INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Parallelization of finite element codes with automatic placement of communications

Laurent Hascoët

N° 3646

Mars 1999

THÈME 4

ISSN 0249-6399

Parallelization of finite element codes with automatic placement of communications

Laurent Hascoët

Thème 4 — Simulation et optimisation
de systèmes complexes
Projet Sinus

Rapport de recherche n° 3646 — Mars 1999 — 25 pages

Abstract: We present a tool dedicated to automatic SPMD parallelization of iterative mesh-based computations, and its application to existing codes. The tool automatically places communication statements into the program, to manage the necessary updates between overlapping parts of the partitioned mesh. It is parameterizable with a description of the type of overlapping between sub-meshes. We present an application of this tool to two-dimensional and three-dimensional Navier-Stokes flow solvers. Performance results are given.

Key-words: parallelization, SPMD, program analysis, program transformation, mesh partition, finite elements, Navier-Stokes

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, B.P. 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : 04 92 38 77 77 - International : +33 4 92 38 77 77 — Fax : 04 92 38 77 65 - International : +33 4 92 38 77 65

Parallélisation de codes éléments finis avec un placement automatique des communications

Résumé : Nous présentons un outil destiné à la parallélisation SPMD automatique de programmes de résolution itératifs basés sur des maillages. Nous présentons la conception de l'outil et son utilisation. Cet outil insère dans le programme les communications nécessaires pour tenir à jour les valeurs aux frontières des sous-maillages. Il est paramétrable grâce à une description du type de recouvrement choisi entre les sous-maillages. Nous présentons l'application de cet outil à des solveurs Navier-Stokes à deux et trois dimensions, et nous donnons des mesures de performances.

Mots-clés : parallélisation, SPMD, analyse de programme, transformation de programme, partition de maillage, éléments finis, Navier-Stokes

Contents

1 Introduction **3**

2 The *Exp2D*, *Imp2D*, and *NSC3DM* Navier-Stokes solvers **5**

3 The SPMD parallel execution model **6**
3.1 Mesh Partitioning 7
3.2 SPMD Program Generation 8

4 Automatic placement of communications **9**

5 From an algorithm to a real tool **14**
5.1 Alignment of data and loops 14
5.2 Solution filtering 15
5.3 Multi-Procedural issues 16

6 Application to the Navier-Stokes solvers **17**
6.1 Use of the tool 17
6.2 Performances 18

7 Related works **20**

8 Conclusion **23**

1 Introduction

For many application programs, parallelization is an appealing, but delicate, way to improve performance. This is especially true in scientific computing. In this domain, one category of programs deserves special attention. These are the iterative computations on unstructured meshes. They are very widespread, and consume a significant part of the computational resources. These programs are natural candidates for parallelization.

There exist many parallelization techniques. This is due, to a large extent, to the numerous machine architectures. As the art evolves, a matching emerges between the classes of programs and the parallelization techniques.

For example, computations on unstructured meshes are now generally parallelized through a geometric partitioning of the mesh itself. The parallelized program uses the SPMD (*Single Program Multiple Data*) model. In this model, each processor of a distributed memory computer runs the same program on one particular portion of the original mesh. Communications are required when a value must travel from a sub-mesh to another. Usually, this happens only for values located on the border of sub-meshes. It is common practice to duplicate a certain amount of these "border" values on the neighboring sub-meshes. This allows us to defer communications until all the duplicated values are out of date and need to be updated again.

The adaption of a program to this SPMD parallelization strategy requires two main operations: *Mesh Partitioning* and *SPMD Program Generation*. These will be discussed in detail in section 3. Writing the SPMD program by hand is a delicate task. It requires knowledge of the application to find the places where values travel between sub-meshes. It also requires knowledge of the overlapping between sub-meshes, to correctly update duplicated values. Therefore, we propose a tool that generates the SPMD program mechanically. It is important to note that this is a specialized parallelization tool: this tool makes sense only for computations on unstructured meshes. This is our deliberate choice, to trade generality for efficiency on a precise domain. Our justifications are that:

- computations on unstructured meshes are a very frequent kind of programs. Generic parallelizers behave poorly due to the intensive use of indirection arrays.
- the knowledge of how meshes are structured, and how programs behave on meshes, allows an automatic choice of optimal communications insertion. This would be out of reach of a generic parallelizer, because it cannot infer this knowledge from the program.

In this paper, we present a tool for automatic generation of a SPMD program for unstructured meshes. This tool determines the optimal locations of communication calls, to give best performance of the parallelized program. This tool is based on the method and algorithms presented in [4], that we shall summarize in section 4. We also show here, how this tool was used to parallelize

existing 2D and 3D Navier-Stokes solvers. The rest of the paper is organized as follows: Section 2 briefly introduces the original sequential Navier-Stokes solvers. Section 3 presents precisely the SPMD model of parallel execution. Section 4 describes the principle of our automatic SPMD program generation. Section 5 explains the specific problems related to the "real" programs, and how they were solved. Section 6 presents the application of our tool to the Navier-Stokes solver, and the generated SPMD program. Section 7 compares the present method with related techniques.

2 The *Exp2D*, *Imp2D*, and *NSC3DM* Navier-Stokes solvers

Our application examples are two-dimensional and three-dimensional flow solvers, that we shall refer to as *Exp2D*, *Imp2D*, and *NSC3DM*. They solve the compressible Navier-Stokes equations using a mixed finite element/finite volume method, designed on unstructured triangular meshes. Steady-state solutions are obtained using a pseudo-transient approach which is based on a linearized formulation. For *Exp2D*, this formulation is *explicit*, whereas for *Imp2D* and *NSC3DM*, the formulation is *implicit*. Therefore in *Imp2D* and *NSC3DM*, each time step requires the solution of a large sparse linear system. Approximate solutions of these systems are obtained by using Jacobi relaxations.

From a programming point of view, it turns out that the most frequent operations on the mesh are of the type known as *Gather-Scatter*. These operations consist of a loop on all the mesh elements of a given kind, say, triangles. For each triangle, the mesh structure gives a direct access to its three nodes. The operations on this triangle are decomposed in three phases:

1. **Gather:** some values attached to the three nodes are read.
2. **Compute:** these values are used, with others, to compute three partial results. Those are the contribution of the triangle to the total results of its nodes.
3. **Scatter:** each partial result is accumulated into the total result of the corresponding node.

At the end of the loop on triangles, each node will hold the correct final result, i.e. the accumulation of the partial results of all its neighboring triangles.

Table 1 gives a rough idea of the size of these codes. Comments are not counted in the number of lines.

Solver	*Exp2D*	*Imp2D*	*NSC3DM*
subroutines	13	18	34
lines	1500	1800	4800

Table 1: Sizes of application examples

3 The SPMD parallel execution model

As we saw in section 1, the SPMD parallel execution model requires two major operations: *Mesh Partitioning* and *SPMD Program Generation*. As shown on figure 1, these operations are independent. However, both require information about the pattern of duplicated elements (*overlapping pattern*). Also, we shall see that the particular SPMD program style used here, requires that the partitioner represents sub-meshes in a precise manner.

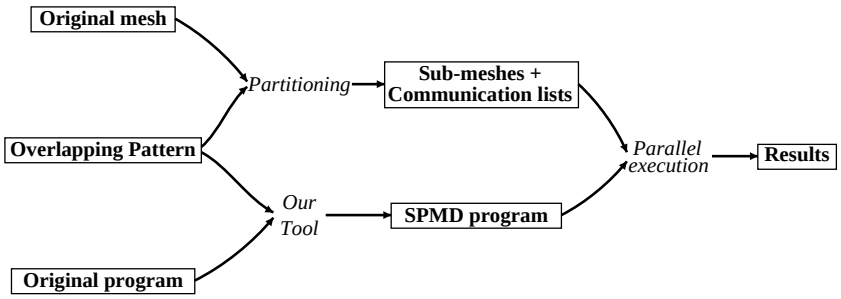

Figure 1: *SPMD parallelization process*

3.1 Mesh Partitioning

The initial mesh must be partitioned into as many sub-meshes as there are available processors. Any mesh partitioner can do that, provided it respects the constraints given below.

A variety of techniques, such as the recursive bisection method, are used to minimize the size of the interface between sub-meshes, therefore minimizing the need for expensive communications. But, whatever mesh partition we end up with, this will not affect the result of our SPMD program generator. Only, the resulting SPMD program will perform better on a better mesh partition. The partitioner is entirely in charge of dealing with *load balancing*. When one processor is over-used or under-used, or when *mesh refinement* changes the number of mesh elements in sub-meshes, the partition must be updated. This task belongs to the mesh partitioner. But the SPMD program itself need not be modified.

Following the decision of the user, the partitioner must create a certain number of overlapping sub-mesh elements. This will allow many communications to be gathered, yielding a dramatic reduction of communication overhead. This requires the partitioner to pre-compute and store the lists of overlapping mesh elements, between any two given sub-meshes. These lists are later used by the communication routines. It is worth comparing this approach with the classical "*inspector-executor*" paradigm. Anticipating on section 7, we remark that the present method amounts to leaving the "*inspector*" task to the mesh partitioner.

For our application, all there is to choose is the amount of overlapping between sub-meshes, that we call the *overlapping pattern*. In two dimensions, we chose to set a one-triangle wide overlapping zone, as shown on figure 2. Similarly in three dimensions, we set a onetetrahedron wide overlapping zone. This choice will strongly affect the placement of communications by our tool. Therefore, because of overlap and communication cost, this will affect the performances of the resulting SPMD program. In [3], one can find a discussion comparing various overlapping patterns. The best candidates seem to be:

1. One layer of overlapping triangles, with neighboring edges and nodes (*cf* figure 2) .
2. Only one layer of overlapping nodes (*cf* figure 4).

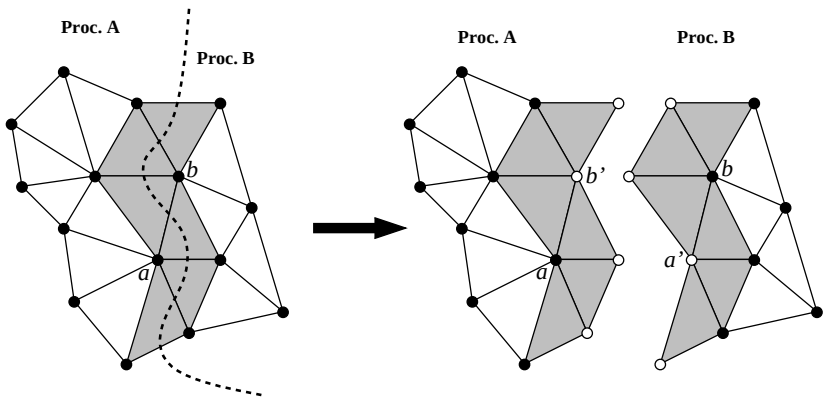

Figure 2: *Partitioning with one layer of overlapping triangles*

Both candidates are interesting. A wide overlapping zone allows us to defer communication more. However, one must be careful not to duplicate too many computations on overlapping elements, or redundant computations might outweigh communication overhead. This is especially important in three-dimensional meshes.

3.2 SPMD Program Generation

The original program must be transformed to deal with sub-meshes. The SPMD style requires that the same code runs on each processor, with different data. An elegant solution is to leave the program unchanged, except for communications. To achieve this, each sub-mesh must be organized in the same manner as the original mesh, with the same addressing scheme. This puts an extra requirement on the mesh partitioner, but ensures that the sub-meshes can be used by the SPMD program just like the original mesh was used by the original program. In particular, this avoids any extra indirection arrays used in the program.

The only transformations are communication insertions and overlap management. The overlap mechanism allows us to regroup all communications at a few selected locations in the program. Since there is an overlap, one must decide, for each program loop on mesh elements, whether it must also run on

overlapping elements or not. Since these decisions depend on the overlapping pattern, our SPMD program generator must accept it as a parameter. This will be shown in section 4.

We can show now on figure 2, how this overlap mechanism allows us to defer and gather communications. Consider node a, located on the frontier of sub-mesh A. Overlapping creates a duplicated node a' in sub-mesh B. Suppose that, just before a *gather-scatter* loop on triangles, the values on overlapping elements are coherent. Then for all triangles, the *gather* and *compute* phases are identical, and the same values are *scattered*. But nodes a' and b' have not received the same contributions as nodes a and b, just because they see fewer neighbors around them. Therefore, there must be one communication to update all duplicated nodes before any new *gather-scatter* loop. Let us call S_1 the set of possible locations for this communication.

In the case of *reduction* operations, such as a sum of all values held by the nodes, we can see that the values on a' and b' must not be taken into account, because the values on a and b are taken. Therefore a' and b' may be out of date. The result of the reduction is a partial sum held by each processor. Therefore, one communication must take place before any use of the sum. This communication will get all partial sums, compute the global sum, and finally send the result back to each processor. Let us call S_2 the set of possible locations for this communication.

When communication locations (such as S_1 and S_2) intersect, it is advantageous to regroup the two communications, thereby saving some overhead.

4 Automatic placement of communications

In this section, we shall describe the principle of our automatic generation of the SPMD program. It is derived from an algorithm that we described in [4]. For the practical application shown here, we adapted it to a new kind of sub-mesh overlapping. We also devised new methods to minimize interaction with the user, and to deal with multi-procedural programs. Without these methods, this algorithm was not really practicable.

The problem is to find the best locations for calls to communication routines, that will update the values held by overlapping mesh elements. Let us

formalize what was done manually. In section 3.2, we saw how, given a mesh partitioned with a given overlapping pattern, the values held by mesh elements evolved, from a state where overlapping values are coherent, to a state where they are incoherent, and return to a coherent state through a communication.

This suggests that there is a notion of *state* of the data, as well as *transitions* between these (few) states, and this forms a finite state automaton, that we call the *Overlap Automaton*. These states apply to the data that "travels" through the program's instructions, which we call the *flowing data*. Precisely, the *flowing data* is what travels along the *data flow graph*, composed of the classical three kinds of dependences: **True dependences** from variable write's to read's returning the value written; **Control dependences** from tests and loops to controlled operations; and **Value dependences** from operands to operation results.

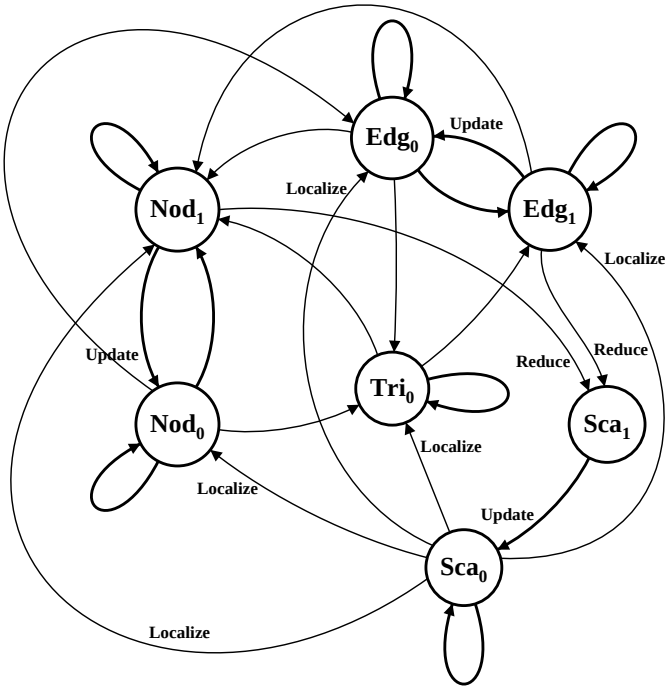

Figure 3: *Overlap Automaton for the overlapping pattern of figure 2.*

Figure 3 shows the Overlap Automaton for our chosen overlapping pattern of figure 2. It specifies how the state of the flowing data may evolve, for a given overlapping pattern. This sort of automata are created manually. A different overlapping pattern would require a different Overlap Automaton. In practice, there are few overlapping patterns used, so most of their overlap automata are already predefined in our tool. For example, another popular overlapping pattern is shown on figure 4, and figure 5 shows its Overlap Automaton. Similarly, we have defined an Overlap Automaton for 3D meshes, in

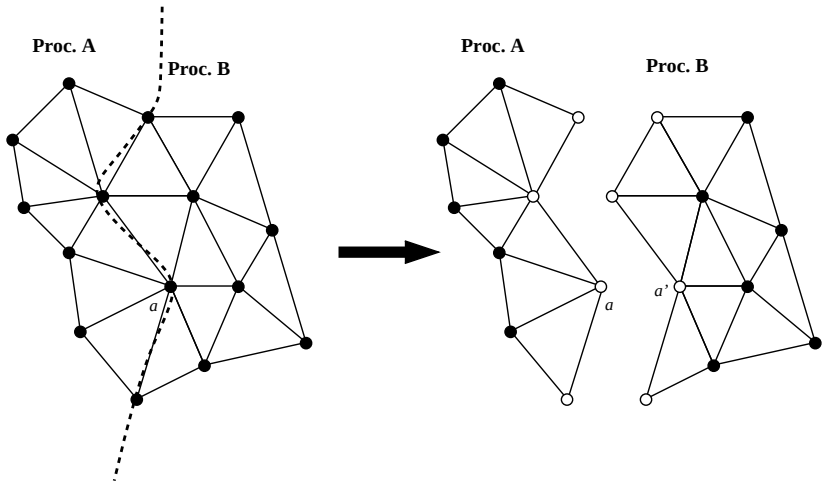

Figure 4: *Another overlapping pattern*

the case of a one tetrahedron wide overlap. This automaton was used for the *NSC3DM* application example. It is shown on figure 6.

Although they are currently hand-built, we feel it may be possible to *generate* these automata, specializing a higher-level automaton that relates all possible overlap situations. This is a subject for further research. To get a feeling of how the automaton on figure 3 was built, one may look at the cycle between states:

- $\mathbf{Nod}_0$: "distributed on mesh nodes, with coherent overlapping values"
- $\mathbf{Tri}_0$: "distributed on mesh triangles"

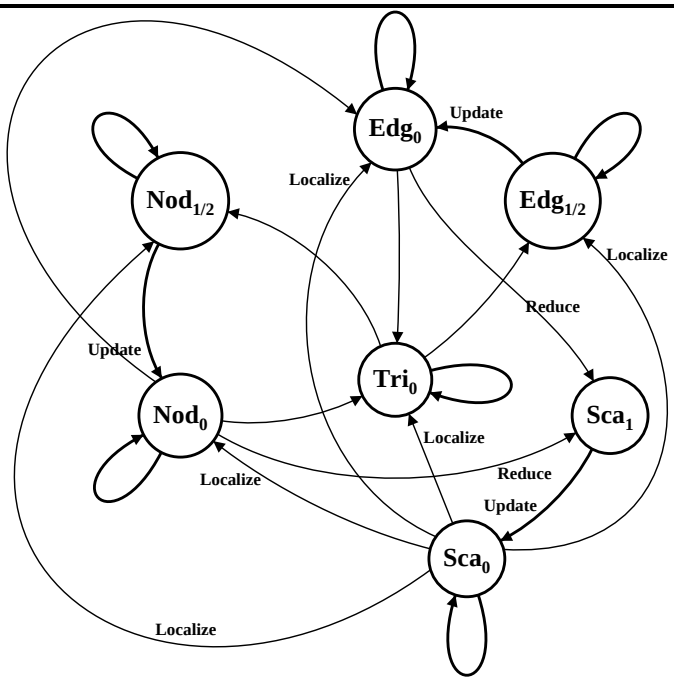

Figure 5: *Overlap Automaton for the overlapping pattern of figure 4.*

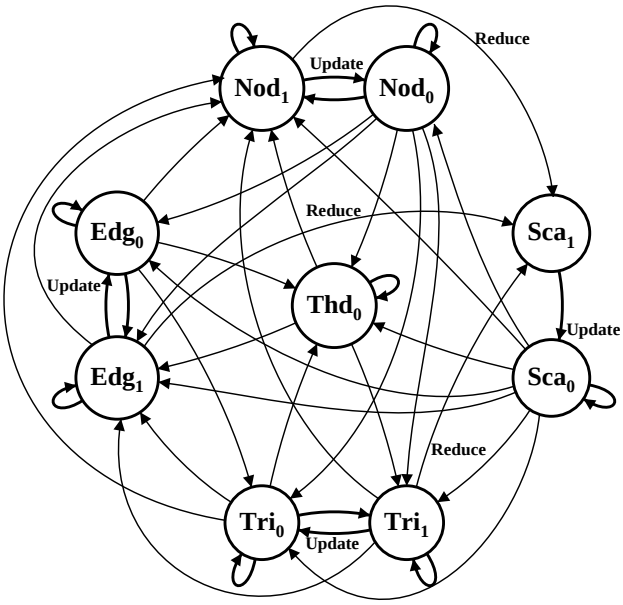

Figure 6: *Overlap Automaton for 3D meshes with overlap*

- $\mathbf{Nod}_1$: "distributed on mesh nodes, with incoherent (neglected) overlapping values"

This cycle corresponds to the evolution of the flowing data across a *gather-scatter* loop, as explained at the end of section 3.2. In the same manner, the evolution across a *reduction* phase, as described at the end of section 3.2, corresponds to the transitions between states

- $\mathbf{Nod}_1$: "distributed on mesh nodes, with incoherent (neglected) overlapping values"
- $\mathbf{Sca}_1$: "incoherent scalar value obtained by partial reduction on each processor"
- $\mathbf{Sca}_0$: "coherent scalar value after global reduction on all processors"

States $\mathbf{Edg}_0$ and $\mathbf{Edg}_0$ are analogous to $\mathbf{Nod}_0$ and $\mathbf{Nod}_1$, but for values distributed on mesh *edges*.

In other words, the above $\mathbf{Nod}_0$ $\mathbf{Tri}_0$ $\mathbf{Nod}_1$ cycle can be read as follows: "A loop on mesh *triangles* must gather values on *nodes* that are *up to date* on the overlap. This results in a scattered result that is *out of date* on the overlap, and must eventually be *updated*". Similarly, the $\mathbf{Nod}_1$ $\mathbf{Sca}_1$ $\mathbf{Sca}_0$ sequence can be read as follows: "A reduction on all *nodes* does not require that the values be *up to date* on the overlap. It will give a partial reduction result on each sub-mesh. There must be a global reduction to get the correct, usable result".

The SPMD program executes correctly if all evolutions of its flowing data are legal, i.e. correspond to existing transitions of the Overlap Automaton. Formally this requires a mapping M_n from the data-flow nodes to the Overlap Automaton states, and a mapping M_a from the data-flow arrows to the Overlap Automaton transitions, such that:

1. For every data-flow node N of the program's input, $M_n(N)$ equals its given initial state.
2. For every data-flow node N of the program's output, $M_n(N)$ equals its required result state.

3. For every data-flow arrow A in the program,

$$origin(M_a(A)) = M_n(origin(A))$$

$$destination(M_a(A)) = M_n(destination(A))$$

This mapping is computed by a simple sweep through the data-flow graph. Some care is necessary because the data-flow graph is generally cyclic. It is important to remark that some mapping choices can lead to impossibilities later in the sweep. Furthermore, the solution, when it exists, is not unique (*cf* section 5.2). Therefore, a backtracking mechanism is necessary.

For each mapping, the corresponding solution is found by application of the following rules:

1. For each data-dependence arrow, when it is mapped to a transition labeled as "**Update**", a communication must be inserted in the program between the beginning and the end of the dependence.

2. For each loop, when the control of the loop is mapped to one of states $\mathbf{Nod}_1$ or $\mathbf{Edg}_1$, then the loop must not iterate on duplicated mesh elements. Otherwise, it must iterate on all mesh elements, including the duplicates.

5 From an algorithm to a real tool

Real programs pose several additional problems that are not dealt with in the above method. The following sections discuss the main three problems, and their solutions in our tool.

5.1 Alignment of data and loops

Each SPMD parallelization session starts by asking the user about the alignment of data arrays and program loops on mesh elements. For each array and loop, the user must tell if it is aligned (distributed) on the mesh nodes, edges, triangles. Otherwise, the variable or loop is "replicated" on each processor.

Note that we don't question the user for parallel and non-parallel loops. The detection of parallelism is done automatically by the tool, for each loop declared as "distributed". If such a loop is found non-parallel, then an error is raised (*cf* section 6.1).

For a real, large program, this can be quite cumbersome. Therefore we apply simple propagation rules, so that the answer to one such "question" will solve many other questions.

- When an array index is the same as an enclosing loop index, the alignment of the loop is the same as the alignment of the array's dimension.
- For any pair of nested loops – even in two different subroutines – only one loop may be aligned on some mesh elements. Therefore the other loop must be replicated.
- Similarly, only one dimension of a given array can be aligned. The other dimensions must be replicated.

These rules apply to multi-procedural programs. For example, the alignment choice of a formal parameter (*Resp.* an array in `COMMON`) is the same as for the actual parameter (*Resp.* the same array in another routine).

Experience shows that these propagation rules are very efficient, and all questions for a large program can be answered in very few clicks. Usually less than ten answers are enough.

5.2 Solution filtering

We mentioned that many solutions may be found for a given routine, for example because the call to communication may be placed anywhere between the place of coherence loss and the following place that needs coherence. Also a loop may be chosen *not to run* on duplicated elements, therefore avoiding redundant computations, but possibly losing coherence. We want to filter these solutions, to provide the user with the best ones only, and let him choose among them.

There are mainly three (often conflicting) reasons why a particular choice of locations can be preferred:

1. it may avoid useless updates, *i.e.* insure that in any execution case, the data that is updated on the overlap was really out of date before.

2. it may delay updates, so that subsequent loops will execute only on the sub-meshes without their overlap.

3. it may regroup updates, therefore minimizing communication overhead.

To reflect this, two figures are sufficient to characterize the cost of one given solution:

1. The amount of communication, that depends on the number of variables that must be exchanged, and also on the communication overhead. This overhead depends on the number of times the SPMD program stops for communication. This expresses that it is preferable, when possible, to put many communications together at the same program line.

2. The amount of redundant computations, that depends on the number and size of all loops that were mapped to status $\mathbf{Nod}_0$ or $\mathbf{Edg}_0$.

This gives a partial order on the solutions, allowing the user to retain only the best solutions.

5.3 Multi-Procedural issues

When a routine calls another routine, it is generally too expensive to analyze them together as a single routine (*inlining*). They must be analyzed separately. In our case, this must be done from the bottom up in the program's call graph. For each solution for a particular subroutine, a summarized signature is computed. This signature specifies, for each input data, the overlap status that is required for this solution for this subroutine. The signature also specifies the overlap status of each data returned by the subroutine. Lastly, it also specifies the selected solution for each recursively called subroutine. Therefore, each solution of the called subroutine is an alternative choice in the solution of the calling routine.

Unfortunately, generally there is not *one best solution* for the called subroutine. So one must consider many alternatives, and this increases the search space for the calling routine.

However, some solutions may be filtered out, because they are certainly not as good as others (*cf* section 5.2), or because some communications are better placed in the calling routine. In particular, suppose a solution for the called subroutine requires a communication on some array, just at the beginning or at the end of the subroutine. This solution is then subsumed by the solution *without* the communication, and it is be the calling routine that inserts the communication, before or after the call, if necessary. This presents two advantages: it decreases the size of the search space, and it tends to place communications higher in the call tree, where they make more sense to the user.

Lastly, immediately after each subroutine is analyzed, the user must be given the opportunity to reject manually some of the solutions. On large programs, this will greatly reduce the search space when analyzing the calling routines.

6 Application to the Navier-Stokes solvers

The application of our tool to the *Exp2D* and *Imp2D* programs takes approximately 10 minutes on a Sun Ultra-1/170. Application to *NSC3DM* took roughly one hour. This time includes CPU time as well as interaction with the user. We believe that someone unfamiliar with these solvers would need much longer to transform the programs by hand. First, we shall describe the interactive process that led to the transformed programs. Then we shall present the performances of the generated programs.

6.1 Use of the tool

We implemented our tool as an additional module inside an existing parallelizer, called PARTITA. To begin with, the user opens a window showing the top routine for parallelization. Then the user designates the section to parallelize, with the mouse. This can be the whole program, but it was more appropriate here to select only the computation kernel, omitting configuration files and final result storage.

Then the user selects the Overlapping pattern of figure 2. This selects automatically the predefined Overlap Automaton of figure 3. After a short

while, the system prompts the user with questions about alignment of data and loops. For instance, in the case of *Imp2D*, answering five such questions, inside four subroutines, was enough to get all required answers. Fifteen answers were necessary for *NSC3DM*.

Then the research of a mapping starts, from the bottom up in the program's call tree. Each time a subroutine presents many solutions, the user is given the opportunity to reject unwanted solutions.

When a distributed loop contains dependences that may actually prevent its parallelization, a warning message is printed. This happened twice for *Imp2D*, because of the order of output messages: when finding error situations, *Imp2D* emits messages (*e.g.* *"Pressure too small"*). When parallelized, the program may print messages in a different order. This difference was judged acceptable.

Finally, six solutions were found for *Exp2D*, three solutions for *Imp2D*, and eight solutions for *NSC3DM*. For each solution, the tool gives an evaluation of the communication cost, and of the redundant computation cost. The user judged one solution best, and saved the generated program into a file.

6.2 Performances

Performance results are given for 64 bit arithmetic computations. In the following tables, N_p denotes the number of processes for the parallel execution; "Elapsed" denotes the total elapsed execution time and "CPU" denotes the total CPU time (taken as the maximum value over the local measures); the parallel speedup $S(N_p)$ is calculated using the elapsed execution times. Simulations have been performed on the following computing platforms :

- a `SGI Origin 2000` system* equipped with `Mips R10000/195 Mhz` processors with 4 Mb of cache memory. The `SGI` implementation of MPI has been used for the implementation of communication steps;
- a cluster of 12 Pentium Pro P6/200 Mhz running the Linux operating system (version 2.0.29). In this case the same code has been compiled using

*This system is located at the *Centre Charles Hermite* in Nancy.

the G77 gnu compiler with maximal optimization options. Communications are performed using MPICH (version 1.1). The cluster nodes are currently interconnected via a 100 Mbit/s FastEthernet switch.

In our first test case, we use the *Imp2D* Navier-Stokes solver to compute the external flow around a `NACA0012` airfoil at a freestream Mach number of 0.85, with a Reynolds number equal to 2000. The underlying triangular mesh contains 48792 vertices, 96896 triangles and 145688 edges. It was partitioned with the **MS3D** [8] tool. Parallel performance results are given in tables 2 and 3. The decreasing sizes of the sub-meshes resulting from the 8 and 16 sub-meshes decomposition are responsible for the superlinear speed-up observed on the `SGI Origin 2000`; on this platform the `R10000` processor possesses a 4 Mb cache memory, a rather large value that contributes to high computational rates.

N_p	Elapsed	CPU	% CPU	$S(N_p)$
1	2793 sec	2773 sec	99	1.0
4	815 sec	809 sec	99	3.5
8	314 sec	310 sec	99	8.9
16	147 sec	145 sec	99	19.0

Table 2: Parallel simulations on a `SGI Origin 2000` system
Laminar viscous flow around a `NACA0012` airfoil

N_p	Elapsed	CPU	% CPU	$S(N_p)$
1	6883 sec	6854 sec	99	1.0
4	2289 sec	2182 sec	95	3.0
8	1440 sec	1218 sec	85	4.8

Table 3: Parallel simulations on a `Pentium Pro` cluster
Laminar viscous flow around a `NACA0012` airfoil

In our second test case, we use the *NSC3DM* Navier-Stokes solver to compute the external flow around an `ONERA M6` wing at a freestream Mach number

of 0.84 and angle of incidence equal to 3.06°. The underlying tetrahedral mesh contains 15460 vertices, 80424 tetrahedra and 99891 edges. Parallel performance results are given in tables 4 and 5. Figure 7 shows the skin mesh and the iso-Mach lines of the steady state solution.

N_p	Elapsed	CPU	% CPU	$S(N_p)$
1	1318 sec	1305 sec	99	1.0
4	390 sec	387 sec	99	3.4
8	198 sec	196 sec	99	6.7

Table 4: Parallel simulations on a `SGI Origin 2000` system
Euler flow around an `ONERA M6` wing

N_p	Elapsed	CPU	% CPU	$S(N_p)$
1	7488 sec	7429 sec	99	1.0
4	2583 sec	2382 sec	92	2.9
8	1409sec	1254 sec	89	5.3

Table 5: Parallel simulations on a `Pentium Pro` cluster
Euler flow around an `ONERA M6` wing

7 Related works

Partitioning the mesh is now the most popular approach for parallelizing grid-oriented applications. But then two main directions emerge, that we could call "HPF" and "SPMD" for short.

The HPF approach, for example chosen by Brezany et al. [2], Saltz et al. [13], or Koppler [7], is clearly more general. No explicit call to communication is allowed. Everything is done by the compiler, following user-given *alignment directives* [5]. This high level of abstraction makes it rather difficult to the user, to impose the compiler a preferred way to organize the data and communications. Even with the `INDIRECT` directive, it is not easy to specify a good

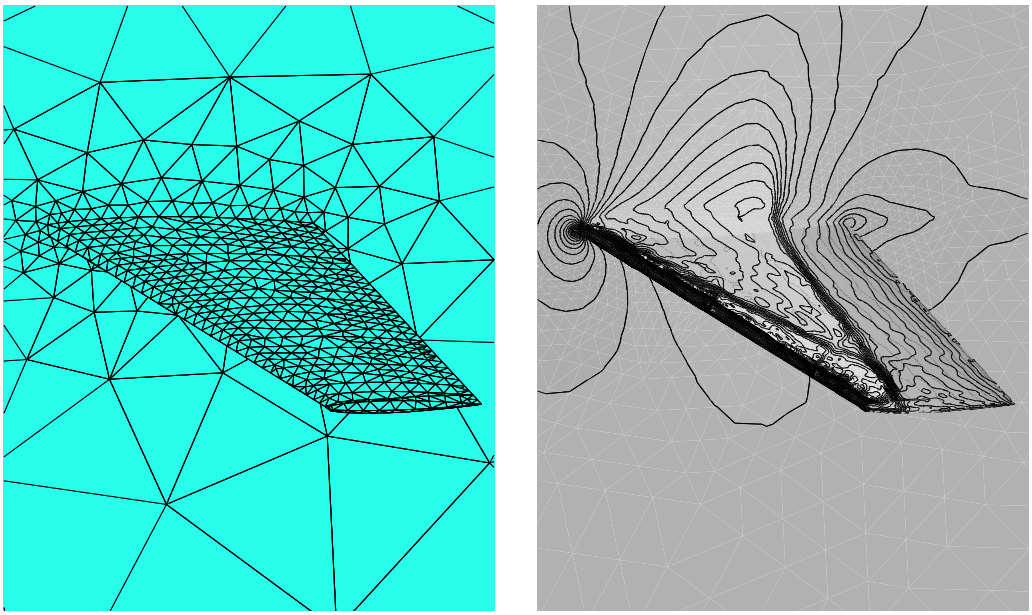

Figure 7: skin mesh and iso-Mach lines for the `ONERA M6` wing

mesh partitioning. Moreover, this approach requires a heavy transformation [2]. In particular one level of indirection must be added for mesh elements. This might make the HPF code difficult to maintain.

The SPMD approach, used for instance by O'Boyle et al. [12], or Farhat and Lanteri [3], uses a sequential high-level language, with explicit insertion of communications, using a standard library, such as MPI [11]. The user has more responsibilities, and freedom. This freedom allow various strategies for overlapping to be adopted. This freedom is dangerous in general but, used with caution, and with a tool, gives excellent results in specific cases, such as mesh-based computations. Since a distributed array actually becomes n separate arrays, no renaming nor extra indirection is required and the program computational part remains unchanged. Portability is just as good as HPF: libraries such as MPI have been ported to many parallel architectures, ranging from real distributed memory to shared memory.

Whatever target "language" is chosen, it is necessary to pre-compile the communications in some way, to reduce communication cost. Even using HPF,

it is necessary to use a strategy to "direct" the compiler towards a solution with lower communication cost. There are many interesting results, for example in O'Boyle et al. [12] when the data is accessed regularly. For irregular data such as meshes, some strategies are given by McManus [9], or Koppler [7]. There comes an important distinction: will the strategy incorporate user-given knowledge of the application?

If the strategy is automatic, it may be the classical "inspector-executor" model [6]. This is a runtime-compilation method, that dynamically determines the array cells that need be communicated across processors. In this direction, Saltz and coworkers at ICASE developed the PARTI [14] runtime primitives. The same paradigm is used by the Vienna school in their VFCS system. The resulting program may become rather complex. But the only essential difference with our approach is that we leave the "*inspector*" task to the mesh partitioner, and may select the overlapping pattern freely.

On the other hand, Koppler [7] and Ujaldón et al. [15] propose a user-given description of the relationship between the application data. In these works, as well as in the DIME [16] or ARCHIMEDES [1] environments, this knowledge spares the *inspector* phase. Koppler has the same target: mesh-based applications. His approach shares similarities with our tool, with two main differences. One is the target language, which is HPF. The other is how the user specifies how mesh elements are related. He uses an object-oriented specification of an *entity relationship diagram*.

When sub-meshes are modified, for example for load balancing, an extra communication step must be inserted just after mesh adaption.This is because moving mesh entities across processors means moving data. This problem is addressed for example by Williams [16] in the DIME environment.

Lastly, the kernel of our algorithm, that maps a graph onto another one, has similarities with the *Simulation* in the calculus of processes [10]. This suggested us optimizations to our algorithm, such as gathering some sequences of *data flow graph* arrows.

8 Conclusion

The process of parallelizing a large application is often composed of a sequence of tedious manipulations. We tried here to mechanize one such manipulation.

The result of this work is an interactive tool to apply SPMD parallelization to real programs on unstructured meshes. Performances show that this specialized SPMD strategy actually gives good speedups on real applications. Although improvements are possible, the tool is already able to find a placement of communication calls, faster than a manual transformation. Moreover, it is clearly less error-prone. Note furthermore that the kind of errors happening with poorly placed communications are very hard to fix. These errors are not likely to make the program crash. But they lead to false results, or degrade the convergence rate.

The underlying SPMD strategy is dedicated to mesh computations. We feel that this strategy, although definitely not general, must be applicable to other categories of programs: when the relationship between data can be formalized precisely, along with an overlapping mechanism. The *Overlap Automaton* specifies this relationship in a way that is independent from the particular program, data, or partition.

Acknowledgment

We especially thank Stéphane Lanteri for his help and enlightening explanations on the numerical aspects.

References

[1] BAO H., BIELAK J., GHATTAS O., KALLIVOKAS L., O'HALLARON D., SHEWCHUK J., XU J., *Large-scale Simulation of Elastic Wave Propagation in Heterogeneous Media on Parallel Computers.*, Computer Methods in Applied Mechanics and Engineering 152(1-2):85-102, (1998).

[2] BREZANY P., SIPKOVA V., CHAPMAN B., GREIMEL R., *Automatic Parallelization of the AVL FIRE Benchmark for a Distributed-Memory System*, ESPRIT Project *PPPE* report, (1995).

[3] FARHAT C., LANTERI S., *Simulation of compressible viscous flows on a variety of MPPs : computational algorithms for unstructured dynamic meshes and performance results*, Computer Methods in Applied Mechanics and Engineering, Vol. 119, pp. 35-60, (1994), (Also as Rapport de Recherche INRIA No. 2154), (1994).

[4] HASCOET L., *Automatic Placement of Communications in Mesh-Partitioning Parallelization*, Proceedings of PPoPP'97, Las Vegas, Nevada, ACM SIGPLAN Notices, Vol. 32, No. 7, pp. 136-144, (1997).

[5] High Performance Fortran Forum, *High Performance Fortran Language Specification*, Tech. Report v1.0, Rice University (1991).

[6] KOELBEL C., *Compiling Programs for Nonshared Memory Machines*, Ph.D. dissertation, Purdue University (1990).

[7] KOPPLER R., *Parallelization of Unstructured Mesh Computations Using Data Structure Formalization*, Proceedings of Euro-Par '98, Southampton, UK, (1998).

[8] LORIOT M., *MS3D : Mesh Splitter for 3D Applications, User's Manual*, Simulog, (1992).

[9] McMANUS K., *A Strategy for Mapping Unstructured Mesh Computational Mechanics Programs onto Distributed Memory Parallel Architectures*, Ph.D. thesis, University of Greenwich, (1996).

[10] MILNER R. *Communication and concurrency*, Prentice Hall, London (1989).

[11] MPI FORUM, *MPI: A Message-Passing Interface Standard*, Technical Report CS-94-230, University of Tenessee (1994).

[12] O'BOYLE M.F.P., KERVELLA L., BODIN F., *Synchronization Minimization in a SPMD Execution Model*, Journal of Parallel and Distributed Computing, Vol. 29, pp. 196-210, (1995).

[13] PONNUSAMY R., HWANG Y.S., DAS R., SALTZ J., CHOUDHARY A., FOX G., *Supporting Irregular Distributions Using Data-Parallel Languages*, IEEE Parallel & Distributed Technology, Vol. 3, No. 1, pp 12-24, (1995).

[14] SUSSMAN A., SALTZ J., DAS R., GUPTA S.,MAVRIPLIS D., PONNUSAMY R., *PARTI Primitives for Unstructured and Block Structured Problems*, Computing Systems in Engineering, vol. 3 num. 4 pp. 73-86 (1993).

[15] UJALDON M., ZAPATA E., SHARMA S., SALTZ J., *Parallelization Strategies for Sparse Matrix Applications*, Journal of Parallel and Distributed Computing, Vol. 38, pp. 256-266, (1996).

[16] WILLIAMS R. D., *DIME: Distributed Irregular Mesh Environment*, Technical Report C3P-861, California Institute of Technology, (1990).

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - B.P. 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Lorraine : Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - B.P. 101 - 54602 Villers lès Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot St Martin (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, B.P. 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399