Irregular Loop Patterns Compilation on Distributed Shared Memory Multiprocessors

Mounir Hahad, Thierry Priol, Jocelyne Erhel

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Irregular Loop Patterns Compilation on Distributed Shared Memory Multiprocessors

Mounir Hahad, Thierry Priol and Jocelyne Erhel

N° 2361

Septembre 1994

PROGRAMME 1

ISSN 0249-6399

INRIA
RENNES

Irregular Loop Patterns Compilation on Distributed Shared Memory Multiprocessors

Mounir Hahad*, Thierry Priol and Jocelyne Erhel

Programme 1 — Architectures parallèles, bases de données, réseaux et systèmes distribués
Projet CAPS

Rapport de recherche n°2361 — Septembre 1994 — 15 pages

Abstract: The Shared Virtual Memory (SVM) is an interesting layout that handles data storage, retrieval and communication. It affords a shared data programming paradigm on an architecture where the physical memory is actually distributed among several nodes. However, minimizing the false sharing and reducing the synchronization remain important issues for parallel efficiency, either at compile time for regular codes or at runtime for irregular ones. This paper addresses irregular loops compilation on Distributed Memory Parallel Computers (DMPCs) that run an SVM. Solutions are introduced to overcome the poor locality that exhibit irregular loops and a case study on an intel iPSC/2 and a Kendall Square KSR1 is presented.

Key-words: Shared Virtual Memory, KOAN, Fortran-S, irregular problems, Runtime compilation.

(Résumé : tsvp)

*e-mail :hahad@irisa.fr

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 99 84 71 00 – Télécopie : (33) 99 84 71 71

Compilation de schémas de boucles irrégulières sur multiprocesseurs à mémoire virtuelle partagée

Résumé : La mémoire virtuelle partagée offre, sur un multiprocesseur à mémoire distribuée, l'abstraction d'un espace d'adressage commun à tous les nœuds de l'architecture. Elle permet ainsi aux processeurs un accès direct transparent à toute donnée partagée, indépendamment de sa localisation physique. Cependant, le faux partage de données et la synchronisation lors de l'accès à des données partagées limitent les performances de tels systèmes. Des solutions sont toutefois envisageables à la compilation lorsqu'il s'agit de problèmes réguliers, et pendant l'exécution pour les problèmes irréguliers. Dans ce rapport, nous proposons un schéma de compilation qui tend à minimiser les effets de la mauvaise localité dans les boucles à accès irréguliers. Ce schéma, appelé *boucles à itérations conditionnelles*, a été testé sur un intel iPSC/2 ainsi que KSR1.

Mots-clé : Mémoire Virtuelle Partagée, KOAN, Fortran-S, problèmes irréguliers, schémas d'exécution.

1 Introduction

Although message passing is the usual programming model of DMPCs, Shared Virtual Memory provides an alternative with a data sharing model on these architectures. Software based SVM (like KOAN [8] and Shiva [6] on the iPSC/2) as well as hardware based ones (like ALLCACHE on the KSR1) are built on top of a paging mechanism : accessing a page which is not present in the local memory awakes the SVM engine which will look for the missing page. Once the page brought, the user's program resumes execution at the faulting instruction. This programming model seems to be simpler for users since communications are handled by the operating system. Despite this easiness, both programming models are faced to the same efficiency problem which is however differently stated : in the message passing programming model, the *data* is distributed among the processors so as to minimize communication (less messages) and in the data sharing model, *computations* are assigned to processors so as to enhance locality (less page faults).

Our contribution within this work aims at enhancing program efficiency in the data sharing model on DMPCs, especially when dealing with indirect accesses to shared arrays. Thus, we will focus our efforts in this paper to loop frameworks such as ℓ :

```
ℓ:  Do i=1,m
        S(L(i)) = S(L(i)) op ...
    EndDo
```

where `S` is a shared array of size n, `L` is an indirection array of size m with $m > n$ and op is a commutative and associative operation. In this study, we suppose that `L` is duplicated among the processors. In very large problems, `L` can be actually shared rather than duplicated since read-only data (as `L`) does not disturb the system performance significantly [5].

On DMPCs, PARTI [9, 4, 7] is one of the most advanced projects in resolving such problems. It has been grafted to several HPF compilers such as FORTRAN-D [10, 11] and Vienna-Fortran[3].

The next section of that paper will go over the context of our study. Section 3 introduces the very heart of our proposal which is the CIL technique. An analytical model is also given in that section. In section 4, an improving extension to the CIL is introduced. The implementation issues, including both automatic code generation and experiments, are discussed in section 5. Experiments are held on an iPSC/2 (our Paragon SVM, called MYOAN, is not yet available) and a KSR1.

2 Problem Statement

Let us first consider a parallel loop with an indirect write access to a shared array, as for instance :

```
ℓ1: Do i=1,n
        S(L(i)) = ...
    EndDo
```

where `S` is a shared vector (i.e. stored in a shared virtual memory region) and `L` represents a permutation σ of the iteration space $\mathcal{I}^n = \{1, \ldots, n\}$ unknown at compile time (problem dependent permutation). That is :

$$\sigma : \mathcal{I}^n \longrightarrow \mathcal{I}^n$$
$$i \longmapsto \sigma(i) = \texttt{L}(i)$$
and $\quad \forall i, j \in \mathcal{I}^n, \quad i \neq j \Longrightarrow \sigma(i) \neq \sigma(j)$

Since `L` is a permutation vector, ℓ_1 is a parallel loop and its iterations can be distributed among the processors according to any pattern, such as '*block*', '*cyclic*' etc... As `L` is unknown at compile time, the access order to the elements of `S` is also unknown. Thus, the only parameter to take into account from a compiler standpoint is to achieve a load balanced distribution of the iterations through the processors. However, the data is stored into memory pages : suppose that two iterations mapped on different processors access elements stored in the same page. Then, that page is moved from one processor local memory to the other's with a '*write*' access right at least once, so that everyone can complete its iteration. This is called a false sharing : two processors access to different addresses located in the same coherency grain (a page) and one of the two accesses at least is a write operation. We may notice that the larger the number of processors that compete for a page is, the more

important is the false sharing phenomenon. Generally, the situation is worse if each processor tries to access that page more than once (ping-pong effect). Depending on `L` values, ℓ_1 is likely to generate a false sharing that usually leads to a drastical loss of performance (in fact, it depends on how fast is the SVM engine).

A more general loop scheme encountered in matrix assembly codes is :

ℓ_2: `Do i=1,m`
`S(L(i)) =` `S(L(i))` *op* ...
`EndDo`

where *op* is an associative and commutative operation (like +, *, etc...) , `S` is of size n and `L` is of size m with $m > n$. Hence, `L` is no more a permutation vector, but a projection of the iteration space, that is :

$\mathcal{P}$: $\mathcal{I}^m \longrightarrow \mathcal{I}^n$
$i \longmapsto \mathcal{P}(i) = \mathtt{L}(i)$
and thus $\exists \;\; i, j \in \mathcal{I}^m / \;\; \mathcal{P}(i) = \mathcal{P}(j)$

Nevertheless, ℓ_2 can be executed in parallel even if it is not a parallel loop provided that each read access to `S` and the corresponding write access (within the same iteration) are atomically executed, preventing any other processor from accessing that element of `S` there in between. This is possible on a shared virtual memory by means of an exclusive page locking mechanism that attributes a page to one and only one processor until it explicitly releases the page lock. So, in addition to the false sharing problem cited above, an overhead is introduced when getting and releasing a lock and a possible loss of parallelism incurs because of mutual exclusion between processors accessing the same page at the same time. Our technique (we introduce in the next section) allows parallel execution of both l_1 and l_2 loops while eliminating false sharing and synchronization needs. Hence, overheads due to page access conflicts and page locking are both avoided.

3 Conditioned Iterations Loop

3.1 Principle

Basically, the idea is that two processors will never try to access the same memory page, at any time. Our proposal is built on what might be called a *Conditioned Iterations Loop* :

Definition 1 *A Conditioned Iterations Loop (CIL) is a loop which iterations are wholly contained into a conditional statement :*

CIL :
```
Do i=1,m
    if (Cond) then
        :
    Endif
EndDo
```

so that if an iteration is executed by several processors, the condition `Cond` *is true on one and exactly one processor (and False on the others)* □

If we assume that `Cond` depends on some data distribution, then the CIL implements the *owner computes* rule on a SVM parallel computer. For example, let us consider the loop :

```
Do i=1,m
   S(L(i)) = S(L(i)) + A(L'(i))
EndDo
```

If we distribute both S and A on the available processors, this loop may be compiled according to the "owner computes" rule on a DMPC, generating a code that looks like :

```
Do i=1,m
   if (I own S(L(i))) then
      if (I own A(L'(i))) then
          S(L(i)) = S(L(i)) + A(L'(i))
      else
          receive A(L'(i)) in tmp
          S(L(i)) = S(L(i)) + tmp
      endif
   else
      if (I own A(L'(i))) then
          send A(L'(i))
      endif
   endif
EndDo
```

In this case, the ownership of a variable is strongly related to the initial mapping of the data on the processors. In a similar manner, the CIL is used on a SVM machine to specify at runtime a particular mapping of the iteration space, in order to minimize the false sharing. Since the coherency grain is the memory page, the objective is that *iterations writing into the same page must be executed on the same processor.* In our sample loop, if `L` and `A` are shared arrays, the loop is compiled on a SVM machine and this code is generated :

```
Do i=1,m
   if (I own the page containing S(L(i))) then
      S(L(i)) = S(L(i)) + A(L'(i))
   endif
EndDo
```

In this context, a processor *owns* a page if it has an exclusive write access to this page. Since it is not allowed to write on pages it doesn't own, the false sharing problem is then eliminated. Notice that thanks to the SVM, the right hand side of the statement in the sample loop doesn't matter : nothing special is done whether the processor in charge of the iteration owns the right hand side operands or not.

During the execution of an application, the page ownership is not a constant function : pages move between processors according to the data access pattern. In our implementation, the user can choose between two ownership functions : the first one returns the actual ownership relation while the second one returns a virtual ownership relation on the basis of a user-defined function.

Example : suppose that we have a 2-processors (PE_1 and PE_2) system with two shared pages (p_1 and p_2). suppose that both pages are in PE_1's local memory with an exclusive write access. The actual ownership function will return `TRUE` to PE_1 for both pages while it will return `FALSE` to PE_2 for any of the pages. By contrast, the user may order the system to *virtually* redistribute the pages among the processors (by block for instance). Then, the virtual ownership function will return `TRUE` to PE_2 about p_2 (and consequently `FALSE` to PE_1 about that second page) even if it is actually in PE_1's local memory. Obviously, a page fault will occur the first time PE_2 will try to access p_2. However, one must compare the cost of knowing the actual distribution and the cost of *cold* or *start-up* page faults. Depending on the architecture, one technique may be cheaper than the other. The aim of virtually redistributing the pages among the processors is to avoid load imbalance that may occur if a large number of pages are owned by a small number of processors. Presently, our implementation handles two virtual redistribution schemes (block and cyclic) in addition to the actual distribution.

3.2 Analytical model of the l_1 loop

Relatively to a block distributed loop (strip-mining), the CIL introduces an overhead due to the execution of all the iterations by all the processors (instead of a loop bounds reduction), and the evaluation of the condition at each iteration. This overhead may be (relatively) too expensive depending on the nature of the computations involved by each iteration. This part of the paper aims at comparing this overhead to other possible and correct compilation schemes. To do so, let us assume that we have the following context :

- our application is running on a DMPC ;
- a coherent page-based SVM is provided on the DMPC ;

and the problem is to decide which compilation technique performs the best execution time among the most common ones :

1P : execution of the ℓ_1 loop on a single processor (no parallelism but no overhead as well) ;

SM : strip-mining : block partitioning of the iteration space among the processors ;

CIL : execution of the ℓ_1 loop according to a regular block distribution of `S` pages.

On a sequential computer the execution time of the ℓ_1-loop is :

$$T_{seq} = N(T_{rhs} + 2T_{mem}) \tag{1}$$

where T_{rhs} stands for the right hand side evaluation time and T_{mem} is the cost of one memory access. Indeed, if we assume that i is stored in a register, only two memory accesses are necessary by iteration, one access to read $L(i)$ and one to store $S(L(i))$.

In our context, we suppose the pages of `S` to be block distributed among P processors and `L` to be a local array. Then, the execution time for 1P includes page write faults and page migration (swap) if the local memory is not large enough to store `S`. Hence,

$$T_1 \quad \geq \quad N(T_{rhs} + 2T_{mem}) + \left(\left\lceil \frac{N}{s} \right\rceil - \left\lceil \frac{\lceil N/s \rceil}{P} \right\rceil \right) . T_{pf} + \sup \left(\left\lceil \frac{\left\lceil \frac{N}{s} \right\rceil - \left\lceil \frac{\lceil N/s \rceil}{P} \right\rceil - E}{U_{mig}} \right\rceil , 0 \right) . T_{mig} \tag{2}$$

where the parameters stand for :

N **:** loop bound

T_{rhs} **:** right hand side evaluation time

T_{mem} **:** one memory access delay

s**:** one page size (number of `S` elements in a page)

P **:** number of processors

T_{pf} **:** page fault service time

E**:** initially available local memory for `S` pages

T_{mig} **:** page migration service time

U_{mig} **:** number of pages migrated per swap operation

The execution time for SM is :

$$T_{SM} = \left\lceil \frac{N}{P} \right\rceil (T_{rhs} + 2Tmem) + \alpha . T_{pf} + \ \beta . T_{mig} \tag{3}$$

where α stands for the number of page faults and β the number of page migrations. These two parameters depend to a great extent on the runtime values of `L`.

As to the CIL, the execution time includes a preceding phase that computes the upper and lower bounds of chunks of `S` virtually attributed to each processor depending on the window of `S` defined in ℓ_1-loop (in case `S` has more than N elements). Thus, it is :

$$T_{CIL} = C_0 + NC_1 + (T_{rhs} + 2T_{mem}) \times \inf \left(\left\lceil \frac{\lceil N/s \rceil}{P} \right\rceil . s, \left\lfloor \frac{\lceil N/s \rceil}{P} \right\rfloor . s + N \bmod (P.s) \right) \tag{4}$$

where C_1 is the time necessary to evaluate the condition. Recall that there is no page fault (nor page migration) in this case. on the other hand, the CIL may introduce a slight load imbalance depending on the page size.

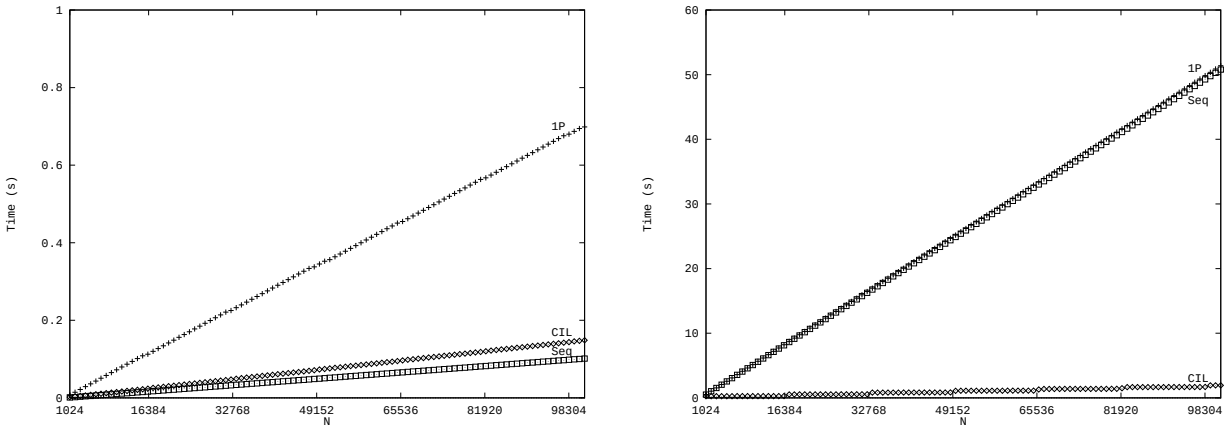
Parametrization

In order to reduce the number of parameters that rule our analytical model, we drop the page migration problem. In other words, we assume that each processor has enough local memory to store the whole of its working set. Although this assumption does not introduce any restriction for SM and CIL, it seems to be quite unrealistic for 1P indeed. Fortunately, in our experiments, an asymptotical behavior has been reached still the problem sizes we used fit in a single processor's memory. Consequently , we assume that $E > \lceil \frac{N}{s} \rceil - \left\lceil \frac{\lceil N/s \rceil}{P} \right\rceil$ in equation 2 and $\beta = 0$ in equation 3.

Experiments held on a 32 nodes Intel iPSC/2 (+KOAN) and a 32 nodes Kendall Square KSR1 machines led to the parametrization depicted in table 1. According to these values, the execution times prediction of the compilation techniques for various loop sizes is done. Figures 1 and 2 report this prediction in two cases : firstly, only a store operation is assumed to be executed by iteration and secondly, we assume that the right hand side calls a 0.5ms delay function. 'Seq' stands for an execution in a pure sequential environment. Setting aside the exception of the KSR1 with 1 memory access as a right hand side per iteration, the CIL performs fairly well on both architectures. The speedup computed for CIL with 0.5ms per rhs is roughly equal to 30. As regards the KSR1, the most striking fact is that a subpage fault service time (7.5μs) is a very low overhead. We must emphasize that over 32 processors, the subpage fault service time becomes 28.5 μs at least.

	$C_0(\mu s)$	$C_1(\mu s)$	$T_{mem}(\mu s)$	s	$T_{pf}(\mu s)$
iPSC/2	14.36	1.43	0.5	512	2995
Ksr	4.62	0.87	0.9	16	7.5 - 30

Table 1: Analytical model parametrization



(a) rhs = 1 local memory access (b) rhs = 0.5ms delay

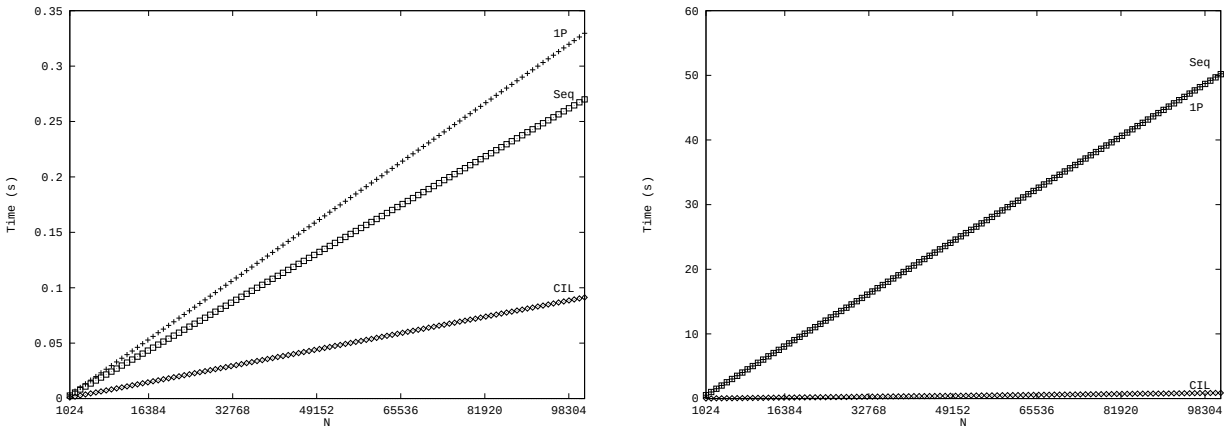
Figure 1: Predicted execution times on iPSC/2 + KOAN (32 procs)

Because the execution time of SM depends on an unknown parameter which is the number α of page faults occurred during the execution, the figures cited above do not include SM curves. As evidence, α depends on the runtime values of `L`. Our analytical model leads to the following overestimation of α, over which CIL performs better than SM :

$$T_{SM} \leq T_{CIL} \Longrightarrow \alpha \leq \alpha_o = \frac{(T_{rhs} + 2T_{mem})\left(\frac{N+s}{P} + s + N\%(Ps)\right) + C_0 + NC_1}{T_{pf}}$$

This leads to the conclusion that no more than 5 accesses per ten thousands should be remote accesses on the iPSC/2, with 1 local access as a rhs and 32 processors, for SM to perform as well as CIL. If we consider 0.5ms per rhs, α_o is equal to 5 remote accesses per thousand. Since irregular applications exhibit less locality than that, we can strongly infer that CIL performs far better than SM.

Regarding the KSR1, the situation is not so clear. In this case, α_o is equal to 11.7 per cent and 220 per cent for 1 local memory access and 0.5ms delay as a rhs respectively. If the former value doesn't allow any



(a) rhs = 1 local memory access (b) rhs = 0.5ms delay

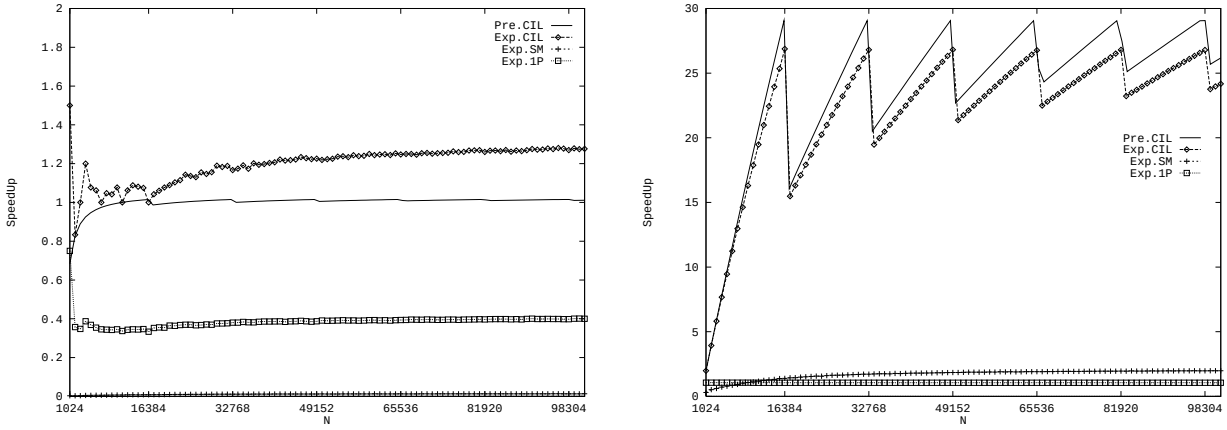
Figure 2: Predicted execution times on KSR/1 (32 procs)

conclusion, there is no doubt about the latter one : it obviously states that SM performs actually better than CIL. Nevertheless, we must point out that for very small values of N (1024), CIL reaches the near maximum efficiency on the KSR1 (0.93). So, in view of all, we may assume that, among the three tested, CIL is the most interesting approach even if the KSR1 (up to 32 processors) is an attractive machine.

Experimental validation

To account for the known elements, it has been estimated that CIL performs better on the iPSC/2 while no strong conclusion was inferred on the KSR1 for costless right hand sides. So as to fade any doubt about it, we experimentally evaluate the three compilation schemes. In addition, experience is useful to know how close is our simple analytical model to the reality.

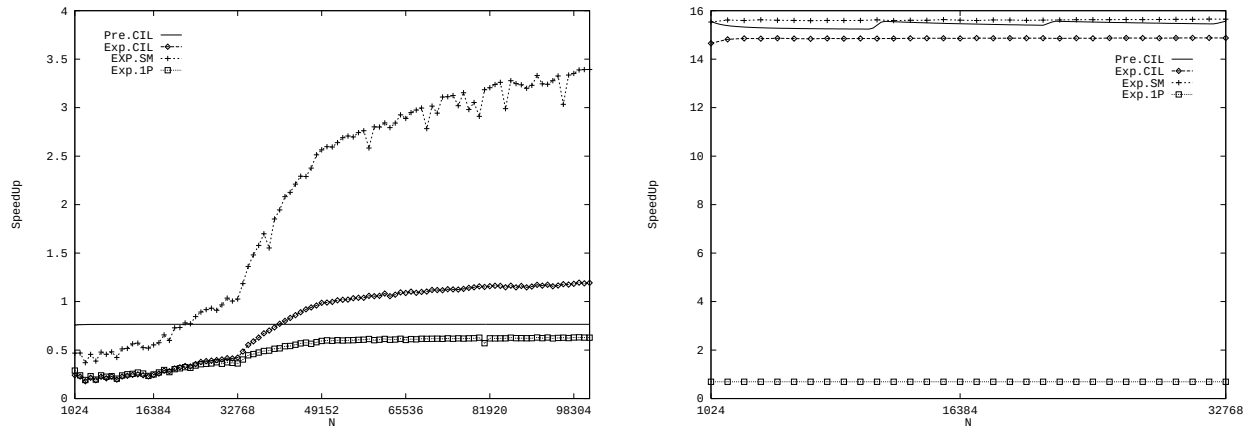
For purposes of experience, a random permutation σ is generated and stored into `L` for different values of N. Then, the ℓ_1 loop is executed according to the CIL and SM schemes with the same permutation. Notice that since there is no page migration, the permutation is not important with respect to the execution time of 1P. The experiments conducted on the iPSC/2 confirm our prediction as it is shown on figure 3.



(a) rhs = 1 local memory access (b) rhs = 0.5ms delay

Figure 3: Experimental and predicted SpeedUps of ℓ_1 on the iPSC/2 (32 procs)

On the contrary, the experiments held on the KSR1 showed that the first level cache (which is not taken into account in our model) plays an important role. This is why the measured CIL speedup is not so close to its corresponding prediction for the costless right hand side. Furthermore, the SM scheme seems to perform better than the CIL one for these costless right hand sides. Regarding the expensive right hand sides, the prediction was right. Figure 4 bears witness to this.



(a) rhs = 1 local memory access (b) rhs = 0.5ms delay

Figure 4: Experimental and predicted SpeedUps of ℓ_1 on the KSR1 (16 procs)

4 Optimizing CIL

As already mentioned, CIL suffers from the fact that all the processors investigate all the iterations. This operation prevents from scalability and sometimes from efficiency (on the KSR1 for instance). To alleviate this problem, we use a learning technique that takes advantage from the first iteration to enhance the efficiency of the following iterations. The model loop is :

```
Do t=1, ntimes
      Do i=1,m
         S(L(i)) = S(L(i)) + ...
      EndDo
EndDo
```

The inner loop can be processed by CIL and the outer loop can be used to collect information about the data access pattern that does not change from one iteration to another (the LEARN loop). So, the model loop cited above is transformed into the following code :

```
t=1
j=0
Do i=1,m
      if (I own the page containing S(L(i))) then
         S(L(i)) = S(L(i)) + ...
         j = j+1
         MyIteration(j) = i
      endif
EndDo
Do t=2, ntimes
      Do ij=1,j
         i = MyIteration(ij)
         S(L(i)) = S(L(i)) + ...
      EndDo
EndDo
```

The code showed above is executed by all the processors and `MyIteration` is a private array. Thanks to this array, the processors do not need to examine all the iterations anymore : this is done only once at the first iteration of the outermost loop (the learning loop). Consequently, the time overhead is reduced while a storage overhead of size $o(\mathtt{m}/\textit{number of processors})$ is needed. In fact, that overhead may be more or less large depending on the actual/virtual page ownership distribution and the actual values in `L`. If the user has some knowledge

about these two parameters (for a given problem), he can insert a directive to allocate a more suitable additional storage.

5 Implementation

We address now issues related to the automatic code generation of the CIL and LEARN, and then we present some experimental results on the iPSC/2 and the KSR1.

5.1 Code Generation

In this section, we describe the compile-time support for our optimizations. Indeed, high level user-inserted annotations allow automatic code generation of both CIL and LEARNing techniques. The compiler used is *Fortran-S* [1]. The source code is a standard (sequential) Fortran77 code where annotations have been inserted to express parallelism : which data is shared, which loop is parallel, etc... The output is a Fortran SPMD code targeted to the desired parallel SVM machine. At present, the supported machines are the Intel iPSC/2, the Kendall Square KSR1 and the Intel Paragon XP/S.

When the user decides that a loop must be executed according to the CIL scheme, he inserts an annotation before that loop in his source program, as for instance :

```
C$ann[CIL(STATIC,BLOCK)]
Do i=1,m
   S(L(i)) = S(L(i)) + A(L'(i))
EndDo
```

Fortran-S generates automatically the corresponding code. In this example, the user has chosen a virtual block distribution of the pages of `S`, and thus, the iterations of the loop will be executed on the adequate processors. Two other options are provided :

`C$ann[CIL(STATIC,CYCLIC)]`
which indicates a virtual cyclic distribution of the pages of `S`
`C$ann[CIL(DYNAMIC)]`
to use the actual distribution of the pages of `S`.

To take advantage of the LEARNing technique, the user inserts two annotations, as for example :

```
C$ann[Learn(1)]
Do t=1, ntimes
C$ann[CIL(STATIC,BLOCK)]
      Do i=1,m
         S(L(i)) = S(L(i)) + ...
      EndDo
EndDo
```

where the parameter 1 for the `Learn` annotation indicates which inner loops to investigate in case there are several ones.

5.2 Experiments

To show the impact of the learning technique on the CIL implementation, we carry out some experiments on both the iPSC/2 and the KSR1. l_2 is used as a loop model with a right hand side of 0.5 ms delay for the iPSC/2 and 1us delay for the KSR1 at each iteration, to take into account the communication to computation ratio of each machine. The indirection array L is built on the basis of data arising from irregular triangular meshes. Three mesh problems with different sizes are considered :

Problem	nodes	triangles	# write access
16K	16384	31976	95928
60K	60590	119332	357996
200K	207691	411380	1234140

Both a renumbered version (with a greedy algorithm that enhances spatial locality [2]) and a randomly mixed version of each problem are used. '-r' is appended to the names of the renumbered versions and '-m' is appended to the mixed ones.

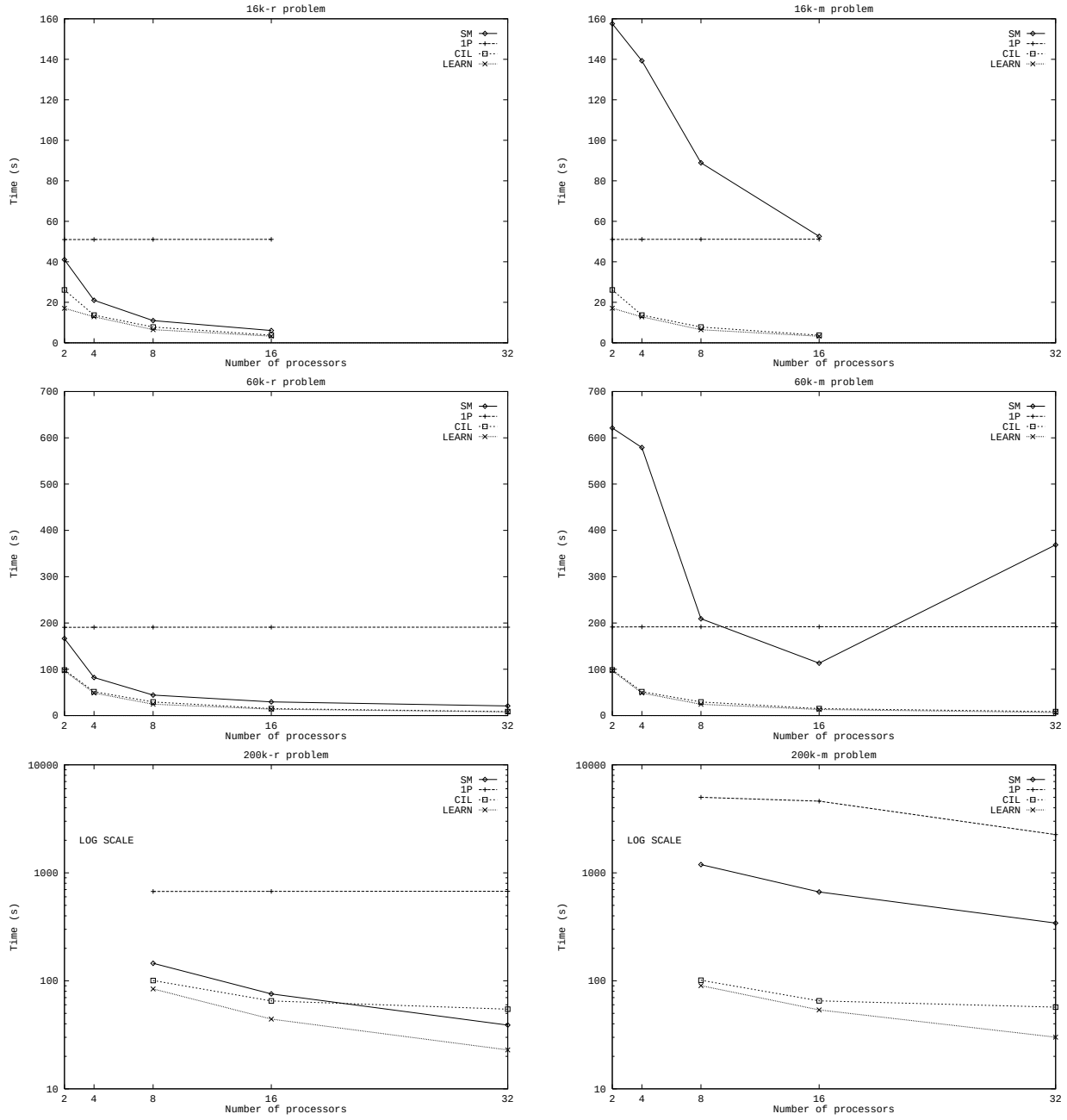


Figure 5: Experimental times of the ℓ_2 loop on the iPSC/2

Figure 5 shows the timing results obtained on the intel iPSC/2. It should me mentioned that no experiments were carried out on 32 processors for the 16k problem because of its too small size (the results wouldn't have much significance). On the contrary, the 200k problem is too large to run on less than 8 processors (not enough memory).

As we can notice, the SM technique always performs better on the renumbered data than on the mixed ones. Figure 7 helps to state that it is due to the number of write page faults that occurred in each case : the mixed data exhibit much less locality than that renumbered ones.

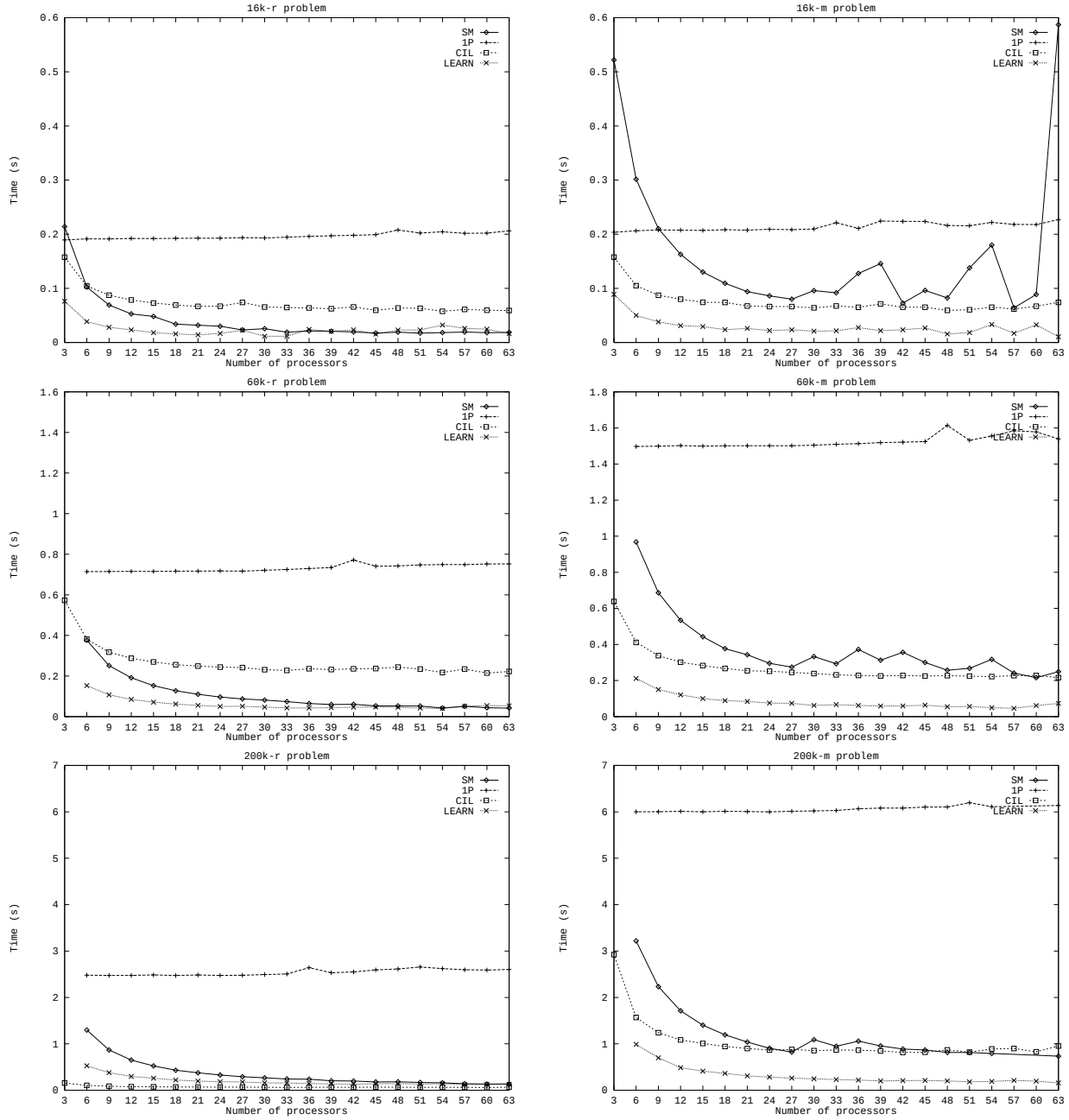


Figure 6: Experimental times of the ℓ_2 loop on the KSR1

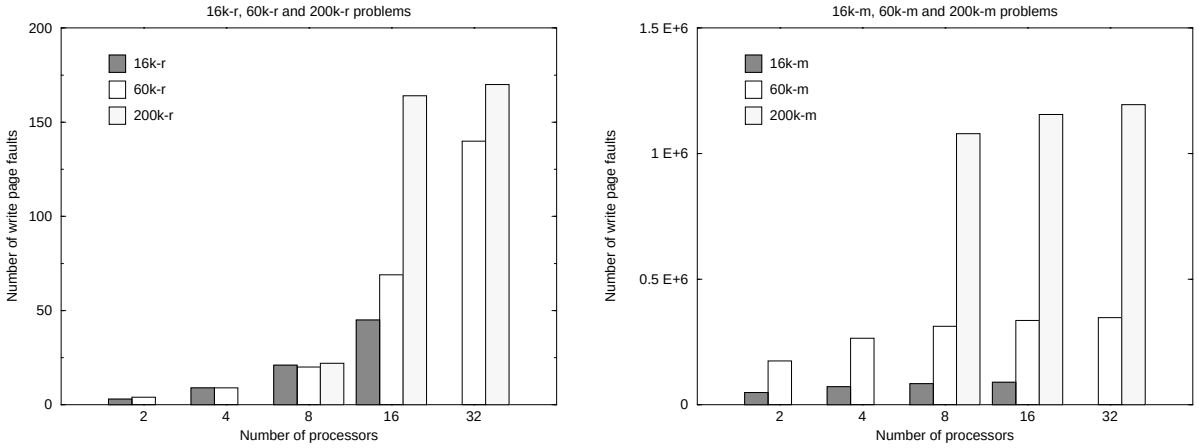


Figure 7: Page faults on the iPSC/2

Figure 5 also shows that the 200k problem is the only one for which 1P does not execute in the same time for both the renumbered and the mixed data. This is because the size of one processor's local memory is too small to hold both `S` (since it is wholy accessed by one processor) and the private array `L`. In fact, it was necessary for us to implement `L` as a shared array too (for the 200k problem). When the processor in charge of the execution of the loop has no more local memory and needs to bring a new page, the SVM protocol on the iPSC/2 flushes a page on another processor's memory, freeing some local space for the new page. This *page migration* is done according to a *Least Recently used page* policy. Table 2 depicts the number of page migrations that occurred during 1P execution for both 200k-r and 200k-m.

Problem	on a read fault			on a write fault		
	8	16	32	8	16	32
200k-r	1409	1510	1561	355	380	393
200k-m	na	179789	74189	na	190903	75734

Table 2: Page migration statistics for 200k-r and 200k-m with 1P.

As regards to the CIL, it almost always performs better than 1P and SM but with 200k-r on 32 processors. This is due to the sharing of `L` that incurs page migration. While there is no page migration at all with SM, CIL causes 6695 page migrations on 8 processors, 12983 on 16 and 25559 on 32. We expect the same problem to raise for LEARN with larger problems than those presented here. However, we also expect those problems to be run on larger memories than ours (1.5Mb shared per node). Despite all these problems, it should be emphasized that the LEARNing technique pays off fairly well the first iteration since it always performs better than the other techniques. The times presented for LEARN in all curves stand for one iteration mean time (experiments are carried out with 10 iterations of the outermost loop). We should also note that the larger the data set is, the better is the improvement. During the execution of LEARN with 200k-m on 32 processors, more than 10s per iteration are spent migrating `L` pages. Even though, the technique is the fastest.

On the KSR1, it is somehow different. It is difficult to carry out precise measures because of the sharing of the hardware statistics counters with processes that are not issued from our application. Hence, the user cannot sift its own page faults from the ones caused by an external system activity or even another application running on separate nodes but using his nodes' local memory.

Putting aside these problems, figure 6 shows timing results on the KSR1. Unlike the iPSC/2, the first level cache (subcache) of the KSR1 plays an important role. With roughly the same number of subpage faults, 1P takes much more time to execute when the data are mixed than when they are renumbered. The table 3 summarizes the miss ratios (number of subcache misses / total number of accesses) with 1P.

Thanks to a relatively costless subpage fault service time, SM performs better on the KSR1 than CIL, as long as there are only few page faults. As the mixed data cause a large number of subpage faults (two orders of magnitude higher than the renumbered ones), CIL becomes more advantageous than SM, yet each processor

Problem	16k	60k	200k
-r	7.4 %	7.4 %	7.4 %
-m	12 %	55 %	56 %

Table 3: Data subcache miss ratio with 1P on the KSR1.

scans the whole iteration space. Particularly interesting is that LEARN remains the most profitable technique with no more than 10 iterations. This obviously suggests that the learning is well written off.

6 Conclusion

In this paper, we were interested in compiling irregular access loop patterns on distributed shared memory multiprocessors. As briefly mentioned in the introduction, this study is based on previous works that aimed at providing a SVM mechanism on the iPSC/2 hypercube and a compiler (Fortran-S). Special attention is given to loop frameworks that are encountered in matrix assembly codes arising from unstructured meshes.

To execute efficiently these loops on a distributed shared memory multiprocessor, we introduce the CIL scheme which maps the iteration space on the processors according to page ownership criteria. The CIL preserves the sequential order of the iterations and reduces significantly the data transfers. An analytical model is presented and experiments are carried out on both an intel iPSC/2 and a KSR1. Furthermore, a learning technique is introduced to improve the CIL performance. This technique takes the first iteration to account and collects data about the access pattern. Then, this pattern is used to narrow the iteration space of each processor. The experimental results show that the latter technique outdoes more conventional compilation schemes.

It is needles to say that the learning technique becomes expensive in terms of memory storage requirements if there are too many indirect write access statements per iteration. Our future work aims at reducing that storage overhead and extending the CIL to a whole code to allow page-affinity mapping over multiple code phases.

References

[1] François Bodin, Lionnel Kervella, and Thierry Priol. Fortran-s : a fortran interface for shared virtual memory architectures. In *Supercomputing'93*, November 1993.

[2] Marie Odile Bristeau, Jocelyne Erhel, Philippe Feat, Roland Glowinski, and Jaques Periaux. Solving the helmotz equation at high wave numbers on a parallel computer with a shared virtual memory. *International journal of supercomputer applications and high performance computing*, (9.1), 1995.

[3] Barbara Chapman, Piyush Mehrotra, Hans Moritsch, and Hans Zima. *Dynamic Data Distribution in Vienna Fortran.* Technical Report 93-92, ICASE/NASA LRC, December 1993.

[4] R. Das, R. Ponnusamy, J. Saltz, and D. Mavriplis. *Distributed Memory Compiler Methods for Irregular Problems - Data Copy Reuse and Runtime Partitioning.* Technical Report 91-73, ICASE - NASA Langley RC, September 1991.

[5] Mounir Hahad, Jocelyne Erhel, and Thierry Ppriol. Scheduling strategies for parallel sparse cholesky factorization on a shared virtual memory parallel computer. *International Conference on Parallel Processing*, 3:290–297, August 1994.

[6] Kai Li and Richard Schaefer. A hypercube shared virtual memory system. *International Conference on Parallel Processing*, 1:125–131, August 1989.

[7] Ravi Ponnusamy, Joel Saltz, Alok Choudhary, Yuan-Shin Hwang, and Geoffrey Fox. *Runtime Support and Compilation Methods for User-Specified Data Distributions.* Technical Report 93-99, ICASE - NASA Langley RC, December 1993.

[8] Thierry Priol and Zakaria Lahjomri. Experiments with shared virtual memory and message-passing on an ipsc/2 hypercube. *International Conference on Parallel Processing*, 2:145–149, August 1992.

[9] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, 8:303–312, 1990.

[10] Chau-Wen Tseng. *An Optimizing Fortran D Compiler for MIMD Distibuted Memory Machines.* PhD thesis, Rice University, January 1993.

[11] Reinhard van Hanxleden, Ken Kennedy, and Joel Saltz. *Value-Based Distributions in Fortran D.* Technical Report CRPC-TR93365-S, Rice University, February 1994.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399