INRIA

UNITÉ DE RECHERCHE
INRIA-RENNES

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

1992

25ème anniversaire

N° 1634

Programme 1

Architectures parallèles, Bases de données,
Réseaux et Systèmes distribués

TRADE-OFFS BETWEEN SHARED VIRTUAL MEMORY AND MESSAGE-PASSING ON AN iPSC / 2 HYPERCUBE

Thierry PRIOL
Zakaria LAHJOMRI

Mars 1992

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTEMES ALEATOIRES

Campus Universitaire de Beaulieu
35042 - RENNES CEDEX FRANCE
Tél. : 99 84 71 00 - Télex : UNIRISA 950 473 F
Télécopie : 99 38 38 32

Trade-offs Between Shared Virtual Memory and Message-passing on an iPSC/2 Hypercube

Mémoire virtuelle partagée ou échange de messages sur un hypercube iPSC/2 ?

Thierry PRIOL and Zakaria LAHJOMRI
IRISA/INRIA
Campus de Beaulieu 35042 Rennes Cedex
tel: 99 84 72 10
e-mail:priol@irisa.fr

February 11, 1992
26 pages
Publication Interne n°637 - Programme I - Projet PAMPA

Abstract

This paper presents the results of an experiment which evaluates the performance of shared virtual memory running on a distributed memory parallel architecture (iPSC/2 hypercube). Two parallel versions of the modified Gram-Schmidt algorithm are presented. Each uses two disparate distribution schemes: data distribution; and control distribution. In the first case, cooperation between processes is achieved by sending messages, whereas shared variables are applied in the second case. This latter technique is made possible by implementing a shared virtual memory as part of the operating system. We present briefly an example of such a mechanism, called *KOAN*, which has been designed specifically for the iPSC/2 hypercube. This shared virtual memory allows us to compare experimentally the performances of the two distribution schemes. Theoretical execution models are also addressed in order to extrapolate expected performance on a large number of processors.

Résumé

Ce papier présente des résultats d'expérimentation d'un dispositif de mémoire virtuelle partagée sur une architecture à mémoire distribuée (l'hypercube iPSC/2). Deux versions parallèles de l'algorithme de Gram-Schmidt modifié sont présentées. La première est fondée sur une parallélisation orientée par les données où la coopération entre processus est réalisée à l'aide d'échange de messages. La seconde exploite une parallèlisation orientée par le contrôle où les processus communiquent par variables partagées. Cette deuxième technique est rendue possible grâce à la mise en œuvre d'un dispositif de mémoire virtuelle partagée. Nous présentons brièvement un exemple d'un tel dispositif, appelé KOAN, qui a été réalisé pour un hypercube iPSC/2. Ce dispositif nous permet ainsi de comparer les deux schémas de parallèlisation. Des modèles d'exécution sont présentés afin d'extrapoler les performances de ces algorithmes sur un nombre important de processeurs.

Keywords: **Shared Virtual Memory, modified Gram-Schmidt algorithm, Hypercube iPSC/2, Distribution schemes**

1 Introduction

Distributed memory parallel computers (DMPCs) seem to be the route to building teraflop computers in the near future. Research in the design of parallel algorithms for these machines have shown that achieving high performance is feasible in most cases. However, programming these machines is still awkward due to the lack of a convenient programming environment including operating systems and compilers. The operating systems generally perform some low level functions like process and message management. Consequently, most of the experiments are based on a message-passing programming model. There are several available implementations of this model such as blocking, non-blocking or rendez-vous, etc.; sometimes the semantic is fuzzy or unknown. As a result, the programming of DMPC's is not an easy task. Since few years, researches in the field of distributed operating system have shown that implementing a shared programming model on distributed architectures is feasible [12]. The implementation of the concept of shared virtual memory (SVM) has been intensively studied. However, few efforts have been made in comparing the advantages and drawbacks of both message-passing and shared memory programming models for DMPC's. In an attempt to compare these programming models, we have conducted several experiments in the design of parallel algorithms for one of the most well known algorithm in computer graphics: the ray-tracing algorithm. These studies have allowed us to compare two different distribution schemes: data oriented distribution with a message-passing programming model and control oriented distribution with a shared memory programming model.

The first approach consists of partitioning the data domain (as for instance: matrices, vectors, lists, trees) of the algorithm into sub-domains, each of which is associated with a processor. Computations are assigned either statically to processors which own the data used by these computations or dynamically by moving the data between processors. Communication between processes is by means of messages. This sort of distribution is exploited in several experimental programming environments for DMPC's [10, 14, 15, 16, 18, 19]. The user must express his data distribution by using pragmas, and a compiler is then able to generate processes and the communication patterns.

The second approach focuses on the parallelization of loops. Loops are split into parts that can be executed in parallel. Each part is assigned statically or dynamically to a processor. Data accesses are managed through a shared memory. This technique is mostly used on shared memory parallel computers such as those made by Cray Computer. Several Fortran compilers exploit control distribution for parallelizing algorithms [1, 24].

These two approaches can be applied to various algorithms. For instance, the results we obtained from running the ray-tracing algorithm reveale that, in all cases, the best speedup is achieved by using control distribution together with shared virtual memory programming model [2, 3]. This is mainly due to the facts that the database is accessed in read-only mode, and that control oriented distribution facilitates load balancing. As a matter of fact, the relationship between computation and data is unknown; therefore it is difficult to map data on processors for the purposed of having a load balance when using data oriented distribution.

The purpose of this paper is to compare these two distribution schemes on a numerical algorithm that has both write and read access to data. To keep the scope of the article within reasonable limits, we will fix our attention on the modified Gram-Schmidt algorithm that creates an orthogonal set of vectors of a given matrix [9]. A systolic version of this algorithm has been studied in [21, 22] and implemented on a iPSC/2 hypercube. Their results could be compared with ours. This paper is organized as follow: section 2 sketches briefly the concept of shared virtual memory. An example of an implementation on an iPSC/2 hypercube is given. Section 3 describes the modified Gram-Schmidt algorithm whereas section 4 and 5 introduce two parallel algorithms. Both experimental and theoretical results are provided. Section 6 compares the two parallel algorithms from which we derive an optimization we describe in section 7.

2 SVM backgrounds

A Shared Virtual Memory (SVM) provides to the user an abstraction from an underlying memory architecture. This concept [12] is somewhat similar to the one which is currently used on classic mainframe computers. However, it differs in the fact that this virtual memory is shared by several processors. It provides a virtual address space that is shared by a number of processes running on different processors of a distributed memory parallel computer. In order to distribute the virtual address space, the SVM is partitioned into pages[1] which are spread among local processor memories. Each local memory is subdivided into two parts. The first one is used for storing the code of processes and their local variables whereas the second part acts as a large software cache for storing pages as shown in figure 1. At any time a processor may have a page with *no access*, *read access*, or *read&write access*. Pages in read access can be replicated in the local memories of many processors at the same time. Since several copies of a page may exist, they must be kept coherent if a processor want to modify a page (i.e. to have a read&write access). This is done by using a *cache coherence protocol*. One of the most well known is the *invalidation* protocol: a node can have read&write access to a page if it has the only copy of the given page, and it can have a read access if multiple copies of the given page exist on different nodes.

When a memory reference causes a fault on a page of the SVM (i.e. when a process violates access rights), a dialog is established between the faulting node and the owner of the page (i.e. the last one that have a read&write access on the page), and some actions are executed in other to maintain the memory coherent: the owner of the page sends a copy to the faulting node, changes its local access rights to read access in the case of read page fault, otherwise (i.e. write page fault) it sends an invalidation message to all other nodes with a copy of the given page and changes its local access rights to no access. On receipt of an invalidation message, the copy of the given page is destroyed.

Since the size of the physical memory on a processor is much less than the size of the SVM, the part of the local memory which acts as a cache is managed according to a LRU (Least Recently Used) policy. Any non-owned page can be discarded away from the cache. However an owned page cannot be flushed from the cache; it requires an ownership page migration in another physical memory or in other storage devices if any are used.

There are several implementations of the concept of SVM. They concern only high latency networks which link together several workstations [12, 17, 5, 6, 23, 20]. Unfortunately, there are few implementations on hypercube or 2D-mesh distributed memory parallel computers (DMPC). Shiva [13] seems to be the only attempt to implement a SVM on a hypercube multicomputer (Intel iPSC/2). However, it appears that this implementation has been done at the user's level without modifying the iPSC/2 operating system and consequently add a lot of overhead. Therefore, comparing different programming models (message passing or shared variables) with DMPC cannot be really conducted. KOAN is an implementation of a SVM embedded in the NX/2 operating

[1] the granularity afforded by hardware virtual memory

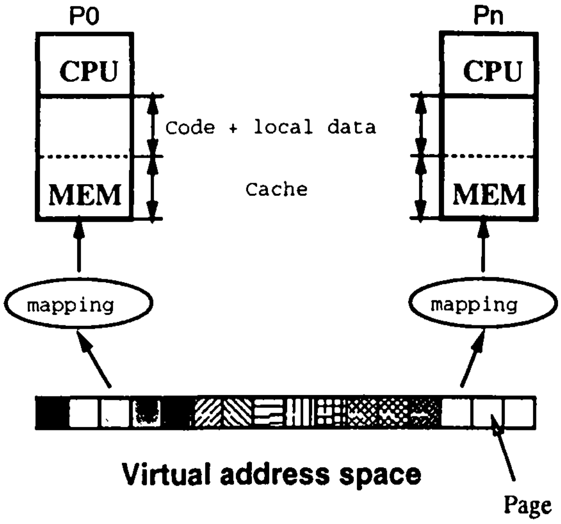


Figure 1: Mapping a Shared Virtual Memory on DMPC.

system for the hypercube iPSC/2. It does not address a new SVM algorithm, as such algorithms are well known. We have chosen the *Fixed Distributed Manager* [12] where each processor uses a known manager for each page to locate the current owner of the page and we have modified with the aim of implementing it on a Intel iPSC/2. Several functionalities have been added such as several different cache coherence protocols, prefetching techniques and trace analysis tools. Implementation details can be found in [11].

3 The Modified Gram-Schmidt algorithm

Given a set of independent vectors $\{v_1, ..., v_n\}$ in $\mathbb{R}^m$, the Modified Gram-Schmidt algorithm produces an orthonormal basis of the space generated by these vectors [9]. As shown in figure

```
for $i = 1 : n$
    (* $v_i$ = new vector *)
    $v_i = v_i \;/\; \| v_i \|_2$
    for $j = i + 1 : n$
        (* Orthogonal projection of $v_j$ on $v_i^T$ *)
        $v_j = v_j - (v_i^T . v_j) * v_i$
    end
end
```

(a)

```
for $i = 1 : n$
    (* Normalization *)
    $[i, i]$
    for $j = i + 1 : n$
        (* Correction *)
        $[i, j]$
    end
end
```

(b)

Figure 2: (a) : Modified Gram-Schmidt algorithm (MGS). (b) : Simplified MGS algorithm.

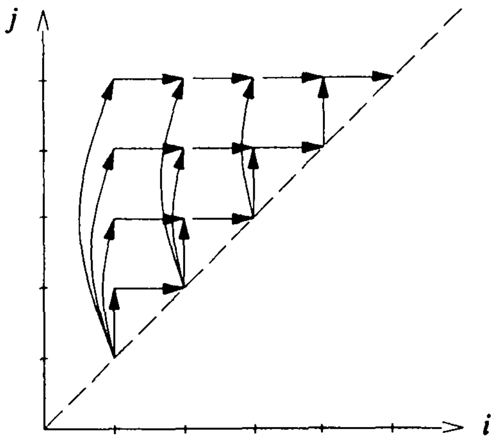


Figure 3: Graph dependency for the MGS algorithm

2, the basis is constructed step by step, each new computed vector replaces the old one. This algorithm can be be simplified by using the following notation given in [22] (figure 2)). $[i,i]$ means that vector i is normalized ($v_i = v_i/\|v_i\|_2$) and $[i,j]$ means that vector j is corrected with vector i ($v_j = v_j - (v_i^T.v_j) * v_i$).

The number of floating point operations N involves in this computation is:

$$N = 2n^2m + nm + n$$

where n is the number of vectors and m is the vector size. Consequently, if τ_a is the average time for computing a floating point operation on a processor, the sequential time is equal to:

$$T_{seq} = N\ \tau_a$$

The dependence graph is illustrated in figure 3. Axis i may be seen as the time and axis j as the processors. This algorithm has potential medium-grain parallelism. Indeed, each correction (i.e. $[i,j]$) can be computed in parallel. Since each of them has the same number of computation, load balancing can be easily achieved. Let us now describe two general approaches for implementing parallel algorithms on DMPCs.

4 Data oriented distribution

Data oriented distribution has been the suitable technique for designing a parallel algorithm on DMPCs since it takes into account the distributed framework of the memory. This approach has been intensively studied for parallelizing several factorization algorithms [7]. Since it is a well known approach, the aim of this section has a limited scope: it provides readers with both experimental and theoretical results, that will be used for comparison with a shared virtual memory approach.

Let us describe how data oriented distribution can be applied on the modified Gram-Schmidt algorithm. Computations are distributed according to both a data mapping function and a rule that establishes on what processors they are executed. The data mapping function must be chosen carefully since it allows the load to be balanced among the processors. This is discussed in the

```
for $i$ = $1, n$
   if I am the owner of vector $i$ then
      $[i, i]$
      broadcast vector $i$ to all processors
   else
      receive vector $i$
   endif
   for all vector $j$ such as $j > i$
      if I am owner of vector $j$ then
         $[i, j]$
      endif
   end
end
```

Figure 4: MGS algorithm using a data distribution scheme.

next paragraph. As for the assignment rule, computations that modify the data are assigned to the processor which owns them. This technique is illustrated by figure 4. Each vector of the initial matrix is assigned to a processor; the control flow of the parallel algorithm is similar to the sequential one. However, it differs when data are modified: normalizations $[i, i]$ (resp. corrections $[i, j]$) are done by the processor that owns vector i (resp. vector j). Since the result of a normalization is needed by all the remaining processors, it is broadcast to all of them. This parallel algorithm could be optimized by overlapping communications with computations: for each step of the i loop, as soon as the vector (that follows the one which has been normalized) has been corrected, it can be normalized and broadcast without waiting the end of all the corrections.

Since computations are assigned to processors that own the required data, finding a suitable data mapping function influences the load balance among processors. If the number of vectors is greater than the number of processors, several well known strategies can be used to map vectors onto processors: *block* or *wrap* mapping. In the first case, a set of contiguous vectors is associated with a processor whereas in the second approach, vectors are distributed to processors like playing cards are dealt to players (vector i is assigned to processor i *mod* p). Tradeoffs between these two mapping strategies are discussed in [7], Geist et al. have shown that the wrap mapping seems as straightforward and effective as any. In the next section, we will present experimental results that use this mapping technique.

The algorithm was programmed on an iPSC/2 hypercube with 32 nodes, where each node consists of one 80386 microprocessor augmented by an 80387 floating point co-processor and 4 Mbytes of local memory. The code was written in C to run in single precision. It is given in figure 5. Results are summarized in table 1 and clearly demonstrate the efficiency of a data oriented distribution scheme. The sequential time for matrix size 1024 x 1024 and 2048 x 2048 have been extrapolated from the time required for computing a matrix which size is 512 x 512.

For a larger number of processors, results can be predicted thanks to a theoretical model described in section 6.1.1.

```
#define NC 1024
#define MC 1024
#define IRMAX 1024

float A[IRMAX][MC], v[MC];
int i, j, k, me, nproc, iloc, jloc;
float comp, xnorm;

main()
{
  /* Initializing the matrix */
  ....
  /* Computing the modified-gram-schidt */
  me = mynode();
  nproc = numnodes();
  for (i=0; i<NC; i++) {
     /* local index of i */
     iloc = i / nproc;
     if ((i % nproc) == me) {
     /* vector i is assigned to me */
        xnorm = 1.0 / snrm2(MC, A[iloc], 1);
        sscal(MC, &xnorm, A[iloc], 1);
        csend(VECTORTYPE, A[iloc], MC*sizeof(float), ALLNODES, NODEPID);
        v = A[iloc];
     }
     else
        crecv(VECTORTYPE, v, MC*sizeof(float));

     for (jloc=0; jloc¡(NC/nproc); jloc++) {
        /* k is the global index of local index jloc */
        k = jloc * nproc + me;
        if (k > i) {
           comp = - sdot(MC, v, 1, A[jloc], 1);
           saxpy(MC, &comp, v, 1, A[jloc], 1);
        }
     }
  }
}
```

Figure 5: Parallel MGS algorithm using data oriented distribution.

Problem	*Serial*	*8 Processors*		*16 Processors*		*32 Processors*	
Size	*Time (s)*	*Time (s)*	*Speedup*	*Time (s)*	*Speedup*	*Time (s)*	*Speedup*
512 x 512	1375.97	184.75	7.45	99.36	13.85	57.27	24.03
1024 x 1024	11002.29	1425.32	7.72	470.35	14.86	398.74	27.59
2048 x 2048	87974.29	11217.14	7.84	5725.24	15.37	2972.11	29.6

Table 1: Time and speedup for data oriented distribution

```
for $i = 1 : n$
  $[i, i]$
  forall $j = i + 1 : n$
    $[i, j]$
  end
end
```

Figure 6: Parallel MGS algorithm using control distribution.

5 Control oriented distribution

To obtain a parallel implementation of the MGS algorithm with a *control-oriented* distribution scheme, it is necessary to analyze its dependence graph in order to discover which loops can be run in parallel. In the sequential algorithm, $[i, j]$ cannot begin until $[i, i]$ has been completed. There is no restriction on the order in which $[i, j]$ operations are performed. One approach consists of distributing the inner j-loop among the available processors. Figure 6 shows the parallel version of the MGS algorithm. The parallel algorithm consists of a repeated execution of a sequential phase (computing $[i, i]$) followed by a parallel phase (computing all $[i, j]$ where $j \geq i + 1$).

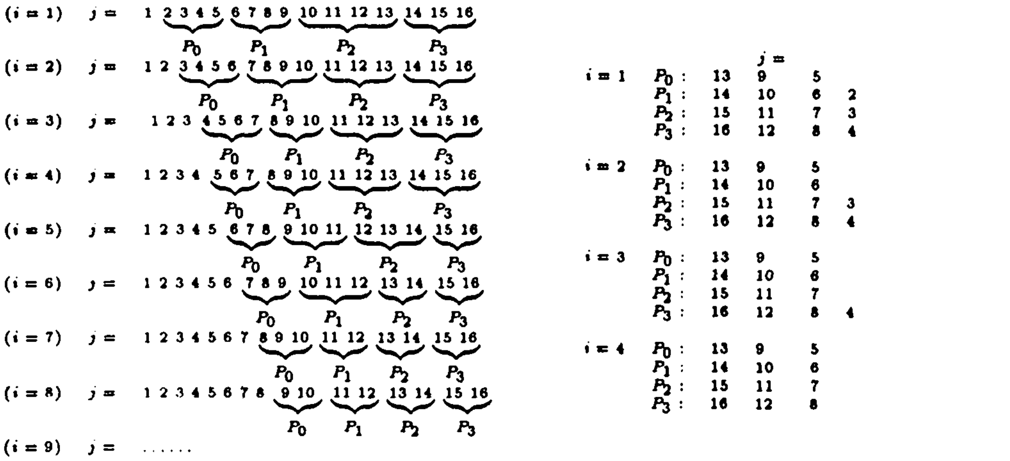
There are several approaches for distributing the j-loop among the processors. They must be done dynamically for each element of the i-loop by splitting the loop domain $[i + 1, n]$. Since the matrix is read from and written to the virtual shared memory, the user has to choose a distribution strategy that will minimize page faults by using *temporal locality.* Two successive parallel phases must use the same data set as much as possible.

A first approach consists to assign to each processor k contiguous element of the loop domain $[i + 1, n]$ as shown in figure 7-a. However, one can ascertain that between two successive steps of the i-loop, each processor does not use the same set of data. For instance, at step $i = 1$, processor P_0 has to read and write in vector 5 whereas it has been used by processor P_1 at step $i = 0$. We can choose a second approach for improving temporal locality. It consists of interleaving (similar to the wrap mapping technique) each element of the loop domain $[i + 1, n]$ as shown in figure 7-b.

If we do not care about how vectors of the initial matrix are stored in the virtual shared memory, we can assert that the second approach is the best one. However, the granularity of access of our shared virtual memory is the size of a page. Depending on the size of a vector, a page can be used for storing several parts of vectors. As shown in figure 8, loop distribution by interleaving may increase the number of page conflicts between processors. In this example, each symbol x represents a potential page conflict when the two vectors are processed by two different processors. The impact of such loop distribution is discussed in the next section.

5.1 Results

The algorithm above was implemented on an hypercube iPSC/2 running our KOAN shared virtual memory. In this implementation, the page size is set to 4096 bytes by the MMU. It cannot



(a) Distribution by block (b) Distribution by interleaving

Figure 7: Two distribution approaches.

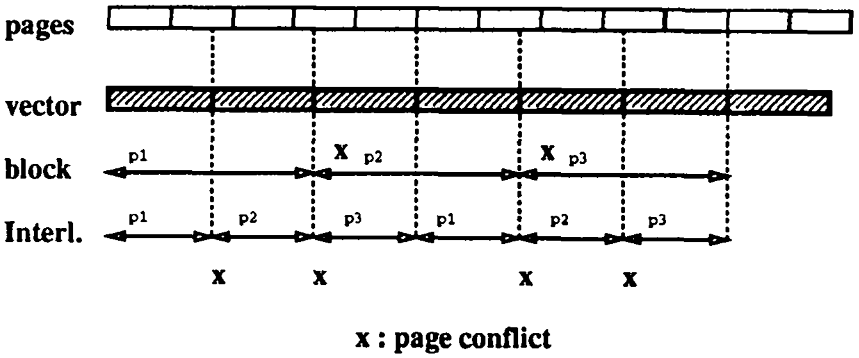


Figure 8: Page conflicts vs loop distribution

Problem	*Serial*	*8 Processors*		*16 Processors*		*32 Processors*	
Size	*Time (s)*	*Time (s)*	*Speedup*	*Time (s)*	*Speedup*	*Time (s)*	*Speedup*
512 x 512	1394.35	220.57	6.32	149.17	9.35	136.57	10.21
1024 x 1024	11149.34	1463.14	7.62	787.98	14.15	499.91	22.01
2048 x 2048	89172.94	11463.27	7.78	5917.52	15.07	3222.93	27.67

Table 2: Time and speedup for control oriented distribution

be changed since it is hardware dependent. The serial times are slightly different from those obtained with a data oriented distribution (table 1). It is due to the fact that KOAN has been implemented using a NX/2 version with kernel debug facilities which adds some overheads. Nevertheless, speedups supplied in this section can be compared with those given in table 1. The serial times 1024 x 1024 and 2048 x 2048 have been extrapolated from the time required for computing a matrix whose size is 512 x 512. The performances of this algorithm depend on various parameters: the way in which the j-loop is distributed or the way in which the vectors are stored in the SVM (page boundaries). Therefore, since several significant improvements could be made to increase the efficiency of this parallel algorithm, we report the impact of each improvement in the following sections.

5.1.1 Preliminary results

In this section, we present some experimental results of a naive implementation of the parallel algorithm: the j-loop is distributed by block and vector i is normalized on processor $i \; mod \; p$ in order to avoid page swapping. If all the normalizations are computed by the same processor, at the end of the computation, that processor will own all the physical pages required for storing all the vectors. For a large set of vectors, the size of a local memory will not be sufficient for storing them. Therefore, pages have to be moved to another processor that has enough memory. These results are summarized in table 2 for different matrix sizes.

The results in table 2 confirm that the control oriented distribution strategies provide appropriate speedups for large matrices. However, the efficiency for small matrix sizes (512 x 512)[2] is poor. It is due to the fact that several vectors share the same page. If these vectors are modified by different processors at the same time, the number of page faults and invalidation increase quickly. Consequently, since there is more communication between processors, the efficiency of the parallel algorithm decreases. Let us now show how to improve these results by modifying this naive implementation. First, section 5.1.2 will show the influence of loop distribution. Then, we will compare two strategies in section 5.1.3 for the purpose of decreasing the number of page conflicts.

5.1.2 Influence of loop distribution

To demonstrate the influence of the loop distribution scheme, we have studied performances on matrices where the number of vectors is set ($n = 1024$) and the vector size m is in the 1024 to 2048 range. Table 3 shows results according to the row length (S=Speedup, INV=Number of local and remote invalidations, WPF=Number of write page faults, RPF=Number of read page faults). The number of processors is set to 32. These results confirm the issue described in figure 8. Even if

[2] this has been confirmed experimentally for smaller matrix sizes

Vector	*Distribution by block*				*Distribution by interleaving*			
size	*S*	*INV*	*WPF*	*RPF*	*S*	*INV*	*WPF*	*RPF*
1024	22.01	16864	16864	48113	22.34	0	0	31248
1028	18.15	92566	53544	123915	9.78	848836	877657	868015
1032	19.46	71959	29894	103464	10.32	794145	815741	814213
1036	19.16	74093	34718	105700	10.33	798912	817330	817626
1088	19.89	67684	27207	100890	9.90	821516	859099	839133
1152	19.36	79042	43807	114163	10.79	749827	769238	757363
1280	21.38	55807	19101	94887	12.31	681890	685805	714122
1536	21.70	58140	39069	104964	13.90	731912	750986	756236
1792	21.81	80747	52883	135387	16.09	761414	786352	789282
1892	21.51	105140	67703	162826	16.95	771419	798743	798362
2048	23.94	33728	33728	96212	23.97	1984	1984	64480

Table 3: Influence of loop distribution.

loop distribution by interleaving allows one to handle the underlying locality of the algorithm, the speedup decreases by a factor up to two due to the increasing number of page conflicts. Except $m = 1024$ and $m = 2048$ where vectors are aligned on page boundaries, the number of invalidations increases by a factor from 7.33 up to 12.58 when interleaving is applied instead of blocking.

These results illustrate that block distribution seems to be the best technique since it is the one that achieves the lowest number of page conflicts. However, this latter does not exploit the underlying locality that can be found in the MGS algorithm. Therefore, we will compare two strategies in the next section. The first one uses both a loop distribution by interleaving and a mapping of vectors onto page boundaries. The second one handles a weak cache coherence protocol with loop distribution by block.

5.1.3 Page alignment versus weak cache coherence

The first solution consists of padding the matrix in order to put the beginning of each vector on a beginning of a page. This technique avoids page conflicts since there is only one vectors in a page. However, a part of the virtual shared memory is lost. This mapping technique allow us to handle efficiently a loop distribution by interleaving since there is no more potential page conflict between processors. We would expect the distribution by interleaving to provide much better speedup than the distribution by block. Unfortunately, Table 4 shows that, even if the number of invalidations decreases by a factor up to 18 and the number of page faults is divided by a factor up to 2, the gains are very small when the j-loop is distributed via an interleaving scheme. This diminution of gain is due to the fact that after the first execution of the i-loop, all the remaining write operations (normalizations and corrections) will be local. For $m = 1024$, there are no invalidation nor write page faults since the initial mapping coincides with the use of the pages. For $m = 1028, ..., 2048$, the behavior of the algorithm is the same as for $m = 2048$ since each vector is padded. The number of invalidation and write page faults are due to the first normalization $[1,1]$ and all the corrections $[1,2]....[1,n]$. This is not true for the block distribution scheme, and that explains why the number of invalidation and write page faults is greater.

Vector	*Distribution by block*				*Distribution by interleaving*			
size	*S*	*INV*	*WPF*	*RPF*	*S*	*INV*	*WPF*	*RPF*
1024	22.01	16864	16864	48113	22.34	0	0	31248
1028	20.47	33728	33728	96212	20.83	1984	1984	64480
1032	20.49	33728	33728	96212	20.85	1984	1984	64480
1036	20.52	33728	33728	96212	20.87	1984	1984	64480
1088	20.81	33728	33728	96212	21.14	1984	1984	64480
1152	21.16	33728	33728	96212	21.44	1984	1984	64480
1280	21.75	33728	33728	96212	21.97	1984	1984	64480
1536	22.69	33728	33728	96212	22.82	1984	1984	64480
1792	23.39	33728	33728	96212	23.46	1984	1984	64480
1892	23.63	33728	33728	96212	23.67	1984	1984	64480
2048	23.94	33728	33728	96212	23.97	1984	1984	64480

Table 4: Influence of loop distribution when vectors are aligned on page boundaries.

Another solution consists of using a weak coherence protocol as described in [8]. Since several processors have to write into different locations of the same page, we can let them modify concurrently their own copy of a page. Giloi et al. have suggested using two new constructs: *begin_weak* and *end_weak* which delimit a program section in which a weak coherency protocol is used instead of a strong coherency protocol. When a *end_weak* is executed, all the copies of a page which have been modified in the weak block are merged into one page that reflects all the changes. The user has to modify his parallel program in order to add these new constructs in the right places. An example of the parallel MGS algorithm with such new constructs is shown in figure 9.

Our implementation has some restrictions. Within a block delimited by *begin_weak* and *end_weak*, each processor has to announce the pages that it will modify during the execution of all statements that belong to the block. The operating system has to know which are the pages that will need to be merged when *end_weak* will be executed. This number of pages has to be as small as possible in order to reduce the size of the data structure managed by the operating system. Therefore, we have chosen to distribute the j-loop by block since this strategy assigns a contiguous set of the j-loop to each processor and consequently each processor modify a small amount of contiguous pages. Table 5 presents the results obtained with this strategy. They can be compared with those described in table 3 with the same loop distribution scheme, the speedup gain is in the 2.26% to 21.38% range. The effect of this strategy is to smooth the speedup curve when the vector size increase from 1024 to 2048 as show in figure 10. This increase in performance is obtained thanks to the reduction of page faults. However the number of invalidations shown in table 5 increases compared to those obtained with strong coherency. Most of these invalidations are local and do not add much overhead. Indeed, when statement *end_weak* is executed, the content of the copies of a page that have been modified concurrently are merged into one copy. Each copy that belong to a processor has to be invalidated in order to guarantee that there is only one copy with write access in the system. This invalidation is made locally by each processor. This kind of invalidation appears in the results shown in table 5.

```
#define NC 1024
#define MC 1024
#define SIZE NC*MC*sizeof(float);

typedef float vec[MC];
vec *v;
int uid_v;
int i, j, me , nproc, idxp;
float comp, xnorm;

main()
{
  /* Creating the shared region */
  uid_v = create_region(SIZE);
  /* Mapping of the shared region */
  v = (vec *) map_region(uid_v, MODULO, WEAK);
  /* Initializing the matrix */
  ....
  /* Computing the modified-gram-schidt */
  me = mynode();
  nproc = numnodes();
  for (i=0; i<NC; i++) {
      idxp = i % nproc;
      /* Processor number idxp normalize vector v[i] */
      if (me == idxp) {
        xnorm = 1.0 / snrm2(MC, v[i], 1);
        sscal(MC, &xnorm, v[i], 1);
      }
      gsync();
      /* Computing in parallel */
      compute_interval(nproc, me, i+1, NC, &low, &end);
      begin_weak(uid_v, &v[low][0], &v[end][MC]);
      for (j=low; j≤end; j++) {
        comp = - sdot(MC, v[i], 1, v[j], 1);
        saxpy(MC, &comp, v[i], 1, v[j], 1);
      }
      /* Global synchronisation */
      end_weak(uid_v);
  }
  /* freeing the shared region */
  free_region(uid_v);
}
```

Figure 9: Parallel MGS algorithm using weak coherency

Vector	*Distribution by block*			
size	*S*	*INV*	*WPF*	*RPF*
1024	23.84	47105	16864	31233
1028	22.03	108758	19730	77400
1032	22.04	108734	19728	77344
1036	22.04	109195	19912	77653
1088	22.36	105302	20355	75256
1152	22.56	102044	21103	73176
1280	22.95	97954	22873	70170
1536	23.66	94089	26441	66277
1792	23.58	121531	31239	85797
1892	23.65	148449	33699	103746
2048	24.48	94210	33728	62466

Table 5: Results when using a weak cache coherence protocol.

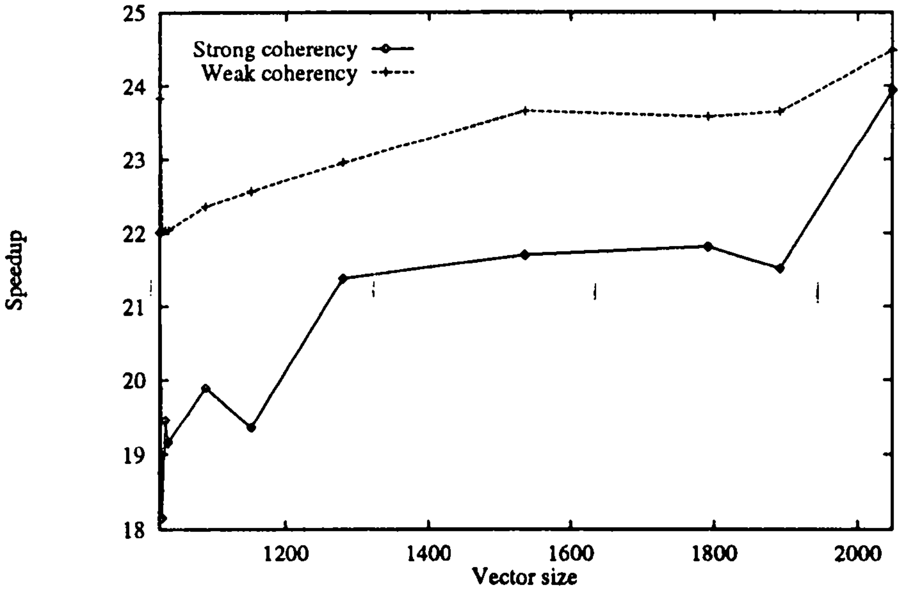


Figure 10: Strong coherency vs weak coherency.

6 Performance analysis

6.1 Modeling communications and computations

6.1.1 Data oriented distribution

This section deals with the study of the behavior of the parallel algorithm that uses a data oriented distribution in order to determine its performance on a large number of processors. For a given number of vectors n and a number of processors p [3], each processor x does the following computations and communications:

$$\begin{array}{ll} Number\ of\ normalizations: & n/p \\ Number\ of\ broadcasts: & n/p \\ Number\ of\ receptions: & n - n/p \\ Number\ of\ corrections: & (n^2 + 2nx - np)/2p \end{array}$$

The total time for the parallel algorithm can be broken down into three parts:

$$T_{par} = T_{normalization} + T_{correction} + T_{overhead}$$

Let us assume that normalizations and corrections are equally distributed so that each processor has the same amount of computations. This is true except at the end of the algorithm when there are fewer corrections than processors. It follows that

$$T_{normalization} + T_{correction} \approx T_{seq}/p$$

A processor has to wait for the vector which has been normalized by another processor. Denoting by τ_a the time to perform one floating point operation (single precision), by β_c the start-up time for one communication and by τ_c the time to transfer one byte, it can be shown that the imputed overhead is as follows:

$$T_{overhead} = (n - n/p)\ (3m + 1)\ \tau_a + n\ (\beta_c + 4m\tau_c)\ log_2(p)$$

For the iPSC/2 hypercube, the values for τ_a, τ_c and β_c are:

$$\begin{aligned} \tau_a &= 8.36\ \mu s \\ \tau_c &= 0.359\ \mu s \\ \beta_c &= 704\ \mu s \end{aligned}$$

Therefore, the time for p processors is equal to:

$$T_{par} = T_{seq}/p + (n - n/p)\ (3m + 1)\ \tau_a + n\ (\beta_c + 4m\tau_c)\ log_2(p)$$

[3] Let us assume that p divides n

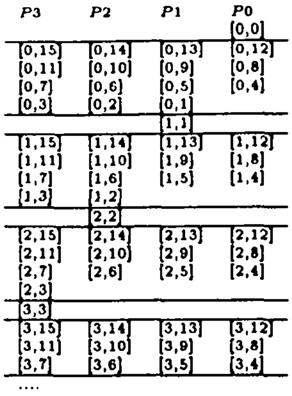


Figure 11: Time course for control oriented distribution (p=4,n=16)

6.1.2 Control oriented distribution

Modeling the behavior of the algorithm running with a shared virtual memory is arduous since we have to model the way in which both the parallel algorithm and the memory accesses behave. In our case, the cost of these memory accesses depends on various parameters such as the size of the matrix, the non deterministic nature of concurrent processes, etc... Nevertheless, a upper bound of the efficiency of the algorithm could be found by choosing the best possible case. This can be achieved by choosing the vector size such that it fits exactly into a page. Therefore, in this analysis, we make the following assumptions:

- p divides n evenly.
- The size of the vector is equal to the size of a page ($m = 1024$), this assumption prevents two different processors from write into the same page.
- The j-loop is distributed according to an interleaving scheme that takes advantage of the underlying locality (cf figure 7-b).
- Vector i is normalized by processor numbered i *mod* p so that it does not require any write page fault since processor i *mod* p has write access to the page that contains vector i.
- We neglect the cost of page faults that occur during the processing of the first step of the i-loop. For all the remaining steps, there are no more page faults thanks to the interleaving distribution scheme of the j-loop.
- We assume that processor $p - 1$ is the last to receive the normalized vector when it makes a request via the SVM (worst case).

With these constraints, an example of a time course for our parallel algorithm is shown in figure 11.

As for data oriented distribution, we can make the following assumption:

$$T_{normalization} + T_{correction} \quad \approx \quad T_{seq}/p$$

Consequently, the total time for the parallel algorithm is T_{par} and is equal to:

$$T_{par} \quad = \quad T_{seq}/p + T_{overhead}$$

The overhead associated with this parallel algorithm originates in the inactivity of processors, the cost of barrier synchronization and last but not least the cost for accessing data throught the shared virtual memory. Denoting by f_{svm} the time to process a read page fault, it can be shown that the overhead is as follows:

$$\begin{aligned} T_{overhead} \quad = \quad & (n - n/p)(3m + 1)\tau_a + \\ & n\ \beta_c\ log_2(p) + \\ & (n - 1)\ (4m/4096)\ f_{svm}(p) \end{aligned}$$

Since the model works only for $m = 1024$, the parallel time can be simplified:

$$T_{par} \quad = \quad T_{seq}/p + (n - n/p)(3073)\tau_a + n\ \beta_c\ log_2(p) + (n - 1)\ (4m/4096)\ f_{svm}(p)$$

Let us describe the meaning of f_{svm}. After each normalization, each processor reads the resulting vector that is located either on a remote processor $(n - n/p)$ or in its local memory $(n/p - 1)$. When the normalized vector is local to processor $p - 1$, each processor has the same number of corrections to compute, therefore, processor $p - 1$ has to wait at the barrier synchronization until the last processor to receive the normalized vector has computed all the corrections. This waiting is take into account in $T_{overhead}$. The cost for accessing a data via the SVM can be modelized by the function f_{svm} that takes as parameter the number of processors involved in the computation. Since the processes of page requests are serialized, f_{svm} is a linear function $f_{svm}(p) = (\beta_p + \alpha_p p)$ where β_p and α_p is fixed experimentally. For our SVM implementation, theses values are:

$$\begin{aligned} \beta_p \quad &= \quad -4312.6 \\ \alpha_p \quad &= \quad 3206.323 \end{aligned}$$

6.2 Comparison

Thanks to these two performance models, we can compute the theoretical speedups for both data oriented and control oriented distribution. Concerning the data oriented distribution, the overhead is in $O(log_2(p))$ while it is in $O(p)$ for control oriented distribution. Consequently, speedups for the two distribution schemes can be approximated as follows:

$$\begin{aligned} S_{data} \quad &= \quad \frac{p}{1+\alpha\ p\ log_2(p)} \\ S_{ctrl} \quad &= \quad \frac{p}{1+\beta\ p^2} \end{aligned}$$

The speedups predicted by the models are shown in figure 12. As the number of processors increases, keeping the number of vectors constant, the speedup obtained with control oriented distribution decreases when the number of processors is greater than 45. The main reason is the serialization of the processing of page requests in the processor that has normalized the vector. The

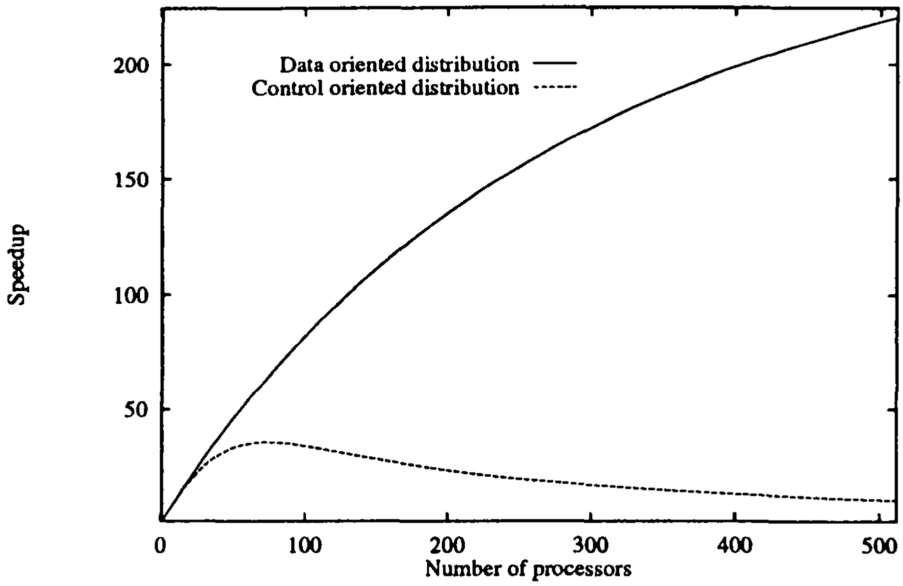


Figure 12: Theoretical speedup

behavior of the two parallel algorithms seems to be the same except that in the first algorithm, the normalized vector is broadcast to all processors in $O(log(p))$ steps (thanks to the hypercube topology) whereas in the second algorithm, each processor ask the normalized vector via the SVM. These resulting page requests are processed sequentially by the processor that owns the normalized vector in $O(p)$ steps. This does not allow the concept of shared virtual memory to be scalable when a parallel algorithm acts as a producer/consumer. More operating system support is needed to take into account the broadcasting facility of the underlying topology. Such support has been proposed in [4] in order to avoid having the processors wait for pages. It consists of moving pages to the processors at which they are going to be used before they are required. The same idea can be used to fully exploit the topology of distributed memory parallel architectures. An implementation in our shared virtual memory is provided in the next section, as well as an analysis of its influence on the performance the Modified Gram-Schmidt algorithm.

7 Optimizing control oriented distribution

As pointed out in the previous section, the major drawback of shared virtual memory on DMPC is its inability to run efficiently parallel algorithms that contain a producer/consumer scheme. Since an operating system does not have any knowledge of the behavior of the user's algorithm, the user has to specify explicitly the producer phases in his algorithm in order for the OS to handle this scheme efficiently. Our idea is to broadcast all pages that have been modified by the processor in charge of running the producer phase to all other processors that will run the consumer phase in parallel. Since the producer has to keep track of all pages that have been modified, two new operating system calls have to be added in order to specify both the beginning and the ending of

Vector size	*Speedup without broadcast*	*Distribution by interleaving with broadcast*				*Speedup gain*
		S	*INV*	*WPF*	*RPF*	
1024	22.34	27.47	0	0	0	23.02%
1028	20.83	26.55	1984	1984	1984	27.43%
1032	20.85	26.53	1984	1984	1984	27.24%
1036	20.87	26.53	1984	1984	1984	27.14%
1088	21.14	26.60	1984	1984	1984	25.85%
1152	21.44	26.68	1984	1984	1984	24.46%
1280	21.97	26.81	1984	1984	1984	22.01%
1536	22.82	27.00	1984	1984	1984	18.31%
1792	23.46	27.14	1984	1984	1984	15.68%
1892	23.67	27.19	1984	1984	1984	14.87%
2048	23.97	27.26	1984	1984	1984	13.74%

Table 6: Influence on performances when pages are broadcast

the producer phase. Figure 13 shows a modified version of the parallel MGS algorithm running with KOAN that includes two new system calls: **begin_broadcast()** and **end_broadcast()**. The first call takes as parameters the shared region identifier and the processor number that run the producer phase. This call is executed by every processor since we use a SPMD programming model, however its behavior is slightly different. If the value of the second parameter is equal to the processor number that runs a **begin_broadcast()** statement, the processor keeps track of every page that will be modified in the producer phase. Otherwise, it does nothing and executes the next statement that follows the **begin_broadcast()**. When a processor runs a **end_broadcast()**, if it is the one in charge of running the producer phase, it broadcasts all the pages that have been modified, otherwise it waits to receive those pages.

The performance figures for the parallel MGS algorithm that uses the broadcast capability of our shared virtual memory are given in the table 6. In this experiment, a strong cache coherence protocol is used and vectors are aligned on page boundaries. The j-loop is distributed by interleaving with a view to exploiting the underlying locality. With these choices, this parallel algorithm is very similar to the one that uses a message-based programming model described in section 4.

Let us comment briefly on these results. When the vector size is equal to 1024, there is no invalidation and page fault, since data and control are distributed so that each processor uses pages that are in its local memory. For all other vector sizes, the algorithm acts as if the vector size was set to 2048 since each vector is padded. The numbers of invalidations and both read and write page faults are equal. They are due to the running of the j-loop when $i = 1$. When a processor makes a correction to a vector that is not in its local memory, it sends first one or several page requests for a read access (sdot BLAS function call) and then sends the page requests for a write access (saxpy BLAS function call). These latter write requests generate the same number of invalidations. After that, all the computations are made locally since the next execution of the j-loop will use the same pages and there is no page conflict between processors since vectors are aligned on page boundaries.

```
#define NC 1024
#define MC 1024
#define SIZE NC*MC*sizeof(float);

typedef float vec[MC];
vec *v;
int uid_v;
int i, j, me , nproc, idxp;
float comp, xnorm;

main()
{
  /* Creating the shared region */
  uid_v = create_region(SIZE);
  /* Mapping the shared region */
  v = (vec *) map_region(uid_v, MODULO, RW);
  /* Initializing the matrix */
  ....
  /* Computing the modified-gram-schidt */
  me = mynode();
  nproc = numnodes();
  for (i=0; i≺NC; i++) {
     idxp = i % nproc;
     /* Processor number idxp normalize vector v[i] */
     begin_broadcast(uid_v, idxp);
     if (me == idxp) {
       xnorm = 1.0 / snrm2(MC, v[i], 1);
       sscal(MC, &xnorm, v[i], 1);
     }
     end_broadcast(uid_v);

     /* Computing in parallel */
     for (j = NC - (nproc - me); j ≥ i+1; j -= nproc) {
       comp = - sdot(MC, v[i], 1, v[j], 1);
       saxpy(MC, &comp, v[i], 1, v[j], 1);
     }
     /* Global synchronisation */
     gsync();
  }
  /* freeing the shared region */
  free_region(uid_v);
}
```

Figure 13: Parallel MGS algorithm with broadcasting facility

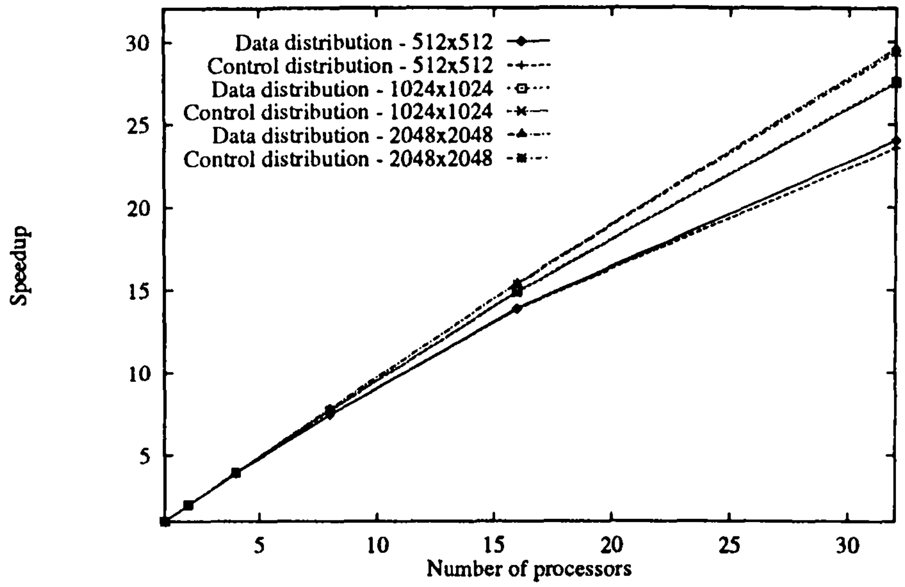


Figure 14: Data oriented versus control oriented distribution.

This kind of optimization can be applied when the j-loop is distributed by block and the shared virtual memory runs a cache coherence protocol such as the one described in section 5.1.3. However, our experiments show worse speedup in the 5% to 9% range. Therefore, the best results have been obtained when vectors are aligned on page boundaries, the j-loop is distributed by interleaving and pages which have been modified during the normalization are broadcast.

8 Conclusion

Based on the foregoing results, one can draw several conclusions. The results show in figure 14 demonstrate that a shared virtual memory with judicious optimizations can provide the same performance as that obtained with message-passing. Unfortunately, these optimizations are still difficult to find. In this example, programming with shared variables seems to be more complex than programming with messages. This is mainly due to the simpleness of the algorithm we have chosen; the parallel algorithm with messages is very simple. Forthcoming improvements in compilers for parallel architectures should simplify getting the best performance out of shared virtual memory.

A clear benefit of an SVM programming model is that it allows the user to easily migrate his application from sequential architectures to parallel ones. Regardless of the size of an application, it can be run on a single processor of a multi-processor architecture since SVM can be used as a simple virtual memory. Step by step, it can be progressively parallelized and optimized to fully exploit the performance of a distributed memory parallel architecture. As SVM techniques mature, this benefit of portability will be available at an ever lower cost in lost efficiency.

9 Acknowledgment

The authors wish to thank J. Erhel and R. Mac Connell for their careful reading and their helpful suggestions and comments for improving this paper.

References

[1] F.E. Allen, P. Charles, R. Cytron, and J. Ferrante. An overview of the ptran analysis system for multiprocessing. *Distributed Computing*, 5:617–40, 1988.

[2] D. Badouel, K. Bouatouch, and T. Priol. Ray tracing on distributed memory parallel computers: strategies for distributing computation and data. In S. Whitman, editor, *Parallel Algorithms and architectures for 3D Image Generation*, pages 185–198, ACM Siggraph'90 Course 28, August 1990.

[3] D. Badouel and T. Priol. An Efficient Parallel Ray Tracing Scheme for Highly Parallel Architectures. In *Eurographics Hardware Workshop*, Lausanne, Switzerland, September 1990.

[4] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Munin: distributed shared memory based on type–specific memory coherence. In *Principle and Practice of Parallel Programming*, March 1990.

[5] John K. Bennett, John B. Carter, and Willy Zwaenepoel. *Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence*. Technical Report Rice COMP TR89-98, Rice University, November 1989.

[6] G. Delp and D. Farber. *MemNet: An Experiment on High-Speed Memory Mapped Network Interface*. Technical Report 85-11-IR, University of Delaware, 1986.

[7] G.A. Geist and M.T. Heath. Matrix factorization on a hypercube multiprocessor. In M.T. Heath, editor, *Hypercube Multiprocessor 1986*, pages 161–180, Oak Ridge National Laboratory, Siam, 1986.

[8] W.K. Giloi, C. Hastedt, F. Schoen, and W. Schroeder-Preikschat. A distributed implementation of shared virtual memory with strong and weak coherence. In Arndt Bode, editor, *Distributed Memory Computing*, pages 23–31, LNCS 487, Springer-Verlag, April 1991.

[9] G.H. Golub and C.F. Van Loan. *Matrix Computation*. The Johns Hopkins University Press, 2nd edition edition, 1990.

[10] K. Ikudome, G. C. Fox, A. Kolawa, and J. W. Flower. An automatic and symbolic parallelization system for distributed memory parallel computers. In *5' Distributed Memory Computing Conference*, April 1990.

[11] Z. Lahjomri and T. Priol. *KOAN: a Shared Virtual Memory for the iPSC/2 hypercube*. Technical Report 1504, INRIA, September 1991.

[12] Kai Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors*. PhD thesis, Yale University, September 1986.

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

[14] J. Van Rosendale P. Mehrotra. *Programming Distributed Memory Architectures Using Kali.* Technical Report , ICASE, 1990. Draft.

[15] E. M. Paalvast and A. J. Van Gemund. A method for parallel program generation with an application to the *Booster* language. In *Int. Conf. on Supercomputing*, pages 457–469, June 1990.

[16] Jean-Louis Pazat. Code generation for data parallel programs on dmpcs. In *European Distributed Memory Computers Conference*, 1991.

[17] Umakishore Ramachandran and M. Yousef A. Khalidi. An implementation of distributed shared memory. *Distributed and Multiprocessor Systems Workshop*, 21–38, 1989.

[18] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *Conference on Programming Language Design and Implementation*, pages 69–80, ACM, June 21–23 1989.

[19] M. Rosing, R. B. Schnabel, and R. P. Weaver. *The DINO Parallel Programming Language.* Technical Report CU-CS-457-90, University of Colorado at Boulder, 1990.

[20] V. Abrossimov and M. Rozier. Generic virtual memory management for operating system kernels. In *12th ACM Symposium on Operating System Principle*, December 1989.

[21] Brigitte Vital. *Etude de quelques méthodes de résolution de problèmes linéaires de grande taille sur multiprocesseur.* PhD thesis, Institut de Formation Supérieure en Informatique et Communication, Novembre 1990.

[22] Brigitte Vital. *Mise en œuvre d'algorithmes numériques sur un hypercube.* Technical Report 450, IRISA, Janvier 1989.

[23] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Chew, W. Boloski, D. Black, and R. Baron. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. In *Eleventh ACM Symposium on Operating Systems Principles*, pages 63–76, 1987.

[24] Hans P. Zima, Heinz-J. Bast, and Michael Gerndt. SUPERB: a tool for semi-automatic MIMD /SIMD parallelization. *Parallel Computing*, (6):1–18, 1988.

LISTE DES DERNIERES PUBLICATIONS INTERNES IRISA

PI 630 EREBUS, A DEBUGGER FOR ASYNCHRONOUS DISTRIBUTED COMPUTING SYSTEM
Michel HURFIN, Noël PLOUZEAU, Michel RAYNAL
Janvier 1992, 14 pages.

PI 631 PROTOCOLES SIMPLES POUR L'IMPLEMENTATION REPARTIE DES SEMAPHORES
Michel RAYNAL
Janvier 1992, 14 pages.

PI 632 L-STABLE PARALLEL ONE-BLOCK METHODS FOR ORDINARY DIFFERENTIAL EQUATIONS
Philippe CHARTIER, Bernard PHILIPPE
Janvier 1992, 28 pages.

PI 633 ON EFFICIENT CHARACTERIZING SOLUTIONS OF LINEAR DIOPHANTINE EQUATIONS AND ITS APPLICATION TO DATA DEPENDENCE ANALYSIS

Christine EISENBEIS, Olivier TEMAM, Harry WIJSHOFF
Janvier 1992, 22 pages.

PI 634 UN NOYAU DE SYSTEME REPARTI POUR LES APPLICATIONS GEREES PAR UN TEMPS VIRTUEL
Janvier 1992, 20 pages.

PI 635 SOME ENHANCEMENTS OF CHERNIKOVA'S ALGORITHM
Hervé LE VERGE
Février 1992, pages.

PI 636 ENSEIGNER LA TYPOGRAPHIE NUMERIQUE
Jacques ANDRE, Roger D. HERSCH
Février 1992, 26 pages.

PI 637 TRADE-OFFS BETWEEN SHARED VIRTUAL MEMORY AND MESSAGE-PASSING ON AN IPSC/2 HYPERCUBE
Thierry PRIOL, Zakaria LAHJOMRI
Février 1992, 26 pages.

ISSN 0249-6399