Automatic Placement of Communications in Mesh-Partitioning Parallelization

Laurent Hascoët

INRIA Sophia-Antipolis
B.P. 93, 06902 Sophia-Antipolis, France
E-mail: hascoet@sophia.inria.fr

Abstract

We present a tool for mesh-partitioning parallelization of numerical programs working iteratively on an unstructured mesh. This conventional method splits a mesh into sub-meshes, adding some overlap on the boundaries of the sub-meshes. The program is then run in SPMD mode on a parallel architecture with distributed memory. It is necessary to add calls to communication routines at a few carefully selected locations in the code. The tool presented here uses the data-dependence information to mechanize the placement of these synchronizations. Additionally, we see that there is not a unique solution for placing these synchronizations, and performance depends on this choice.

1 Introduction

Generic parallelization methods analyze programs only at the language level. Therefore, they apply to every program written in that language. It appears however that such strategies may give poor results, even for highly parallel applications. The programmer, knowing the meaning of the application, would expect a much better parallelization, but this semantic parallelization is out of reach of even the most sophisticated parallelizer. In other words, for some classes of applications, using more abstract information known to the programmer, one can achieve a much better parallel adaptation than with a generic method. Note furthermore that the two strategies are not exclusive of each other.

There exist many tools that support generic parallelization techniques. On the other hand, there are very few tools that are specialized for particular classes of applications. One class of applications deserves attention, because their use is very widespread. These are the iterative resolutions on unstructured meshes, used in computational fluid dynamics, solid mechanics, and elsewhere.

This paper presents a tool to parallelize this class of programs. The basic method is to partition the mesh into several sub-meshes, then to run the program on each sub-mesh in parallel, calling communication routines when values must be exchanged between sub-meshes. There are two main problems:

- Find a good partitioning of the mesh, with a good load balancing and a minimal number of interface nodes. We don't address this problem here.
- Find the locations in the program where communications must take place. The tool we present computes automatically these locations.

The basis for this kind of analysis is the data dependencies graph, or data-flow graph (*dfg*). We use an existing parallelizing analyzer, called **Partita**, to compute the *dfg* of a given program. Then, analyzing every chain of data-dependence relations, we find out the places where a communication is needed.

It turns out that more than one solution may be found. Finding them all gives the opportunity to choose.

The rest of the paper is organized as follows. In section 2, we describe precisely both the parallelization method and the class of applications it addresses. This method, which is well known and not a result of this paper, gives very efficient parallel programs. Section 3 presents an abstract formalization of the method, exhibiting two tasks that may be automated. First, one may mechanically verify that the technique is applicable. Second, it is possible to introduce the communications mechanically where they are needed. Section 4 presents the resulting central algorithm, its implementation and the results on a small prototype example. In Section 5, we contrast our work to other existing techniques.

2 Parallelization by mesh partitioning

We review here the conventional parallelization method that is widely used for numerical programs on unstructured meshes.

2.1 The considered class of programs

A typical example that we consider, solves partial differential equations (e.g. Navier-Stokes) on a triangular 2D mesh, using an iterative method. A mesh is composed of nodes, edges, and triangles, that we shall call mesh "entities". The core of the algorithm consists in computing values on some mesh entities, using previous values on the neighboring entities. This process – called a **time step** – is repeated until some property is satisfied, typically that some measure of the difference between new values and previous values becomes sufficiently small. An entity refers to its neighbors using indirection arrays. In real programs, the loops on mesh entities usually iterate on mesh triangles or edges, while the

To appear in the sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Las Vegas, Nevada, June 18th-21st, 1997

physical values, that are read and assembled, are stored at mesh nodes. This is often called a **gather-scatter** method. Here is a sketch of this kind of algorithm:

`New_Values` = *initialization*
Repeat
  `Old_Values` = `New_Values`
  `New_Values` = 0
  Foreach `Element` ∈ `Mesh`
    gather the `Old_Values` *from neighbors of* `Element`
    compute the contribution of `Element`
    assemble into `New_Values` *for neighbors of* `Element`
  End Foreach
Until ‖ `New_Values` − `Old_Values` ‖ < ϵ

2.2 The mesh partitioning technique

Many parallelization techniques are based on partitioning a mesh. These SPMD (*Single Program on Multiple Data)* parallelization techniques consist in partitioning the program loops on mesh entities, and the arrays based on mesh entities accordingly. Each sub-mesh will be treated by one processor. This is done by a "mesh splitter", that uses the mesh geometry to return compact sub-meshes with a minimal interface size between them, to minimize communications. It is worthwhile noting that parts of the program that are not inside a partitioned loop will be executed identically on all processors, on the same data.

The particular method we use here is described and used in [2]. The **FORTRAN** program remains essentially unmodified, except for a few additional calls to communication routines. It is truly SPMD since exactly the same program runs on each processor. To achieve that, the sub-meshes returned by the mesh partitioner – we use **MS3D** [6] – are organized like the original mesh. The array names and access patterns are the same, and thus the computational part of the **FORTRAN** program remains exactly the same. This also makes code maintenance easier.

2.3 Communications and overlapping

Special attention must be given to the boundary of a sub-mesh, when a needed value resides on another sub-mesh. This is where the original program must be modified: a communication must be inserted, to make that value available to the current sub-mesh, i.e. processor. Since communications have an expensive overhead, they must be gathered. This is done by duplicating some mesh entities at the boundaries of sub-meshes. This is called *overlapping*. Many overlapping patterns are possible. Figure 1 shows one possibility, where triangles on the frontier are duplicated, and the corresponding nodes too. For example node a, on the frontier of sub-mesh A, is duplicated as a' on sub-mesh B. We say that a is in the *kernel* of A, while a' is in the *overlap* of B. Suppose that just before a time step, the values stored in a' are the same as in a. Then we compute correct values for all triangles, even duplicated, and therefore for all kernel nodes. But overlap nodes now carry incorrect values. Before the next time step, every overlap node (a', b'), must receive the correct value from its corresponding kernel node (a, b). All these communications can be gathered into a single procedure called in the source program.

Other overlapping are possible. For example on figure 2, only nodes on the boundary are duplicated. Suppose

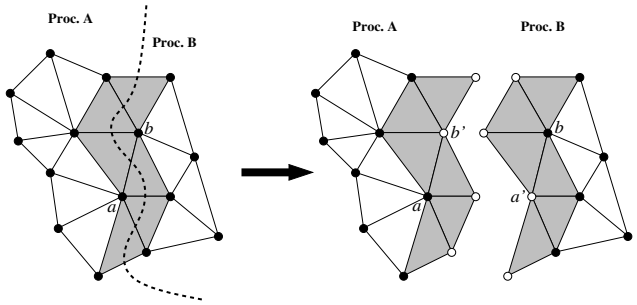

Figure 1: *One possible overlapping pattern*

again that nodes a and a' contain the same value before a time step. After this step, all nodes such as a and a' have incorrect values, but the correct value may be computed from the two. This supposes indeed that the assembly of the value for a node is associative and commutative, but this is generally the case. Therefore the necessary communication action is to get the values of a and a', assemble them, and send the result back to a and a'. The trade-off is a little more communication here, compared to a little redundant computation for the previous method

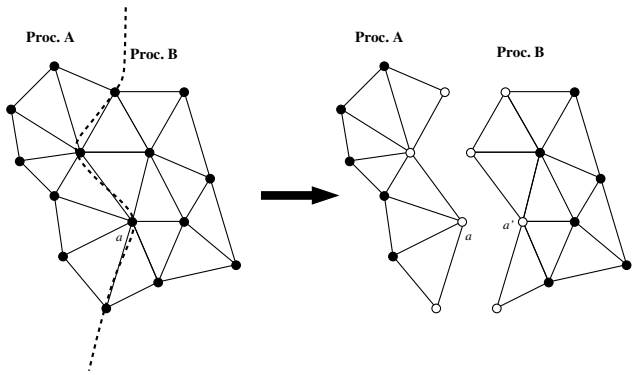

Figure 2: *Another overlapping pattern*

2.4 The complete process

The whole parallelizing process is now summarized on figure 3. We emphasize that splitting the original mesh and transforming the source program are two independent processes that share information on the chosen overlapping pattern chosen. The first process only uses the geometrical description of the mesh, the second process only needs to analyze

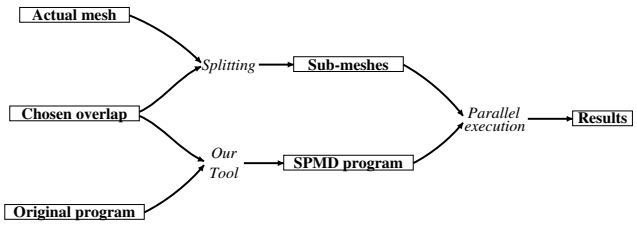

Figure 3: *General parallelization process*

the data dependences in the program. A real-size application of this process is described and evaluated in [2], exhibiting a very good speedup ranging between 20 to 26 for 32 processors. The SPMD program was obtained by manually transforming the original program. Our contribution, described in the remaining of this paper, is to design and implement a tool to construct mechanically this SPMD program.

3 Automatic placement of synchronizations

Our tool is dedicated to a particular class of applications, and takes advantage of information given by the programmer. Thus, section 3.1 describes the information required from the user. Then the tool checks that the input program is really a member of that target class of applications. This is discussed in section 3.2 . In section 3.3, we present the principles that command the need of communication between sub-meshes. Then section 3.4 formalizes these principles through a specification of the evolution of the state of data with respect to communication.

3.1 User input

To begin with, the user must choose the overlapping pattern among a small collection of predefined patterns, such as the patterns shown on figures 1 and 2. This choice is left to the user, according to his own experience and preferences. Obviously, there is a compromise to be found. On one hand, a large overlap width will result in redundant computation, but it will allow to gather manier communications at the same time. Figures 1 and 2 are the most frequently used overlapping patterns. Some people use the pattern of figure 2, with the benefit of no redundant computation, others use the pattern of figure 1, others even advocate patterns with two layers of overlapping triangles, when the value computed at some node depends of nodes two triangles away.

Then the user must designate the loops and the variables that will be partitioned, and how they must be partitioned. For example, for the pattern of figure 1, this means specifying for each loop and variable whether it must be partitioned node-wise, edge-wise, or triangle-wise. This can be done through a small data file, as it is done now. A graphical interface could be helpful.

This information we require is redundant, because the program imposes constraints. For example, an array partitioned on nodes and accessed without indirection may be found only in loops partitioned on nodes too.

This redundancy may be used, either to reduce the information required from the user, or to cross-check it. For example, we feel that it could be sufficient to designate only the partitioned loops, and deduce the partitioned variables.

3.2 Verifying the applicability of the method

Like in any parallelization method, this user-given partitioning will be legal if dependences are not violated by any parallel execution order. Here, the execution order may change only between the iterations of a partitioned loop. Loops that are not partitioned will be executed sequentially and therefore dependences carried by them will always be respected.

Figure 4 summarizes all cases of dependence that may happen. On this figure, we represent the program as a succession of partitioned loops and non-partitioned parts of code. Loops not partitioned are considered as if they were unrolled, and therefore do not appear here. A partitioned loop is shown as a stack of horizontal lines, each line representing one iteration. Thin arrows represent dependences between operations. There are five kinds of such dependences. Three are the classical data-dependences, namely **true** *(write to read)*, **anti** *(read to overwrite)*, and **output** *(write to overwrite)*. We add two dependences, the **control** going from a control structure (e.g. a test, a loop header) to the controlled operations, and the **value** that links operands to operations, inside an instruction. The SPMD execution

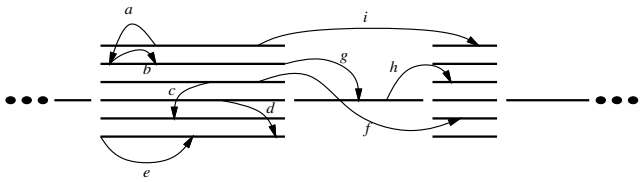

Figure 4: *All cases of dependences*

of the program may execute partitioned iterations in any order, and therefore will not respect dependences such as *a*, *c*, and *d*. So they must not be found in the original program. If *d* is not involved in a dependence cycle, like *a*, then making two loops out of the first loop may transform case *d* into case *f*, which is more acceptable. But this transformation of the original program is outside the scope of this work. Dependences *f* and *g* will be respected, because a communication is necessary between their origin and their destination, and this will ensure a good execution order. There is a restriction if dependence *g* comes from a particular, explicit, partitioned iteration. In that case, we have no way to relate parallel iteration numbers to original ones. So dependence *g* will be forbidden, except for the special case of reductions (global sums, products, etc... of arrays) Dependences *b*, *e*, *h*, and *i* will always be respected.

Lastly, note that classical parallelization methods, such as induction variable detection, variable localization, or reduction operation detection, may help removing some dependences. We shall use these methods to remove forbidden dependences.

To summarize, a loop partitioning provided by the user is acceptable if no dependence (remaining after induction and reduction detection, and localization) is carried across the iterations of the partitioned loop. This checking, when performed manually, is an important source of errors. An important feature of our tool is that it checks all dependences automatically.

3.3 Detecting communication needs

How do we select the places where the overlapping values must be updated? Let us simulate the program's execution. Because communications occur between definitions and uses of variables, it is natural to observe the data-flow graph. Let us choose the overlapping pattern of figure 1. Suppose furthermore that at the beginning of the program sketch on figure 5, the partitioned array `OLD`, indexed on the mesh nodes, contains coherent overlapping values. The value of `OLD` is read inside the first loop, to compute a value for a given triangle. In principle, we obtain an array indexed on triangles. The values on overlapping triangles are coherent. Then, following the dependence arrows, we find out that this array on triangles is used to compute a `NEW` value on

```
Forall i          (split loop on mesh triangles)
   ...
   val2 = OLD(SUMMIT2(i))
   ...
   NEW(SUMMIT1(i)) = ... val2 ...
End
...
Forall j          (split loop on mesh nodes)
   diff = NEW(j) - OLD(j)
   sqrdiff = sqrdiff + diff*diff
End
... sqrdiff ...
Forall i          (split loop on mesh triangles)
   ...
   ... = ... NEW(SUMMIT3(i)) ...
```

Figure 5: *A program sketch*

nodes. As we saw before, the overlapping pattern chosen implies that the array `NEW` now has an **incoherent** overlap. From now on, we follow dependence arrows until we reach a point where this incoherence is forbidden. On figure 5, one such point is the last place where `NEW` is read, since we need a coherent array to build a correct value for each triangle. We conclude that a communication restoring the overlap coherence must be inserted between the place where `NEW` is written, and the last place where `NEW` is read.

A similar thing happens when computing the scalar `sqrdiff`. The reduction instruction computing `sqrdiff` returns an "incoherent" scalar, each processor holding a partial sum. So we need a communication computing the total sum and sending it back to all processors, between this location and the following use of `sqrdiff`, where the total sum is required.

3.4 Formalization

In the above section, we saw that it is natural to follow the data-flow graph rather than the control-flow graph. In fact we only follow **true** *(write to read)*, **control**, and **value** *(operand to operation)* dependences. The **anti** *(read to overwrite)* and **output** *(write to overwrite)* dependences do not represent the chain of values leading to the result, but only constrain execution order, which was already discussed in section 3.2.

What actually flows through the data-flow graph is not a given variable or array, but rather a value. Let us call it the **flowing data**. The flowing data may have one of some predefined states. These states indicate whether the data is based on nodes, edges, or triangles, and whether the overlap is coherent or not.

The state of the flowing data evolves across data-flow dependences. The allowed evolutions form a set of transitions between flowing data states. This results in a finite state automaton, consequence of the overlapping pattern, that we call the **overlap automaton**. Figure 6 shows the overlap automaton derived from the overlapping pattern of figure 1. There are five possible states:

- when the flowing data is shaped as a partitioned array on nodes, it may be in state $\mathbf{Nod}_0$, if overlap values are correct, or else in state $\mathbf{Nod}_1$.

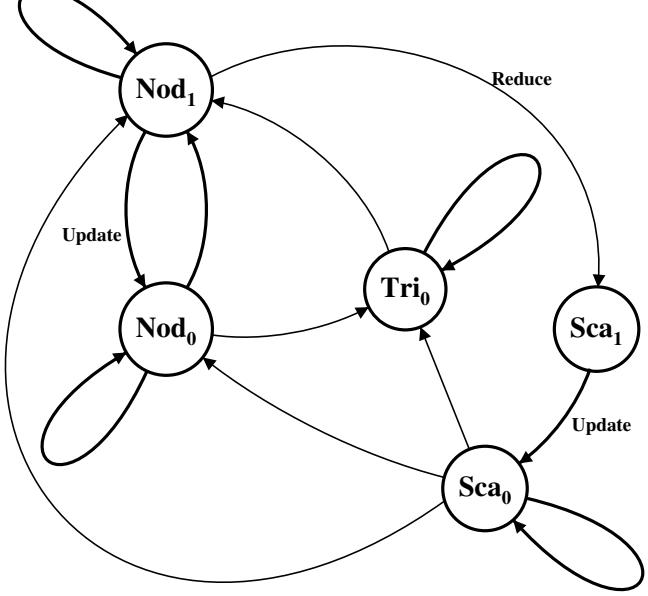

Figure 6: *Overlap automaton for the overlapping pattern of figure 1.*

- when the flowing data is shaped as a partitioned array on triangles, with coherent values on overlapping triangles, its state is $\mathbf{Tri}_0$. There is no state allowed with incoherent values.
- when the flowing data is on a scalar, it may be in state $\mathbf{Sca}_0$, if each processor holds the same value, or else in state $\mathbf{Sca}_1$.

Localized variables are partitioned along with their partitioned enclosing loop. This happens typically for temporary scalars or intermediate operation results inside partitioned loops.

Let us now examine some sample transitions:

$\mathbf{Tri}_0 \longrightarrow \mathbf{Nod}_1$: Using a triangle-based flowing data to compute a node-based value, returns incoherent values on the overlap, i.e. state $\mathbf{Nod}_1$.

$\mathbf{Nod}_1 \longrightarrow \mathbf{Nod}_0$: Reading a node-based variable with incoherent overlap, at a location where a coherent overlap is required, forces the insertion of a communication to update the values on the overlap.

$\mathbf{Nod}_1 \longrightarrow \mathbf{Sca}_1$: In the case of a reduction operation, a node-based value with incoherent overlap may be used to compute a scalar. The resulting scalar is the partial reduction on each processor, therefore in state $\mathbf{Sca}_1$.

The two transitions labeled by "**Update**" are special. Traversing them implies that a communication must be inserted somewhere between the extremities of the data-dependence. Also the overlap automaton of figure 6 shows two kinds of transitions. The thick arrows represent transitions that may happen only when crossing **true** dependences, whereas thin arrows correspond only to **value** or **control** dependences.

There is one specific overlap automaton for each overlapping pattern. To give another example, figure 7 shows the overlap automaton for the other overlapping pattern shown on figure 2. The differences from the automaton of figure 6

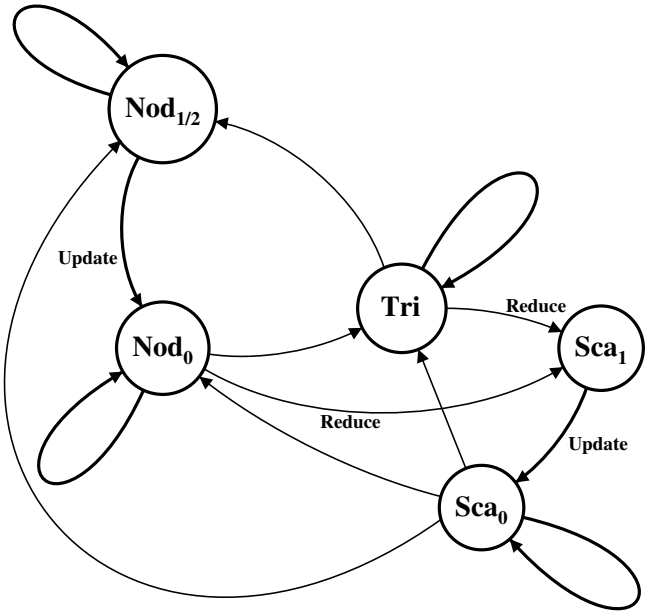

Figure 7: *Overlap automaton for the overlapping pattern of figure 2.*

are interesting:

- The "incoherent overlap" state ($\mathbf{Nod_{1/2}}$) is different. Now the correct value does not reside on any of the duplicated nodes, but will be obtained by combining all of those values.
- The reduction on node-based arrays now requires that the correct value be available on the overlapping nodes too.
- It is no longer possible to consider a coherent state as a special case of an incoherent state, since updating it twice would result in *doubling* the values stored on overlapping nodes, which would be incorrect

This formalization is not restricted to 2-D meshes. For example, figure 8 shows the overlap automaton for partitioning a three-dimensional tetrahedra mesh, with an overlapping pattern of one overlapping layer of tetrahedra. It is therefore analogous to the one shown on figure 1, but for three dimensions. The new state $\mathbf{Thd_0}$ stands for a tetrahedron-based data, coherent on its overlap. The new states $\mathbf{Edg_0}$ and $\mathbf{Edg_1}$ represent respectively the coherent and incoherent states of a data based on edges. It is interesting to notice that the automaton of figure 6 can be derived from the one on figure 8, simply by forgetting the unused states ($\mathbf{Thd_0}$, $\mathbf{Tri_1}$, $\mathbf{Edg_0}$, and $\mathbf{Edg_1}$), and forgetting the corresponding transitions.

Now we can formulate rigorously the informal method of the previous section. The choice of the overlapping pattern amounts to the choice of one particular overlap automaton, among a small set of predefined ones. Then, to find the places where to set communications, one needs to

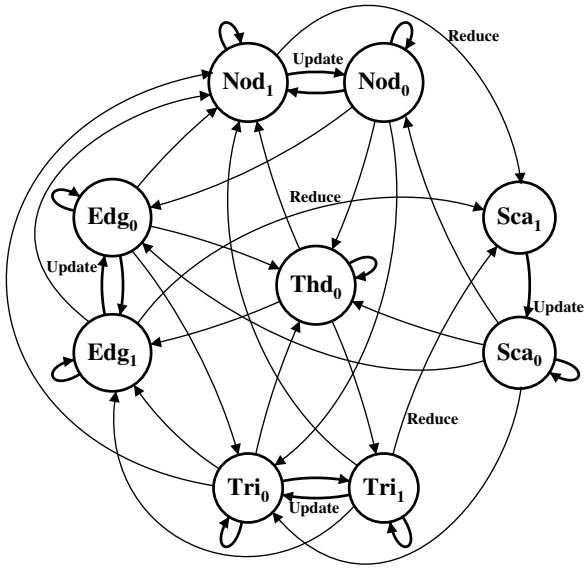

Figure 8: *An overlap automaton for a 3-D mesh partition*

find 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 a program input, $M_n(N)$ equals its given initial state.
2. For every data-flow node N of a program output, $M_n(N)$ equals its required result state.
3. For every data-flow arrow A in the program,

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

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

4 Algorithm and implementation results

The formalization in the previous section leads to an algorithm for finding the mappings M_a and M_n. Then from M_a we shall get the places where to set communications, and from M_n, we shall get the precise iteration domain of each partitioned loop, i.e. for a loop on nodes, whether it should iterate on kernel nodes only, or also on overlap nodes.

To find these mappings, we traverse the data-flow graph, propagating the overlap state of the flowing data. For every input data, the overlap state is given. Then the state propagates according to the overlap automaton. Each data-flow node can only be associated to one state. In particular, when the data-flow graph cycles, we require that the propagated state be identical on each visit of the data-flow node. This allows us to stop propagation on the second visit, a fortiori avoiding infinite looping. Finally, for every output data, the propagated overlap state must be identical to a given result state.

The propagation algorithm is nondeterministic. Some alternatives lead to unacceptable situations (many states per

node, no applicable transition, exit states different from required, etc...). In general, for a given program and a given overlapping pattern, there may be more than one solution mapping. For these reasons, a backtracking mechanism is necessary. In the following sketch of the algorithm, this backtracking mechanism is simplified for clarity. It resides in the second function, where each available transition in the overlap state is tried, until one that leads to success is found.

Function cross_node(node$_{<dfg_node>}$, state$_{<overlap_state>}$, $<M_n \bullet M_a>$)
  Switch on cases
    • M_n(node)==state
      Return $<M_n \bullet M_a>$
    • M_n(node)==state$_2$ **and** state$_2 \neq$ state
      Return false
    • M_n(node) **undefined**
      $M_n = M_n +$ node↦state
      arrows = *data_flow arrows leaving* node
      Foreach arrow ∈ arrows
        propagation_success =
          cross_arrow(arrow, state, $<M_n \bullet M_a>$)
        If propagation_success **then**
          $<M_n \bullet M_a>$ = propagation_success
        else
          Return false
        End If
      End Foreach
      Return $<M_n \bullet M_a>$
  End Switch
End Function cross_node

Function cross_arrow(arrow$_{<dfg_arrow>}$, state$_{<overlap_state>}$, $<M_n \bullet M_a>$)
  transitions =
    all transitions leaving state, *with respect to* arrow
  Foreach transition ∈ transitions
    propagation_success =
      cross_node(*destination*(arrow),
        destination(transition),
        $<M_n \bullet M_a +$arrow↦transition$>$)
    If propagation_success **then**
      $<M_n \bullet M_a>$ = propagation_success
      Return $<M_n \bullet M_a>$
    End If
  End Foreach
  Return false
End Function cross_arrow

We have implemented this algorithm in Lisp on top of the data-dependence graph computed by the **Partita** tool. For efficiency, recursive functions have been implemented iteratively. In the generated output, the communication instructions appear as comments. The user replaces them by calls to subroutines using any communications package, such as PVM [3] or MPI [8]. The choice of the package has no influence on our algorithm. We show the results on an short example **FORTRAN** program that summarizes all the features of our target class of programs. Figure 9 shows one possible solution. Figure 10 shows another possible placement of communications, obtained after some backtracking after the previous solution is found. This placement happens to be the same as what was done initially by hand, on the real program from which this example is derived. Both

```
      subroutine TEST7v1(INIT,RESULT,nsom,ntri,SOM,AIRETRI,AIRESOM,
     &                   epsilon,maxloop)
C
      integer nsom,ntri,maxloop
      integer SOM(2000,3)
      real epsilon
      real INIT(1000),RESULT(1000),AIRESOM(1000)
      real AIRETRI(2000)
C
      integer i,loop,s1,s2,s3
      real vm,sqrdiff,diff
      real OLD(1000),NEW(1000)
C
C$ITERATION DOMAIN: OVERLAP
      do i = 1,nsom
        OLD(i) = INIT(i)
      end do
      loop = 0
 100  loop = loop + 1
C$ITERATION DOMAIN: OVERLAP
      do i = 1,nsom
        NEW(i) = 0.0
      end do
C$ITERATION DOMAIN: OVERLAP
      do i = 1,ntri
        s1 = SOM(i,1)
        s2 = SOM(i,2)
        s3 = SOM(i,3)
        vm = OLD(s1) + OLD(s2) + OLD(s3)
        vm = vm * AIRETRI(i) / 18.0
        NEW(s1) = NEW(s1) + vm/AIRESOM(s1)
        NEW(s2) = NEW(s2) + vm/AIRESOM(s2)
        NEW(s3) = NEW(s3) + vm/AIRESOM(s3)
      end do
      sqrdiff = 0.0
C$ITERATION DOMAIN: KERNEL
      do i = 1,nsom
        diff = NEW(i) - OLD(i)
        sqrdiff = sqrdiff + diff*diff
      end do
C$SYNCHRONIZE METHOD: overlap-som ON ARRAY: NEW
C$SYNCHRONIZE METHOD: + reduction ON SCALAR: sqrdiff
      if (sqrdiff .lt. epsilon) goto 200
      if (loop .eq. maxloop) goto 200
C$ITERATION DOMAIN: OVERLAP
      do i = 1,nsom
        OLD(i) = NEW(i)
      end do
      goto 100
C$ITERATION DOMAIN: OVERLAP
 200  do i = 1,nsom
        RESULT(i) = NEW(i)
      end do
C
      end
```

Figure 9: *One possible generated SPMD programs*

solutions set basically the same communications, but the solution on figure 10 has the advantage of grouping the two main communications, thereby saving an additional communication overhead. On the other hand, the solution on figure 9 delays one communication so that the iteration space of some loops may be restricted to the kernel nodes, saving some instructions on the overlap. The choice between these solutions is, for the moment, left to the user.

```
      subroutine TEST1v2(INIT,RESULT,nsom,ntri,SOM,AIRETRI,AIRESOM,
     &                   epsilon,maxloop)
C
      integer nsom,ntri,maxloop
      integer SOM(2000,3)
      real epsilon
      real INIT(1000),RESULT(1000),AIRESOM(1000)
      real AIRETRI(2000)
C
      integer i,loop,s1,s2,s3
      real vm,sqrdiff,diff
      real OLD(1000),NEW(1000)
C
C$ITERATION DOMAIN: KERNEL
      do i = 1,nsom
        OLD(i) = INIT(i)
      end do
      loop = 0
 100  loop = loop + 1
C$SYNCHRONIZE METHOD: overlap-som ON ARRAY: OLD
C$ITERATION DOMAIN: OVERLAP
      do i = 1,nsom
        NEW(i) = 0.0
      end do
C$ITERATION DOMAIN: OVERLAP
      do i = 1,ntri
        s1 = SOM(i,1)
        s2 = SOM(i,2)
        s3 = SOM(i,3)
        vm = OLD(s1) + OLD(s2) + OLD(s3)
        vm = vm * AIRETRI(i) / 18.0
        NEW(s1) = NEW(s1) + vm/AIRESOM(s1)
        NEW(s2) = NEW(s2) + vm/AIRESOM(s2)
        NEW(s3) = NEW(s3) + vm/AIRESOM(s3)
      end do
      sqrdiff = 0.0
C$ITERATION DOMAIN: KERNEL
      do i = 1,nsom
        diff = NEW(i) - OLD(i)
        sqrdiff = sqrdiff + diff*diff
      end do
C$SYNCHRONIZE METHOD: + reduction ON SCALAR: sqrdiff
      if (sqrdiff .lt. epsilon) goto 200
      if (loop .eq. maxloop) goto 200
C$ITERATION DOMAIN: KERNEL
      do i = 1,nsom
        OLD(i) = NEW(i)
      end do
      goto 100
C$ITERATION DOMAIN: KERNEL
 200  do i = 1,nsom
        RESULT(i) = NEW(i)
      end do
C$SYNCHRONIZE METHOD: overlap-som ON ARRAY: RESULT
C
      end
```

Figure 10: *Another possible generated SPMD programs*

5 Related and future works

5.1 Mesh partitioning techniques

Mesh partitioning is now recognized as an important approach for the parallelization of grid-oriented applications. When the mesh is structured, the arrays on this mesh follow its topology. Then the partitioning problem is easier, since splitting the mesh means partitioning the arrays in a natural way. In that case, special extensions of the programming languages, such as **High Performance Fortran** [4], help the user specify a *regular* mesh partitioning, and it is not necessary to set an overlap and associated communications by hand.

But unstructured meshes are more powerful, because they allow mesh *adaption*. Unfortunately HPF is not adapted to express parallelism for unstructured meshes. However some authors propose methods to reorder and renumber mesh entities, so that the HPF primitives may be sufficient to express the data distribution. For example paper [10] proposes a way to emulate irregular distributions in HPF. Others are advocating an evolution in HPF to express directly such distributions.

Partitioning an unstructured mesh is a special case of *data decomposition*. Paper [9] presents a general formal framework for parallelization by *data decomposition*. Since the framework is general, it cannot take advantage of the geometry of the mesh, so there is no notion of nodes on the border or in the inside of a sub-mesh. There is no notion of overlap either, so an explicit communication must be set each time a single instruction fetches a value that resides on another processor. Some optimizations are given, but in the case of monotonic indirection functions, which does not apply here.

The *inspector/executor* paradigm [5] is a popular method to optimize communications when partitioning a mesh. This is a runtime-compilation method, that dynamically determines the array cells that need to be communicated across processors. In this direction, Saltz and coworkers at ICASE devellopped the PARTI [11] runtime primitives. The same paradigm is used by the Vienna school in their VFCS system.

The principal difference is that the set of values that must be communicated is computed at runtime, during a special execution of *one* time step. Then this information is used to optimize subsequent time steps, by regrouping communications, and by storing these values into "ghost cells", that are very close to our "overlap". So the way the information is used is very similar to our method, but the way it is collected differs. A runtime analysis may be hard to beat, especially in the general case. But we feel that in specific cases, such as meshes, the static analysis may return excellent results using geometric, user-given knowledge. To put it differently, in our tool, the run-time *inspector* phase is replaced by an extra static analysis done by the mesh splitter. This is possible because we focus on a special case (e.g. meshes).

Another difference is that there is no emphasis on the "placement" of synchronizations, and there is no choice of the overlap size. In *inspector/executor* methods, the overlap width is minimal, and therefore communications must be done between each split loops. This is not a bad strategy, but the user may want to regroup communications further, using a larger overlap. In this respect, there is a complementarity with our work. One could try to combine the *inspector* results with our method, to place communications less frequently, choosing a larger overlap.

Another interesting point in the PARTI tool is the "flocalize" step. This rearranges split objects, to group "ghost cells", for example at the end of split lists. The same mechanism would apply to our tool, and would allow more efficiency on processors that may perform computations and communications independently. In our tool, this "flocalize" step would become an extra reordering in the mesh splitter.

Some environments exist, that combine all necessary tools, from the domain meshing and mesh splitting, to the SPMD parallel code generation. In the ARCHIMEDES project at CMU, or in the DIME [12] environment, the algorithm is specified in a higher-level formalism, and the

overlapping pattern is restricted to the one on figure 2. As one could expect, there is no need for an *inspector* phase, since the programs are known to function on a triangular 2-D mesh. The higher-level formalization is certainly a good method to solve the communications placement problem. However, it does not apply to legacy code.

The DIME++ extension of DIME tries to extend the domain of application of DIME to something more general than 2-D meshes. DIME++ works on "sets of objects that have indexes to other sets of objects". There is a relationship with our approach, where the sets of objects and their relations are described into the overlap automaton, together with the chosen overlapping pattern. This also raises the question of how to become independent from the notion of mesh, extracting this relationship information from the program itself.

5.2 Automata and Simulation

The current, straightforward implementation may become expensive on large programs.

The central task of our algorithm is to map one graph, the *dfg*, to another, the overlap automaton. In another domain of computer science, the calculus of processes, this is called *Simulation* [7]. To simplify things, suppose that we start with the *dfg* with communication calls already placed. Then our algorithm may run in test mode, checking that this particular placement gives a behavior compatible with the overlap. This is checking that in every possible execution, i.e. in every possible path in the *dfg*, the state of the flowing data follows a legal evolution in the overlap automaton. The *dfg* is then said to "simulate" the overlap automaton.

Simulation – and its symmetric variant called "bisimulation" – are important methods for which optimized algorithms exist. It can be interesting to apply these optimized algorithms to our case.

For example significant speedup would come from reducing the "simulating" graph (the *dfg*), by merging sequences of dependences that would not change the "simulated" state (the overlap state). This results in a faster visit of the *dfg*, and faster backtracks too.

5.3 Load balancing

Meshes are split according to their geometry. This implicitly supposes that all mesh nodes will require similar computation time. This is a rather rough approximation. After one time step, one sub-mesh may turn out to require more computation. We will need a way to improve the load balancing according to the effective computation time on each sub-mesh. The same need will arise with *adaptive mesh computations*. After a solution is computed, it is useful to refine the mesh, adding more elements where the physical solution varies rapidly (e.g. shocks), and resume execution. This will greatly affect the load-balance among sub-meshes.

This problem was not studied here. We believe, although, that the placement of synchronizations needs not change, since this placement did not depend on the geometry of the sub-meshes. However, an extra communication step must be inserted just after mesh adaption, since moving mesh entities across processors implies moving data. This problem is addressed for example by Williams [12] in the DIME environment.

Using *adaptive meshes* implies modifications mostly in the mesh splitter. Since this splitter would be called repeatedly on refined meshes, the splitting algorithm must be particularly efficient, and incrementality would be a capital advantage.

6 Conclusion

The result of this work is an automated method to select the locations where to insert communications. This method is specialized for the cases where the relations between sets of data, and the chosen overlapping pattern, may be summarized into an *overlap automaton*. The overlap automaton is independent from the particular program, mesh, or partition. This paper presents the overlap automata for various cases on 2-D and 3-D meshes.

Note that this operation is usually performed manually. An engineer typically needs several days to do that, especially when the original program is *legacy code*. Note furthermore that errors in manual transformation may occur. These errors may be very difficult to trace, since bad synchronizations sometimes imply a small imprecision of the result, and/or a different convergence rate.

References

[1] 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).

[2] 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).

[3] GEIST A., BEGUELIN A., DONGARRA J., JIANG W., MANCHEK R., SUNDERAM V., *PVM 3 User's Guide and Reference Manual* Oak Ridge National Laboratory TM-12187, (1993).

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

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

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

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

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

[9] PAALVAST E.M., SIPS H.J., van GEMUND A.J., *Automatic Parallel Program Generation and Optimization from Data Decompositions*, International Conference on Parallel Processing, Vol. 2, pp. 124-131 (1991).

[10] PONNUSAMY R., HWANG Y.S., SALTZ J., CHOUDHARY A., FOX G., *Supporting Irregular Distributions in FORTRAN 90D/HPF Compilers*, University of

Maryland, Department of Computer Science and UMIACS Technical Reports CS-TR-3268, UMIACS-TR-94-57 (1994).

[11] 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).

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