A method for automatic placement of communications in SPMD parallelisation

Laurent Hascoët

INRIA *Projet Tropics, 2004 Route des Lucioles, B.P. 93, 06902 Sophia-Antipolis Cedex, France E-mail: Laurent.Hascoet@sophia.inria.fr*

Abstract

We present a method for automatic SPMD parallelisation of applications, when the data is partitioned along processors with overlapping, redundant values. The problem is to insert just enough communication calls to ensure coherence of the redundant values. Our method uses a representation of the topology of the data and of the duplicated values, that we call the *Overlap Automaton*. We present a tool that implements this method, and its application to two-dimensional and three-dimensional Navier-Stokes flow solvers. Performance results are given.

Key words: single program multiple data. parallelisation. mesh partition. static analysis. program transformation.

1 Introduction

Partitioning data and computation is a classical approach to parallelisation. In this approach, each processor is in charge of one part of the computation domain, and data is sent to, and received from other processors when necessary, by means of calls to standard communication libraries.

Generally, an *overlap* mechanism is utilised: some data is replicated on two or more processors, and is updated when it becomes out of date. Communications may then be deferred, and gathered into a single communication call. The same program is run on each processor, and it is very close to the sequential program. Only calls to communication routines are inserted at carefully selected places, and these calls use data files that define which values must travel to which processor ("*communication lists*"). This is commonly called the "*Single Program Multiple Data*" model (SPMD).

Therefore, parallelisation by partitioning needs to know what communications must be performed, and when. This decision depends on the program itself, and also on the distribution of the replicated variables ("*overlapping pattern*"). For example for mesh-based programs, this SPMD parallelisation process is summarised on figure 1.

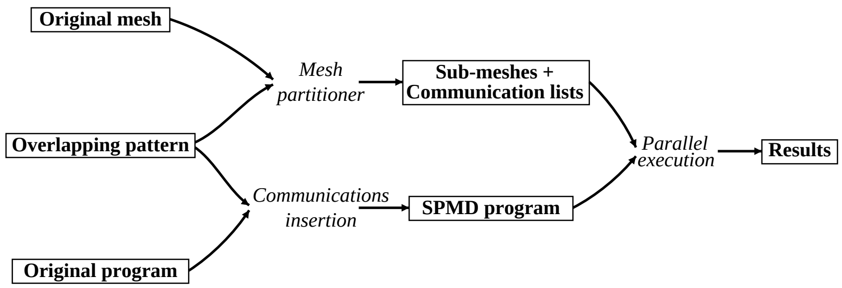

Fig. 1. *SPMD parallelisation process*

Inserting communication calls is usually tedious and error-prone, and it must be done each time the program is modified. Errors due to poorly placed communications are very hard to fix. Instead of making the program crash, they lead to false results or degrade the convergence rate. Also, a good understanding of the program's algorithm is required, which is hard to gain on a "legacy" application. In this paper, we present a method for automatic placement of these communications into the SPMD program. This method is based on the program's *data-flow graph* on one hand, and on the other hand on the particular overlapping pattern, whose effect is represented by a finite state automaton ("*overlap automaton*"). This automaton summarises the allowed evolutions of the state of each duplicated variables ("up to date" or "out of date"), along the data-flow paths.

The SPMD model amounts to *distributing* the loops by partitioning their iteration domain, and distributing the arrays that are used by these loops accordingly. The method presented here is specialised to a (wide enough) class of programs, where distributed arrays are relying on a limited number of data structures, and are accessed during distributed loops, through so-called "*indirection arrays*", which are constant. This is in particular the case for computations on unstructured meshes, using "*gather-scatter*" loops, that will be used for illustrating examples throughout this paper and for the final application to real 2D and 3D flow solvers.

The rest of the paper is organised as follows: after a survey of SPMD parallelisation methods and tools for communication insertion (section 2), we shall describe our method for inserting communications in section 3. Then in section 4, we shall discuss complexity questions for the resulting algorithm. Finally, section 5 will present the tool that we built to demonstrate our method, and its application to two numerical flow solvers.

2 Methods and Tools for SPMD parallelisation

To begin with, a classical alternative to SPMD parallelisation is the HPF approach (Brezany et al. [3], Saltz et al. [12], Koppler [8]), which is more general but less efficient. In HPF, communications are not explicit. This is left to the compiler, following user-given *alignment directives*. The drawback is that one cannot easily impose a preferred organisation of data and communication. HPF is still heavily dependent on structured data, with affine array indexes, and gives poor results otherwise. The `INDIRECT` directive partly copes for that, but this approach often requires a heavy transformation [3], with an additional indirection level.

The SPMD approach (e.g. O'Boyle et al. [11], Farhat and Lanteri [4]), uses a sequential language, plus explicit communications (e.g. MPI). Portability of MPI is just as good as HPF. The user has more responsibilities, and freedom. This freedom may be dangerous in general: methods and tools are necessary to avoid this risk. This approach gives excellent results in specific cases, such as mesh-based computations. Moreover, this can be combined with finer-grain parallelisation: each copy of the SPMD program can be itself parallelised [6].

Whatever approach is chosen, it is more efficient to pre-compile communications. Even in the HPF context, it is necessary to use a strategy to "direct" the compiler towards a solution with lower communication cost. There are many interesting results (O'Boyle et al. [11]), when the data is accessed regularly. For irregular data such as meshes, see McManus [9], or Koppler [8]. In our context, this implies to place communications calls into the program.

One classical way is the "inspector-executor" model [7]. This is a runtime-compilation method, that dynamically determines the array cells that need be communicated across processors. See for example the Saltz et al. PARTI primitives [13], or the Vienna school VFCS system.

Others (Koppler [8], Ujaldón et al. [15]) propose a user-given description of the relationship between the application data. This spares the *inspector* phase, like also in the DIME [16] or ARCHIMEDES [2] environments. But these tools are specialised to a specific application. Koppler's method is devised for mesh-based computations, using an object-oriented specification of a so-called *entity relationship diagram*, but the target language is HPF. Lastly, numerical libraries like PETSc [1] or Aztec [14], internally use SPMD parallelism with an overlap. But again, this is specialized for e.g. sparse linear systems, and there is no way to specify another arrangement of data and overlap.

Thus, we found very few work on the question of placement of communications for SPMD parallelisation, addressing various overlapping strategies in a unified manner. The method described here, first presented in [5], has this goal.

3 Automatic placement of communications

We are going to formalise the intuitive reasoning that one usually follows: for a number of reasons, the data that flows through the program becomes more or less out of date on the replicated variables. At some later stage, this data is used in such a way that the replicated variables must be up to date. Therefore a communication must be inserted in between to update them. The general reason behind that is that partitioned arrays are accessed using indirection arrays. This happens classically in mesh-based computations, which is the illustration we shall use in the sequel.

Consider for example the piece of code sketched on figure 2, which is taken from a mesh-based flow solver. It composes of two gather-scatter loops, with a

```
Forall i        (split loop on mesh triangles)
   ...
   val2 = OLD(VERTEX2(i))
   ...
   NEW(VERTEX1(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(VERTEX3(i)) ...
```

Fig. 2. *A piece of code from a flow solver*

reduction loop between them. The arrows show the data-flow graph. On sub-mesh borders, replicated mesh nodes do not have all their neighbor triangles, and thus the first loop will assign wrong values of `NEW` for them. Since the third loop needs correct values on the overlap, a communication must be inserted between the first and third loop.

The above is a consequence of the mesh and overlap topologies. Our method is to formalise the effect of this topology with a user-given finite state automaton, that we call "*Overlap Automaton*". This automaton defines the *valid* states, i.e. states of distributed arrays where replicated values are either up to date, or can be updated by a communication call. When an array is up to date, it contains the same values as for the sequential program. The automaton also defines the *valid* transitions, that take a value in a valid state and use it to generate another value still in a valid state.

For example, consider a 2D triangular mesh as shown on figure 3, with an

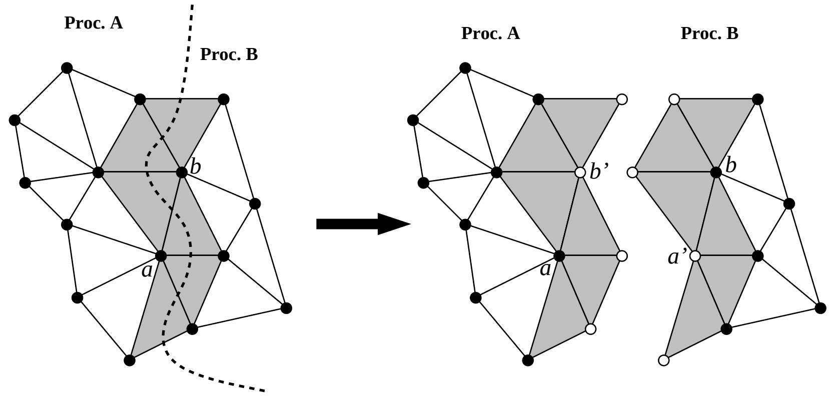

Fig. 3. *Partitioning with one layer of overlapping triangles*

overlapping pattern of one layer of duplicated triangles. A gather-scatter loop on triangles, such as the first loop of figure 2, may gather values from neighbor nodes, but this requires that replicated values on node a' (*resp.* b') of figure 3 be up to date, i.e. identical to node a (*resp.* b). On the other hand, if the resulting value `val2` – which, as a localised scalar, is equivalent to an array on triangles – is scattered to neighbor nodes, this results in out of date values on overlapping nodes a' or b', because these nodes cannot see all of their neighbor triangles. The Overlap Automaton formalises these evolutions from one state to another. Each state specifies two things:

(1) Along which data structures is the flowing data partitioned? This can be on nodes (**Nod**), edges (**Edg**), triangles (**Tri**), or tetrahedra (**Thd**). The flowing data can also be scalar (**Sca**), meaning that it holds a global value for the whole (sub-)mesh. Note that this property is fixed for any given array or loop, e.g. an array on triangles will always hold values aligned on triangles.
(2) Are the overlap elements up to date? We denote this by the subscript of the state name: “0” means everything is up to date, whereas “1” means the layer of duplicate elements is out of date. By extension, $\mathbf{Sca}_0$ denotes a global value coherent on all sub-meshes, while $\mathbf{Sca}_1$ denotes partial reduction values on sub-meshes, requiring a global communication.

The *transitions* between these states come from analyses like the one above. For example again, for a gather-scatter loop on triangles and duplicated triangles (state $\mathbf{Tri}_0$), the required states for gathered values and the resulting states for scattered values are shown on the left of figure 4. The right part of figure 4 shows the analogous transitions for a loop in state $\mathbf{Nod}_1$, i.e. which runs on nodes except replicated ones. For example, if the middle loop in figure 2 is in that state, these transitions specify that node-based array `NEW` may

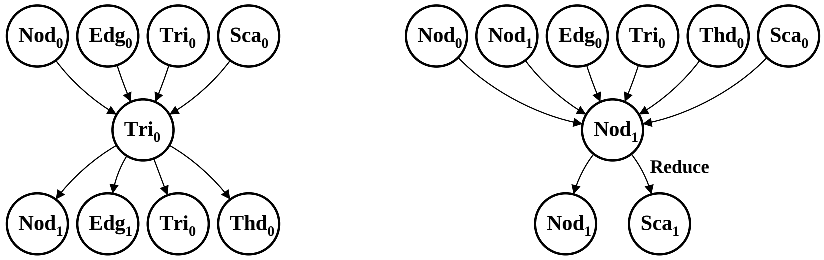

Fig. 4. *State of the flowing data across loops on triangles and nodes*

be up to date or not, and that the result of the reduction `sqrdiff` is a local sum (state **Sca**$_1$), that eventually needs a reduction communication call, to become a global sum.

Similar graphs exist for loops on each mesh entities. We also create the "**Update**" transitions, that represent the updates of the duplicate variables by appropriate communication calls. Merging these elementary graphs gives the Overlap Automaton. Figure 5 shows the Overlap Automata for our chosen overlapping pattern of figure 3, and for the 3D case, with a one tetrahedron wide overlap.

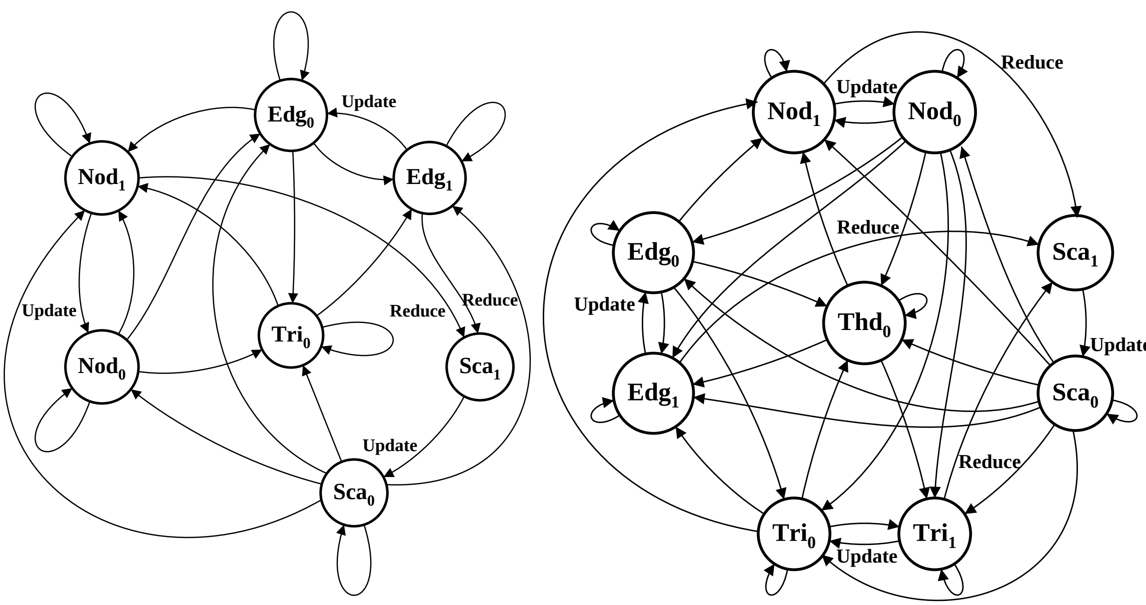

Fig. 5. *Overlap Automata for 2D and 3D meshes with overlap*

The Overlap Automaton is specific to each data organisation and its overlapping pattern. It is independent from the particular program, data, or partition. In principle, it must be provided by the user. However, in the very frequent case of 2D triangular meshes and 3D tetrahedron meshes, these automata are predefined for the most frequent overlapping patterns (cf figure 5).

Suppose now that we 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 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 defines a SPMD program in the following manner:

(1) Each data-dependence mapped to a transition labelled "**Update**", generates a communication between its origin and its destination.
(2) Each loop mapped to a state with subscript 1, must not be run on duplicated mesh elements.

We can show that this SPMD program gives the same results as the sequential program. This is because each possible execution of the program corresponds to one sweep on the data-flow graph (possibly with cycles), and therefore can itself be mapped to the Overlap Automaton. As a consequence, all values remain in a *valid* state throughout all the program's execution, and therefore contain the same data as in the sequential execution.

4 Analysis of the algorithm

The above method leads to an algorithm that sweeps through the data-flow graph, and maps each loop and each variable occurrence to an Overlap Automaton state. The state assigned to a distributed loop is indeed the state for each array indexed by the loop index, and of each localised scalar in the loop. The data-flow graph is generally cyclic, mapping choices are many, and some choices may lead to contradictions. Therefore this sweep must implement a backtracking mechanism. Backtracking occurs when a data-flow arrow can be mapped to no transition, or when a data-flow node or arrow is encountered twice, requiring mapping to two different states or transitions.

This mapping algorithm is similar to the so-called *Simulation* in the calculus of processes [10]. This is known to be exponential with respect to the size of the analysed program. If we call n the number of data-flow nodes, and A the maximum number of alternatives for mapping a data-flow arrow, i.e. the maximum number of transitions from a given state, the complexity of the algorithm is bounded by $O(A^n)$.

However, there exist classical heuristics to remain very far from this upper bound. The general idea is to define recursively nested data-flow sub-graphs, and to run the analysis on each sub-graph independently. Then the sub-graph is considered as a "black box" node at the enclosing sub-graph level. Natural sub-graphs to be considered are loops and subroutines. In principle, each sub-graph must be swept to find *all possible mappings*. Then each of these mappings will be considered at the enclosing level. There is a big speedup if only the "best" mappings are stored and retrieved. This might miss some solutions in very rare cases, but experience shows there is no problem in practice. Here are two heuristics we propose:

- **Pushing communications upwards:** some of the mappings found for a subroutine turn out to insert a communication call just at the beginning or at the end of the subroutine. These solutions are always overridden by another solution, *without* the initial or final communications. This amounts to moving the communication up, into the calling routine, where they make more sense to the user anyway.
- **Partial order on mappings:** it is not always possible to decide one mapping better than another. This is because one mapping can trade more duplicate computation for fewer communications. But there can be a partial order between mappings. Any mapping may be evaluated with two figures. One represents the amount of redundant computations, that depends on the number of loops whose state is labelled 0. The other represents the amount of communication, that depends on the number of "**Update**" transitions. This last figure also uses a communication "overhead", used to promote grouped communications. This gives a partial order on the solutions, and we retain only the best solutions.

5 Application and performances

We had to build a specific tool, called *Gaspard*, to validate our method on 2D and 3D mesh-based applications in FORTRAN. The user is asked to specify, with a specialised interface:

- the program segment to parallelise,
- the Overlap Automaton corresponding to the current geometry of the mesh and of the overlap,
- an assignment from each variable name and loop to the set of its possible states. This set specifies that each of these objects (variables and loops) is *aligned* once and for all on one category of mesh elements (nodes, edges,

triangles, ...), and is partitioned accordingly.

Gaspard returns the set of all possible mappings, i.e. all possible SPMD programs, with communications calls inserted at the correct places.

Our application examples are two-dimensional and three-dimensional flow solvers, written in Fortran 77, named *Exp2D* (13 routines, 1500 code lines), *Imp2D* (18 routines, 1800 code lines), and *NSC3DM* (34 routines, 4800 code lines). They solve the compressible Navier-Stokes equations on unstructured triangular meshes. Time advancing is *explicit* for *Exp2D*, and *implicit* for *Imp2D* and *NSC3DM*, therefore requiring the resolution of a large sparse linear system at each time step. This is done with Jacobi relaxations.

The partitioned meshes have one layer of duplicated elements (*cf* figure 3), and therefore we select the Overlap Automata of figure 5. This choice is justified by the comparison of various overlapping patterns given in [4]. Running *Gaspard* on *Exp2D* and *Imp2D* takes some 10 minutes on a Sun Ultra-1/170. Application to *NSC3DM* takes 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 parallelise the programs by hand.

Each application comes with its test case:

- *Imp2D* computes the external laminar viscous flow around a `NACA0012` airfoil (Mach number: 0.85, Reynolds number: 2000). The 2D mesh contains 48792 nodes, 96896 triangles and 145688 edges.
- *NSC3DM* computes the external flow around an `ONERA M6` wing (Mach number: 0.84, angle of incidence: 3.06°). The 3D mesh contains 15460 nodes, 80424 tetrahedra and 99891 edges.

Although this is more the result of the SMPD tactic itself, rather than a result of our method to insert communications, we show some performance results of the SPMD programs obtained. More can be found in [4]. In the following tables, N_p is the number of processes for the parallel execution; "Elapsed" is the average total elapsed execution time and "CPU" the total CPU time; the parallel speedup $S(N_p)$ is calculated on the elapsed times. Simulations are performed with 64 bit arithmetic computations, on the following computing platforms:

- *(cf table 1)* a `SGI Origin 2000` (located at the *Centre Charles Hermite* in Nancy), equipped with `Mips R10000/195 Mhz` processors with 4 Mb of cache memory. The `SGI` implementation of MPI was used.
- *(cf table 2)* a cluster of 12 Pentium Pro P6/200 Mhz, with a 100 Mbit/s FastEthernet switch. The code was compiled by the G77 GNU compiler with maximal optimization options. Communications used MPICH (version 1.1).

Table 1
Performances on a `SGI Origin 2000` system

	Imp2D on `NACA0012` airfoil			*NSC3DM* on `ONERA M6` wing		
N_p	Elapsed	CPU	$S(N_p)$	Elapsed	CPU	$S(N_p)$
1	2793 s	2773 s	**1.0**	1318 s	1305 s	**1.0**
4	815 s	809 s	**3.5**	390 s	387 s	**3.4**
8	314 s	310 s	**8.9**	198 s	196 s	**6.7**
16	147 s	145 s	**19.0**			

Table 2
Performances on a cluster of 12 `Pentium Pro`

	Imp2D on `NACA0012` airfoil			*NSC3DM* on `ONERA M6` wing		
N_p	Elapsed	CPU	$S(N_p)$	Elapsed	CPU	$S(N_p)$
1	6883 s	6854 s	**1.0**	7488 s	7429 s	**1.0**
4	2289 s	2182 s	**3.0**	2583 s	2382 s	**2.9**
8	1440 s	1218 s	**4.8**	1409 s	1254 s	**5.3**

The results show a good parallel speedup, especially on table 1. The decreasing sizes of the sub-meshes resulting from the 8 and 16 sub-meshes decomposition explain the super-linear 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. On table 2 the speedups are still good, but show the impact of higher communication costs prevailing on the cluster architecture.

6 Conclusion

We proposed a method to mechanise an important step of SPMD parallelisation, namely the placement of communications. From this method, we derived a specialised tool for programs on unstructured meshes, partitioned with an overlap, The tool was tested on fluid mechanics applications.

Performances show that this specialised SPMD strategy actually gives good speedups on real applications. Currently, the tool is able to find a provably correct placement of communication calls, faster than by hand.

A slightly modified version of our method could also be used in “checking” mode. This means verifying an existing placement of communications inside an SPMD application.

The present method and tool, although probably not general, must be applicable to other data structures than meshes. As long as one can define states of coherence of these data on their overlap, one can specify an *Overlap Automaton* that says how the program can use these distributed data. This *Overlap Automaton* only reflects the data *geometry*, and is independent from the particular program, data, or partition.

References

[1] S. Balay, W. Gropp, L.C. McInnes, B. Smith, PETSc 2.0 Users Manual, Technical report ANL-95/11, Argonne National Laboratory, 2000.

[2] H. Bao, J. Bielak, O. Ghattas, L. Kallivokas, D. O'Hallaron, J. Shewchuk and J. Xu, Large-scale Simulation of Elastic Wave Propagation in Heterogeneous Media on Parallel Computers, *Computer Methods in Applied Mechanics and Engineering* 152 (1998) 85-102.

[3] P. Brezany, V. Sipkova, B. Chapman and R. Greimel, Automatic Parallelization of the AVL FIRE Benchmark for a Distributed-Memory System, ESPRIT Project *PPPE* report, 1995.

[4] C. Farhat and S. Lanteri, 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* 119 (1994) 35-60.

[5] L. Hascoët, Automatic Placement of Communications in Mesh-Partitioning Parallelization, *ACM SIGPLAN Notices, Proceedings of PPoPP'97* 32(7) (1997) 136-144.

[6] A. Kneer, E. Schreck, M. Hebenstreit and A. Goszler Industrial mixed OpenMP/MPI CFD application for calculations of free-surface flows *WOMPAT'2000, Workshop on OpenMP Applications and Tools, San Diego, California*, (2000)

[7] C. Koelbel, Compiling Programs for Nonshared Memory Machines, Ph.D. Thesis, Purdue University, 1990.

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

[9] K. McManus, A Strategy for Mapping Unstructured Mesh Computational Mechanics Programs onto Distributed Memory Parallel Architectures, Ph.D. Thesis, University of Greenwich, 1996.

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

[11] M.F.P. O'Boyle, L. Kervella and F. Bodin, Synchronization Minimization in a SPMD Execution Model, *Journal of Parallel and Distributed Computing* 29 (1995) 196-210.

[12] R. Ponnusamy, Y.S. Hwang, R. Das, J. Saltz, A. Choudhary and G. Fox, Supporting Irregular Distributions Using Data-Parallel Languages, *IEEE Parallel & Distributed Technology* 3(1) (1995) 12-24.

[13] A. Sussman, J. Saltz, R. Das, S. Gupta, D. Mavriplis and R. Ponnusamy, PARTI Primitives for Unstructured and Block Structured Problems, *Computing Systems in Engineering*, 3(4) (1993) 73-86.

[14] R.S. Tuminaro, M. Heroux, S.A. Hutchinson, and J.N. Shadid, Official Aztec User's Guide: Version 2.1, Technical Report, Sandia National Laboratories, 1999.

[15] M. Ujaldon, E. Zapata, S. Sharma and J. Saltz, Parallelization Strategies for Sparse Matrix Applications, *Journal of Parallel and Distributed Computing* 38 (1996) 256-266.

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