizations must be done afterward to remove the unnecessary copies. The second one considers the variables that are written and only generate a copy for them for each process. It is what traditional data flow analysis can provide. But in the case of an array, when one cell of the array is written, all the array elements are copied on the other processes. In such case, an optimization is also required to remove extra copies of the array elements. The third one uses the *array regions* available in PIPS, more precisely the *out regions* resulting from each task. It has two advantages, one is to copy only the part of the array that is really written, up to an over approximation that may occur. The other advantage is that it only considers the variables or array elements that are written in the task and that are used later in the execution. So it reduces the number of unnecessary copies generated.

The third pass substitutes the initial variables by the dedicated variables in the computations of the task . It is done by *alpha renaming* or *variable substitution* of the variables used in the task with the appropriated local variables. Since the mapping is static, this information is given by the parameter `on_cluster` of our `pragma`.

The last pass consists in removing the initial variables with a *clean declaration* transformation. Since the third pass removes all the use of the original variables, their declarations are naturally removed from the code. Moreover an additional *identity-copy elimination* transformation eliminates statements with template `x=x`.

Once the Preparation Step is completed, our Code Generation Step can proceed, possibly without Optimizations Step. But the resulting code remains inefficient because too many unnecessary copies, representing communications, are still present, even with the restrictive strategy of copy generation used during the second pass. So additional optimizations are presented in the following section to improve the code.

The formatting step of Harris&Stephens algorithm is illustrated in Listing 3. New variables for each process are declared, and initial variables have been removed. At the end of each task, each variable that is written and useful later in the program execution has been copied.

C. Optimizations

The optimizations focus on reducing the memory footprint and the amount of communications generated during the Code Preparation Step. Three different optimizations are proposed in this section.

The first optimization removes useless communications that may occur when a process receives a piece of data but does not use it. When copies are generated, in Pass 2 of Code Preparation, the procedure is systematic for all the processes. Assuming we use *out regions* to generate the copy statements, two main cases can occur (see Table I). In the first case, we consider a task on process `P` computing the value of `x`, then none of the following tasks on process `Q` do use `x`, but one task on process `R`, different from `Q`, uses it to make some

Listing 3: Formatting code for Harris&Stephens algorithm

```
int main(int argc, char **argv) {
  /* Variable declaration/allocation. */
  double __in_0[6000][5900], __in_1[6000][5900],
__in_2[6000][5900]; /* ... */
#pragma distributed on_cluster = 0
  {
    /* ...Initialize array... */
    {
      int PHI1, PHI2;
      for (PHI1 = 0; PHI1 <= 5999; PHI1 += 1)
        for (PHI2 = 0; PHI2 <= 5899; PHI2 += 1) {
          __in_1[PHI1][PHI2]=__in_0[PHI1][PHI2];
          __in_2[PHI1][PHI2]=__in_0[PHI1][PHI2];
        }
    }
  }
#pragma distributed on_cluster = 0
  {
    SobelX(6000, 5900, __Gx_0, __in_0);
    /* Copy __Gx_1=__Gx_0 */
    /* Copy __Gx_2=__Gx_0 */
  }
#pragma distributed on_cluster = 1
  {
    SobelY(6000, 5900, __Gy_1, __in_1);
    /* Copy __Gy_0=__Gy_1 */
    /* Copy __Gx_2=__Gy_1 */
  }
#pragma distributed on_cluster = 0
  {
    MultiplY(6000, 5900, __Ixx_0, __Gx_0, __Gx_0);
    /* Copy __Ixx_1=__Ixx_0 */
    /* Copy __Ixx_2=__Ixx_0 */
  }
#pragma distributed on_cluster = 1
  {
    MultiplY(6000, 5900, __Iyy_1, __Gy_1, __Gy_1);
    /* Copy __Iyy_0=__Iyy_1 */
    /* Copy __Iyy_2=__Iyy_1 */
  }
#pragma distributed on_cluster = 2
  {
    MultiplY(6000, 5900, __Ixy_2, __Gx_2, __Gy_2);
    /* Copy __Ixy_0=__Ixy_2 */
    /* Copy __Ixy_1=__Ixy_2 */
  }
#pragma distributed on_cluster = 0
  {
    Gauss(6000, 5900, __Sxx_0, __Ixx_0);
    /* Copy __Sxx_1=__Sxx_0 */
    /* Copy __Sxx_2=__Sxx_0 */
  }
#pragma distributed on_cluster = 1
  {
    Gauss(6000, 5900, __Syy_1, __Iyy_1);
    /* Copy __Syy_0=__Syy_1 */
    /* Copy __Syy_2=__Syy_1 */
  }
#pragma distributed on_cluster = 2
  {
    Gauss(6000, 5900, __Sxy_2, __Ixy_2);
    /* Copy __Sxy_0=__Sxy_2 */
    /* Copy __Sxy_1=__Sxy_2 */
  }
#pragma distributed on_cluster = 0
  {
    Coarsity(6000, 5900, __out_0, __Sxx_0, __Syy_0, __Sxy_0);
    /* Copy __out_1=__out_0 */
    /* Copy __out_2=__out_0 */
  }
#pragma distributed on_cluster = 0
  { /* ...Print result... */ }
  return 0;
}
```

	case 1	case 2
task on P	write x_P ~~x_Q = x_P~~ x_R = x_P	write x_P x_Q = x_P ~~x_R = x_P~~
task on Q Q≠P	...	read x_Q write x_Q x_P = x_Q x_R = x_Q
task on R R≠{P,Q}	read x_R	read x_R

Table I: Removing useless copy

computation. The systematic procedure generates a copy of x on process P for each process including Q. This latter is useless and has to be eliminated. In the second case, a task on process P computes the value of x, then the following tasks on process Q use and modify it, finally another task on process R different from P and Q uses it. In such example, the copy of x from P to R will be out of date when R will be executed because Q will have been updated it. To remove these useless copies, a *dead-code elimination* pass [39] is applied. This classical optimization removes unused statements. In the first case, since each variable is associated to a process, if the tasks executed on this process do not need the variable. No reads of this variable are present, *dead-code elimination* removes all the affectations of this variable. Similarly, in the second case, the *dead-code elimination* pass considers that the first affectation is useless and has to be removed.

The second optimization is an improvement of the *dead-code elimination* transformation. Because *dead-code elimination* only works on statements and variables, if only part of an array is useful to the following tasks of the program, when all its elements are computed, no *dead-code elimination* is performed. For this purpose, we designed a new optimization called *dead-iteration elimination*. This new transformation follows the same rules as *dead-code elimination*. It considers as useless a write that is never read or a write that is rewritten. Moreover, it works on the iterations of the loop nests by taking into account the information on the *regions* of the arrays that are really useful later in the program execution. The *out regions* associated to the loop are used to perform this optimization. For instance, a loop iterates from n0 to m0 and modifies each cell of a from n0 to m0. If its *out regions* are only the part of array a from n1 to m1, such that n0<n1<m1<m0, thus our *dead-iteration elimination* updates the loop iterator to iterate only from n1 to m1.

The last optimization improves the memory footprint that is needed for the execution of the program. This optimization is an *array resizing* transformation [40]. Unlike Fortran, which supports array declaration from any starting point or without any explicit size, arrays in C must always start at 0 and must have a size that is known numerically, or symbolically in C99 standard. For computing the minimal size necessary to an array in a program, we have to know which parts of the array are really used in the program. The *read/write regions* give us this information. The computation of the new array size is deduced from the difference between the upper and lower bounds of the used array elements. But all the uses, read or write, of this array have to make a shift corresponding to the lower bound found. Depending if some tricky optimizations are present inside the code, the *array resizing* can be done for all dimensions of the array or only for the first one, in C code. For instance, addresses can be directly generated in an optimized way that disrupt the array resizing.

To conclude this section, thanks to the first two optimizations, we succeed in reducing considerably the number of static copies present inside the generated code. Since these copies would become communications and because communications are most of the time a bottleneck in distributed codes, our generated code is more efficient. Additionally, the last optimization reduces the memory space needed for the execution of the code. In the case of big data, benchmark with large data sets, incompatible with a single node, can be executed. Once the previous optimizations are applied, the distributed parallel code is generated, as the next section described.

The result of the optimizations performed on Harris&Stephens algorithm are presented in Listing 4. They mainly improve communication. For instance, in variable is only declared for processes 0 and 1, and its initialization is only copied from process 0 to 1. The results of Sobel filters are only sent to process 2 from process 0 and 1, since process 0 only needs Gx that it computes itself, same for process 1 with Gy, when 2 needs Gx and Gy for its computations. The final result in out is no longer copied to processes 1 and 2 because they do not need it. At the end, only five copy statements remained in this optimized code.

Listing 4: Optimized code for Harris&Stephens algorithm

```
int main(int argc, char **argv) {
  /* Variable declaration/allocation. */
  double __in_0[6000][5900], __in_1[6000][5900]; /* ... */
#pragma distributed on_cluster = 0
  {
    /* ...Initialize array... */
    {
      int PHI1, PHI2;
      for (PHI1 = 0; PHI1 <= 5999; PHI1 += 1)
        for (PHI2 = 0; PHI2 <= 5899; PHI2 += 1) {
          __in_1[PHI1][PHI2]=__in_0[PHI1][PHI2];
        }
    }
  }
#pragma distributed on_cluster = 0
  {
    SobelX(6000, 5900, __Gx_0, __in_0);
    /* Copy __Gx_2=__Gx_0 */
  }
#pragma distributed on_cluster = 1
  {
    SobelY(6000, 5900, __Gy_1, __in_1);
    /* Copy __Gx_2=__Gy_1 */
  }
#pragma distributed on_cluster = 0
  { MultiplY(6000, 5900, __Ixx_0, __Gx_0, __Gx_0); }
#pragma distributed on_cluster = 1
  { MultiplY(6000, 5900, __Iyy_1, __Gy_1, __Gy_1); }
#pragma distributed on_cluster = 2
  { MultiplY(6000, 5900, __Ixy_2, __Gx_2, __Gy_2); }
#pragma distributed on_cluster = 0
  { Gauss(6000, 5900, __Sxx_0, __Ixx_0); }
#pragma distributed on_cluster = 1
  {
    Gauss(6000, 5900, __Syy_1, __Iyy_1);
    /* Copy __Syy_0=__Syy_1 */
```

```
  }
#pragma distributed on_cluster = 2
  {
    Gauss(6000, 5900, __Sxy_2, __Ixy_2);
    /* Copy __Sxy_0=__Sxy_2 */
  }
#pragma distributed on_cluster = 0
  { Coarsity(6000, 5900, __out_0, __Sxx_0, __Syy_0, __Sxy_0); }
#pragma distributed on_cluster = 0
  { /* ...Print result... */ }
  return 0;
}
```

D. Code Generation

This last step performs the generation of the distributed code in two passes. The first one translates the formatted sequential code into a distributed code. The second one applies an optimization to the code, that might be performed during the previous translation, but is done afterwards to keep the translation as simple as possible.

The first thing we do for generating the distributed code is to add a parallel context. For MPI, we add 1) call to `MPI_Init` that is required for MPI execution; 2) call to `MPI_Comm_rank` to know which process is actually running the code, and to associate the tasks to each corresponding processes; and 3) call to `MPI_Comm_size` to check that we really have the required minimum processes to run the code. After this, the translation of the sequential code can begin. The transformation is done task by task. Since the computations and the copies, which represent communications, are independent, the computations can be associated to a process without any trouble. For each copy, two statements are generated: a send with `MPI_Send` and a receive with `MPI_Recv`. The send statement is associated to the same process as the related task, and it sends the value of the *rhs*, *right hand side*, of the copy statement. The receive statement receives the value on the *lhs*, *left hand side*, of the copy statement. Since each variable is associated to a unique process, with the *lhs*, the receive statement can be associated to the proper process. When the copy is inside a loop, this loop is replicated on each process. This translation is done for all the copies of the tasks, and for each task of the program. At the end of the generated code, before the return statement, a call to `MPI_Finalize`, required for all MPI program, is added.

In order to improve the performance of the distributed code execution, a final optimization is applied: the aggregation of the communications for arrays. This *communication aggregation* is related to send and receive statements present inside a loop and communicating array elements. The array elements to be communicated must be adjacent. Otherwise some other more complex optimizations could be performed, but they are more or less efficient depending of the MPI implementation, so we do not presented them here. Under the assumption of array elements adjacency, a loop nest related to communications is replaced by a unique corresponding communication, send for a send and receive for a receive. Instead of performing the communication element-by-element, this new communication instruction communicates the pack of initial elements together. Since during the generation of the distributed code, the loops related to communications are strictly the same on the receiver and sender sides, the transformation occurs on both sides and no incoherent communication can appear. This optimization can also be applied on existing MPI code that programmer have written.

This last step finalized our automatic generation of a task distributed parallel code. Moreover this code is a source code readable by any programmer who has experience in MPI.

The generated code of Harris&Stephens algorithm using MPI library is presented in Listing 5.

Listing 5: MPI distributed code for Harris&Stephens algorithm

```
int main(int argc, char **argv) {
  MPI_Status status0;
  int size0, rank0;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size0);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank0);
  /* ...Check number of process running... */
  /* Variable declaration/allocation. */
  double __in_0[6000][5900], __in_1[6000][5900]; /* ... */
  if (rank0 == 0) {
    /* ...Initialize array... */
    init_array(6000, 5900, __in_0);
    MPI_Send(&__in_0[0][0], 6000 * 5900, MPI_DOUBLE, 1, 0,
             MPI_COMM_WORLD);
  }
  if (rank0 == 1) {
    MPI_Recv(&__in_1[0][0], 6000 * 5900, MPI_DOUBLE, 0, 0,
             MPI_COMM_WORLD, &status0);
  }
  if (rank0 == 0) {
    SobelX(6000, 5900, __Gx_0, __in_0);
    MPI_Send(&__Gx_0[0][0], 6000 * 5900, MPI_DOUBLE, 2, 0,
             MPI_COMM_WORLD);
  }
  if (rank0 == 2) {
    MPI_Recv(&__Gx_2[0][0], 6000 * 5900, MPI_DOUBLE, 0, 0,
             MPI_COMM_WORLD, &status0);
  }
  /* ... */
  if (rank0 == 0)
    CoarsitY(6000, 5900, __out_0, __Sxx_0, __Syy_0, __Sxy_0);
  if (rank0 == 0)
    { /* ...Print result... */ }
  MPI_Finalize();
  return 0;
}
```

V. Experiments and Results

The experimental results of our method of automatic code generation are given in this section. Our hardware configuration and the selected benchmarks are first described. Then our experimental results are presented and discussed.

A. Hardware Configuration

The experiments are executed on a cluster of eight nodes each with a Intel(R) Xeon(R) CPU X5450 @ 3.00GHz processor and 16 Go RAM. The inter-node communications use a one gigabit Ethernet. All the clusters run Linux 2.6.32 64-bits. The MPI implementation is Open MPI 1.6.2 [41]. All codes are compiled with gcc 4.4.7 and optimization option '-O3' has been used. For the experiments, in order to fully take into account the communications through the different nodes, only one core runs on each node.

B. Benchmark

To evaluate the performance of our method, we performed the experiments on some linear C-BLAS kernels: `gemm`, `gemver`, `gesummv`, `symm`, `syr2k`, `syrk` and `trmm`. For the latter, we used the Polybench v4.2 implementation [42]. Polybench benchmarks include the infrastructure to compute the program execution time without the initialization step of arrays, and print the results to enable the verification of computations. For the BLAS, we increased the problem size proposed in Polybench, to obtain interesting and representative problems for distributed machines. We used sizes of 3840 for mono-dimensional arrays and 3840x3000, 3840x4000 or 3000x4000 for bi-dimensional arrays, all in doubles. For a good estimate of the execution time, each test is run five times, the lowest and fastest times are removed and the average of the remaining three execution times is given.

The target codes are generated for 2, 4 and 8 processes for the BLAS kernels. Since BLAS kernels are composed of `for` loops, a simple approach has been used to generate the tasks: Loops are split by the number of target processes and each new loop is assigned to a process.

C. Results

The execution times are summarized in Table II, and their corresponding speed-ups are illustrated in Figure 3.

`gemm` and `gesummv` are close to a perfect scaling with the increase in the number of used processes. This good performance is obtained because the outermost loop is fully parallel. Moreover, the accesses to the arrays follow the outermost loop iterations on the first dimension, that enables to fully take advantage of the cache.

`syrk` and `syr2k` also scale with the number of processors with a good speed-up, and `trmm` has a little speed up. Like `gemm` and `gesummv`, their outermost loop iterations are independent. But the arrays are accessed both along the outer and the inner loop iterators. So cache reloads are more frequent. Moreover in `trmm` case, these accesses are done on the array that is updated.

An execution onto a distributed architecture for `gemver` and `symm` do not improve their performance. This is due to the fact that the generated code is sequential. Therefore, communications only add execution time. In `symm`, dependencies exist between each iteration of the outermost loop, so splitting the loop is not enough to generate several independent tasks. A rescheduling or a new tiling of the loop nest must be performed to introduce an independent outermost loop iteration before mapping the different tasks. For `gemver`, four successive loops are present in the computation kernel. These four loops have flow dependences, so they cannot be considered as independent tasks being able to be executed simultaneously. Moreover, in `gemver`, the reads and writes on array elements in the successive loop nests are not done in the same loop iterator order. For instance, the first loop computes new values for `A[i][j]`, but the second loop accesses elements `A[j][i]`, so all loops have to communicate their results to each other.

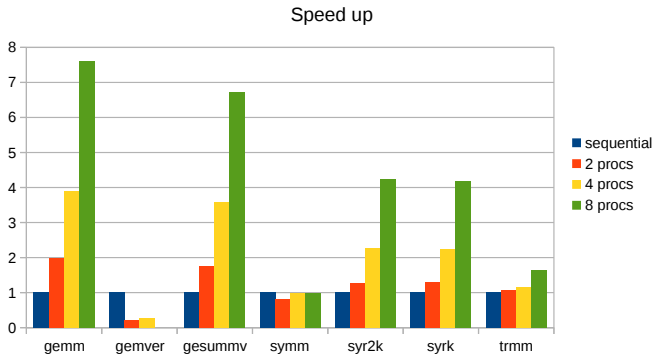

Figure 3: Performances of the automatic parallelization

benchmarks	sequential	2 processors	4 processors	8 processors
gemm	104,4091	52,5408	26,7061	13,7405
gemver	0,8917	4,4416	3,2143	NA
gesummv	0,0644	0,0368	0,0180	0,0096
symm	182,6694	223,2030	186,5294	188,0440
syr2k	157,8145	123,9313	69,3526	37,1894
syrk	84,8091	64,9078	37,6302	20,2551
trmm	459,5931	432,4614	397,6002	278,3207

Table II: Execution time of the benchmarks (in s)

D. Limitations

These experiments show that our compilation process generates automatically a correct parallel distributed code for task parallelization from a sequential code. For good candidates suitable to the distribution, the generated code scales with the number of used processors, do not forget that only one core is used by node to force the communication inter node.

But, some performance problems remain. These issues come mainly from an inefficient naive distributed mapping applied to the initial code. Some improvements on this mapping must be done to show better results on all the benchmarks, for instance by adding a rescheduling of the loops.

VI. Conclusion

This article presents a compilation process to automatically generate a parallel task code executable onto distributed memory machines from a sequential code. This compilation process is based on a small-step approach. It has many advantages: 1) all the transformations are as simple as possible in order to enable a proof of their correctness, that are under processes, 2) the source of the transformation results represent an important informative support for the user, 3) each transformation pass can be replaced by another equivalent one if a better solution exists, for instance the initial mapping pass can be replaced, 4) new transformations can be added during the optimization step or after the generation of the distributed code. The modularity allows an easy testing of new optimizations (additional source-to-source polyhedral optimizations, etc.) or different options of the target codes (asynchronous communications, other libraries, etc.). The experimental results present some benchmarks with good execution times, scaling with the number of processors.

In the near future, a new `pragma` to process `for` loops is considered. Its goal is to automatically map the intended blocks of iterations into tasks. Associated with this `pragma`, loop rescheduling can be performed to ensure that the iterations of the outermost loop are not dependent. This can be done by applying PIPS transformations or by using another tool like Pluto+. Another optimization is to group the statements executed on the same process into a single test case, then to outline them into separate functions, one for each process. This last optimization would take full advantages of the *array resizing* optimization.

In the future, the generation of asynchronous communications is planed, or a transformation from synchronous to asynchronous communications.

REFERENCES

[1] H. Sutter, "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software," *Dr. Dobb's Journal*, vol. 30, no. 3, pp. 202–210, 2005.

[2] R. H. B. Netzer and B. P. Miller, "What Are Race Conditions?: Some Issues and Formalizations," *ACM Lett. Program. Lang. Syst.*, vol. 1, no. 1, pp. 74–88, Mar. 1992.

[3] W. J. Bolosky and M. L. Scott, "False Sharing and Its Effect on Shared Memory Performance," in *USENIX Systems on USENIX Experiences with Distributed and Multiprocessor Systems - Volume 4*, ser. Sedms'93. Berkeley, CA, USA: USENIX Association, 1993, pp. 3–3.

[4] M. Wolfe, *Techniques for Improving the Inherent Parallelism in Programs*, ser. UILU-ENG / 17: UILU-ENG. Department of Computer Science, University of Illinois at Urbana-Champaign, 1978.

[5] Cetus A Source-to-Source Compiler Infrastructure for C Programs. [Online]. Available: http://cetus.ecn.purdue.edu/

[6] C. Dave, H. Bae, S.-J. Min, S. Lee, R. Eigenmann, and S. Midkiff, "Cetus: A Source-to-Source Compiler Infrastructure for Multicores," *Computer*, vol. 42, no. 12, pp. 36–42, Dec. 2009.

[7] Bondhugula, Uday. PLUTO - An automatic parallelizer and locality optimizer for affine loop nests. [Online]. Available: http://pluto-compiler.sourceforge.net/

[8] U. Bondhugula, "Compiling Affine Loop Nests for Distributed-memory Parallel Architectures," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. New York, NY, USA: ACM, 2013.

[9] Polly Labs. PPCG - Polyhedral Parallel Code Generator. [Online]. Available: https://www.openhub.net/p/ppcg

[10] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gómez, C. Tenllado, and F. Catthoor, "Polyhedral Parallel Code Generation for CUDA," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 54:1–54:23, Jan. 2013.

[11] XFOR, A Programming Structure to Ease the Formulation of Efficient Loop Optimizations & A Polyhedral Language. [Online]. Available: http://xfor.gforge.inria.fr/

[12] I. Fassi and P. Clauss, "XFOR: Filling the Gap between Automatic Loop Optimization and Peak Performance," in *14th International Symposium on Parallel and Distributed Computing*, IEEE, Ed., Limassol, Cyprus, Jun. 2015.

[13] R. HABEL, "High Performance Programming for Hybrid Architectures," Theses, Ecole Nationale Supérieure des Mines de Paris, Nov. 2014.

[14] Free Software Foundation, Inc. GCC, the GNU Compiler Collection. [Online]. Available: https://gcc.gnu.org/

[15] Intel®. Intel® C++ Compiler. [Online]. Available: https://software.intel.com/en-us/c-compilers/ipsxe/

[16] LLVM Team. clang: a C language family frontend for LLVM. [Online]. Available: http://clang.llvm.org/

[17] Y.-K. Kwok and I. Ahmad, "Benchmarking and Comparison of the Task Graph Scheduling Algorithms," *J. Parallel Distrib. Comput.*, vol. 59, no. 3, pp. 381–422, Dec. 1999.

[18] S. Jin, G. Schiavone, and D. Turgut, "A Performance Study of Multiprocessor Task Scheduling Algorithms," *J. Supercomput.*, vol. 43, no. 1, pp. 77–97, Jan. 2008.

[19] OpenMP ARB. OpenMP. [Online]. Available: http://openmp.org/wp/

[20] Intel®. Threading Building Blocks. [Online]. Available: https://www.threadingbuildingblocks.org/

[21] ——. Cilk™ Plus. [Online]. Available: https://www.cilkplus.org/

[22] Oak Ridge National Laboratory. PVM, Parallel Virtual Machine. [Online]. Available: http://www.csm.ornl.gov/pvm/

[23] V. S. Sunderam, "PVM: A Framework for Parallel Distributed Computing," *Concurrency: Pract. Exper.*, vol. 2, no. 4, pp. 315–339, Nov. 1990.

[24] Message Passing Interface Forum. MPI, Message Passing Interface. [Online]. Available: https://www.mpi-forum.org/

[25] U. Bondhugula, A. Acharya, and A. Cohen, "The Pluto+ Algorithm: A Practical Approach for Parallelization and Locality Optimization of Affine Loop Nests," *ACM Trans. Program. Lang. Syst.*, vol. 38, no. 3, pp. 12:1–12:32, Apr. 2016.

[26] C.-T. Yang and K.-C. Lai, "A Directive-based MPI Code Generator for Linux PC Clusters," *J. Supercomput.*, vol. 50, no. 2, pp. 177–207, Nov. 2009.

[27] M. Y. Wu and D. D. Gajski, "Hypertool: A Programming Aid for Message-Passing Systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 1, no. 3, pp. 330–343, Jul. 1990.

[28] T. Yang and A. Gerasoulis, "PYRROS: Static Task Scheduling and Code Generation for Message Passing Multiprocessors," in *Proceedings of the 6th International Conference on Supercomputing*, ser. ICS '92. New York, NY, USA: ACM, 1992, pp. 428–437.

[29] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta, "CellSs: A Programming Model for the Cell BE Architecture," in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, ser. SC '06. New York, NY, USA: ACM, 2006.

[30] D. Millot, A. Muller, C. Parrot, and F. Silber-Chaussumier, "From OpenMP to MPI: first experiments of the STEP source-to-source transformation tool." in *Parallel Computing: From Multicores and GPU's to Petascale, Proceedings of the conference ParCo*, 2009, pp. 669–676.

[31] MINES-ParisTech, "PIPS," http://pips4u.org, 1989–2016, open source, under GPLv3.

[32] M. Amini, C. Ancourt, F. Coelho, B. Creusillet, S. Guelton, F. Irigoin, P. Jouvelot, R. Keryell, and P. Villalon, "PIPS Is not (just) Polyhedral Software Adding GPU Code Generation in PIPS," in *First International Workshop on Polyhedral Compilation Techniques (IMPACT 2011) in conjonction with CGO 2011*, Chamonix, France, Apr. 2011, 6 pages.

[33] D. Khaldi, P. Jouvelot, and C. Ancourt, "Parallelizing with BDSC, a Resource-constrained Scheduling Algorithm for Shared and Distributed Memory Systems," *Parallel Comput.*, vol. 41, no. C, pp. 66–89, Jan. 2015.

[34] B. Creusillet and F. Irigoin, "Interprocedural Array Region Analyses," *Int. J. Parallel Program.*, vol. 24, no. 6, pp. 513–546, Dec. 1996.

[35] C. Harris and M. Stephens, "A combined corner and edge detector," in *In Proc. of Fourth Alvey Vision Conference*, 1988, pp. 147–151.

[36] M. Klemm and C. Terboven, "Full Throttle: OpenMP 4.0," *The Parallel Universe Magazine*, vol. 16, pp. 6–16, 2013.

[37] BSC Programming Models group. The OmpSs Programming Model. [Online]. Available: https://pm.bsc.es/ompss

[38] M. Tillenius, E. Larsson, R. M. Badia, and X. Martorell, "Resource-Aware Task Scheduling," *ACM Trans. Embed. Comput. Syst.*, vol. 14, no. 1, pp. 5:1–5:25, Jan. 2015.

[39] A. V. Aho, R. Sethi, J. Ullman, and M. S. Lam, *Compilers: Principles, Techniques, & Tools*, 2nd ed. Boston: Pearson/Addison Wesley, 2006.

[40] C. Ancourt and T. V. N. Nguyen, "Array Resizing for Scientific Code Debugging, Maintenance and Reuse," in *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, ser. PASTE '01. New York, NY, USA: ACM, 2001, pp. 32–37.

[41] Open MPI. [Online]. Available: https://www.open-mpi.org/

[42] PolyBench/C 4.2. [Online]. Available: https://sourceforge.net/projects/polybench/