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

N° 1504

Programme 1
Architectures parallèles, Bases de données,
Réseaux et Systèmes distribués

KOAN : A SHARED VIRTUAL MEMORY FOR THE iPSC/2 HYPERCUBE

Zakaria LAHJOMRI
Thierry PRIOL

Septembre 1991

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTÈMES ALÉATOIRES

Campus Universitaire de Beaulieu
35042-RENNES CÉDEX
FRANCE
Téléphone: 99.84.71.00
Télex: UNIRISA 950 473 F
Télécopie: 99 38 38 32

KOAN: a Shared Virtual Memory for the iPSC/2 hypercube

KOAN : une mémoire virtuelle partagée pour un hypercube iPSC/2

Zakaria Lahjomri
Thierry Priol

IRISA
Campus de Beaulieu
35042 Rennes Cedex
e-mail: priol@irisa.fr

Juillet 1991
Publication Interne n° 597 - 32 pages - Programme I

Abstract

In this paper, we describe the salient features of an implementation of a shared virtual memory, named KOAN, running on a iPSC/2 hypercube. We then discuss its performance on a non-numerical algorithm like ray-tracing as well as a numerical one: the Modified Gram-Schmidt algorithm.

Résumé

Nous décrivons dans ce papier un dispositif de mémoire virtuelle partagée, appelé KOAN, fonctionnant sur un hypercube iPSC/2. Nous présentons également les performances d'un tel dispositif à l'aide d'un algorithme non-numérique tel que le lancer de rayon ainsi qu'un algorithme numérique : l'algorithme de Gram-Schmidt modifié.

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE
(L. A. 227)
UNIVERSITÉ DE RENNES 1 I. N. S. A. DE RENNES

INSTITUT NATIONAL DE RECHERCHE
EN INFORMATIQUE ET EN AUTOMATIQUE
(LABORATOIRE DE RENNES)

1 Introduction

Programming distributed memory parallel computers (DMPC) using a Shared Virtual Memory (SVM) seems to be in fashion. However, there is few such system available for DMPCs. Most of the research in this area has been done for a network of workstations such as IVY [18], Clouds [9, 21], Munin [6, 5], Memnet [10], Mach [27] and Chorus [1, 24]. These implementations concern only high latency networks and do not allow the comparison of the efficiency of different strategies for parallelizing algorithms. Distributed memory parallel computers with a hypercube or 2D-mesh topology have been commonly used for designing and testing parallel algorithms. Several results are now available. Hence, it would be interesting to compare these results with those obtained by using a SVM on the same machine. This paper is a first step in this direction. We present an implementation of a SVM, named KOAN, on a iPSC/2 hypercube that allows us to compare the benefits of using either shared variables or message passing as a communication mechanism between parallel processes. It does not address a new SVM algorithm, such algorithms are well known. We have chosen one and we have modified and added new functionalities in order to implement it on a DMPC.

2 Shared virtual Memory

A SVM is 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 memory is divided into several pages as shown in figure 1. Pages are spread among local processor memories according to a mapping function which ensures that each processor has roughly the same number of pages. 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 remaining part acts as a large *software cache* for storing pages. This cache is managed according to a LRU (Least Recently Used) policy. A memory management unit (MMU) is needed to provide the user with a linear address space from the contents of the cache. An algorithm that implements a shared virtual memory has to solve three problems: *coherence*, *page ownership* and *swapping*. The following sections present some existing solutions for solving this problems and those we chose in implementing KOAN.

2.1 Cache coherence protocol

Since processors may have to read from or to write to the same page, several processors have a copy of a page in their cache. If one processor modifies its copy, other processors run the risk of reading an old copy. A cache coherence protocol is needed to ensure that the shared address space is kept coherent at all times. A memory is considered *coherent* if the value returned by a read from a location of the shared address space is the value of the latest store to that location [7]. Cache coherence protocol can be divided into two categories. The first one assumes that there is only one copy of a page with write access mode and all other copies are in read-only access mode. The processor that has the page in write access mode is called the *owner* of the page. When a processor needs to write to a page, that is not present in its cache or is present in read-only mode, it sends a message to the owner of the page in order to move it to the requesting processor. Then it invalidates all the copies in the system by sending a message to the relevant processors. This strategy is called the **invalidation** approach. The second category allows several processors to

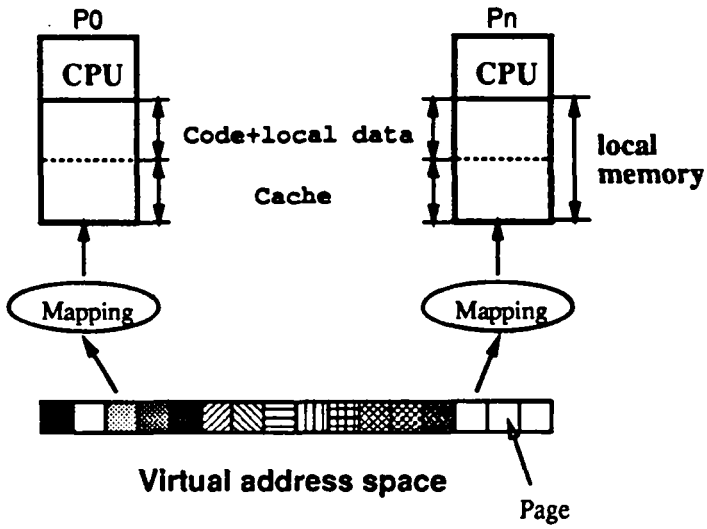


Figure 1: Mapping a Shared Virtual Memory on DMPC.

write to a page. However, the processor that does the write has to update all the copies of the page by sending a message to the relevant processors. This technique is called **distributed write** in [2] or **write-back** in [18]. A cache coherence protocol based on this approach is expensive and difficult to implement efficiently on "general- purpose" DMPCs. Every write to a shared page must generate a fault on the writing processor and a small message is sent to update all the copies. Sending small messages is very costly due to the latency of the interconnection network of DMPCs. Consequently, the invalidation technique seems to be the best approach for DMPCs. The faulting management mechanism of a MMU is sufficient to implement this approach efficiently. Moreover, the granularity of an access is the size of a page which is large enough to amortize the latency of the network.

2.2 Page ownership

When a processor needs to access a page, either in write or read access mode, which is not located in its cache, it must ask the owner to send it a copy of the page. This problem is related to the cache coherence protocol described previously. With the invalidation protocol, there is always one owner for a page and the ownership changes according to the page requests coming from other processors. Therefore, the problem is how to locate the current owner of a given page considering that the owner of a page changes. A solution is to update a database that keeps track of the movement of pages in the system. This database can be either **centralized** or **distributed** [18]. In the centralized approach, a processor (called the manager) is in charge of updating the database for every page. When a processor needs a page, it sends a request to the manager which forwards the request to the owner of the page. Consequently, the manager is aware of all the movement of pages in the system. However, it is a bottleneck since one processor receives requests from all other processors. The user's process running on the manager will be interrupted frequently and this approach will create potential contention into the network. Distributing the database on several processors is a means to avoid these drawbacks. There are two ways to distribute the database:

- **Fixed distributed manager**
  Each processor knows exactly the owner of a subset of pages. The subset is fixed by a mapping function that takes the page number as an argument and returns the processor number of the owner. The function $H(p) = p \ mod \ N$ is an example of such mapping function, where p is the page number and N is the number of processors in the system.

- **Dynamic distributed manager**
  Each processor has partial knowledge for every page of the virtual address space. Each processor knows a probable owner. This information is updated by the page fault handler or the servers. In the worst case, several processors are needed to determine the "true" owner of a page. K. Li [18] has proved that a page request reaches always the true owner eventually.

Formal analyses of these three approaches for solving the ownership problem can be found in [18]. They show that the best results are obtained with a dynamic distributed manager.

2.3 Page swapping among processors

The problem arises when a processor is the owner of all the pages located in its cache and there is no more room in the cache. If it requests a new page, it has to find space in its cache. It cannot throw away a page from its cache since it owns all the pages. Moreover it cannot save the pages on external high speed storage devices, like disks, since most of DMPCs do not offer such things. Consequently, it has to find a processor which has either a copy of the page or enough space in its cache. If another processor has a copy of the page to be stored, it requires only ownership migration. Otherwise, it requires both page saving and ownership migration. A solution to this problem is proposed in [18] for the dynamic distributed ownership approach. Section 3.3.2 describe a new one for the fixed distributed ownership approach.

3 Implementing KOAN on an iPSC/2 hypercube

The KOAN SVM is embedded in the operating system of an iPSC/2 hypercube. It allows the use of fast and low-level communication primitives as well as a Memory Management Unit (MMU). It differs from the work described in [19] in that it is an operating system based implementation. We present briefly the architecture of the iPSC/2 in section 3.1 and its operating system in section 3.2. Finally, section 3.3 describes the SVM algorithm we have chosen for KOAN.

3.1 Architecture of the iPSC/2

The iPSC/2 system consists of two main components: the cube and the system resource manager. The cube houses all the nodes which are connected by the hypercube network. It consists of several cabinets (up to 4). Each of them houses up to 32 computational nodes. Each node consists of one Intel 80386 microprocessor augmented by an 80387 floating point co-processor and 4 Mbytes of local memory. The MMU on the Intel 80386 implements a large virtual address space instead of physical one. The node is equipped with the Direct Connect Module (DCM) for high speed routing message between nodes. These DCMs allow programmers to view the network as a complete communication graph. Each processor can send a message directly to any other processor. This is

very useful for implementing our shared virtual memory because the communication graph is not known in advance.

Software development tools are available on the System Resource Manager (which acts as a host processor), which is connected to node 0 via a special link. The SRM performs compilation, program loading and I/O operations for the cube.

3.2 Operating system of the iPSC/2

The operating system of the iPSC/2 consists of two parts. The first part runs on the SRM and consists of several UNIX processes. It allows several users to run their programs simultaneously by splitting the cube into sub-cubes. Each is assigned to a user. Several commands have been added to allow the management of sub-cubes or parallel processes. As for the second part, it is a small kernel, called NX/2, which runs on each processor of the cube. It handles memory and process management as well as interprocess communication. We describe briefly each of these modules.

3.2.1 Memory management

The physical memory associated with each processor is managed by NX/2 in paging mode which simplifies the allocation and deallocation of memory for system processes and user application codes. The size of a page is 4096 bytes and cannot be modified. The page tables and system data structures are large enough to map all physical pages.

3.2.2 Process management

The NX operating system allows multiple **heavy-processes** running on each node. Each process runs in a separate address space. There is no shared memory between processes running on the same processor. A *round-robin* scheduling algorithm assigns a time slice of 50 milliseconds for a process before running the next one. However, if a process is waiting to receive a message, the scheduler picks the next process from the ready queue.

3.2.3 Interprocess communication

The interprocess communication mechanism offered by NX/2 is asynchronous. Communication can be either blocking, non-blocking or interrupt driven. Blocking communication means that a process executing a send waits until the send is complete. The message is in the network and there is no guarantee that it has been received. A blocking receive suspends a process until a message arrives. Non-blocking communication never suspends a process and allows the user process to run concurrently with the communications. However, it is more difficult to use correctly since the user must ensure that the communication buffers are not reused before a send or receive is completed. NX allows the user to specify when a procedure should start running when receiving a message. A message is identified by its type (an integer), its source and its destination node. Type is used to allow the receipt of messages of a particular type.

The message-passing protocols provide flow control and buffer management to ensure that no messages are lost due to buffer overflow. However, it does not solve deadlock if all the communication buffers are used. The user must take this problem into account when designing its algorithm. Two protocols are used in NX/2 depending of the message size. Messages with a size lower or equal

to 100 bytes are sent in a single step. Each node maintains a set of buffers which are assigned exclusively to the other nodes. Each processor knows the number of buffers available on the other processors. Therefore, processor can send the user's message directly to the destination processor. This protocol is called the *short protocol.*

Messages greater than 100 bytes are sent in three steps. In the first step, the source node sends a PROBE message to the destination node. The destination node determines where it can place the message data. When a buffer has been found (second step), the destination node sends a REQUEST message back to the source node, starting that it can send the entire message. The source node then sends the entire message from the send buffer (third step).

3.3 KOAN design choices

The KOAN SVM implements the fixed distributed manager algorithm as described in [18] with an invalidation protocol for keeping the shared memory coherent at all times. Li's algorithm is implemented using a set of handlers and servers. In each processor, two exception handlers are used to trap page faults. A read fault handler manages the page faults which occur when a processor needs to read from a page not located in its cache. If a processor needs to write to a page which is in read-only access mode or not located in its cache, the write fault handler is called. Seven servers are needed to process page requests or page invalidations. The handlers and servers share two data structures:

- A page table, arranged in a two level hierarchy, is used by the MMU for translating virtual addresses to physical ones. It also contains several fields, one of which indicates the access rights associated with a page.
- A table located in each manager. It has as many entries as pages associated with the manager. For each entry, a field contains the processor numbers that have read copies of the page. Another field store the current owner of the page.

The handlers and servers run concurrently and must be synchronized if several processes, running either on the same processor or on a remote processor, generate multiple page faults on the same page. An implementation requires both lightweight processes as well as semaphores in order to synchronize them. Unfortunately, the NX/2 operating system does not provide such mechanisms. Its kernel is driven by interrupts (system calls, exceptions, external interrupts). The synchronization protocol must be modified to compensate for the missing primitives.

3.3.1 Synchronization protocol

Let us outline our implementation. As shown in figure 2, the KOAN kernel consists of seven servers and two handlers. The servers are installed in the receive interrupt routine of NX/2 where they process all the requests coming from remote processors. The handlers are activated by the MMU when a page fault occurs. The servers and handlers are described in table 1.

In each processor, the KOAN kernel must serialize both the local requests (page faults) and remote requests (pages requests, invalidations) which occur on the same page. This is done by using an **arbiter**. All incoming messages are processed by the arbiter which can choose to forward the request to the relevant servers, to store it in a request queue (that will be processed later), or to reject the messages by sending it back. Moreover, if a page fault occurs on a page which is currently

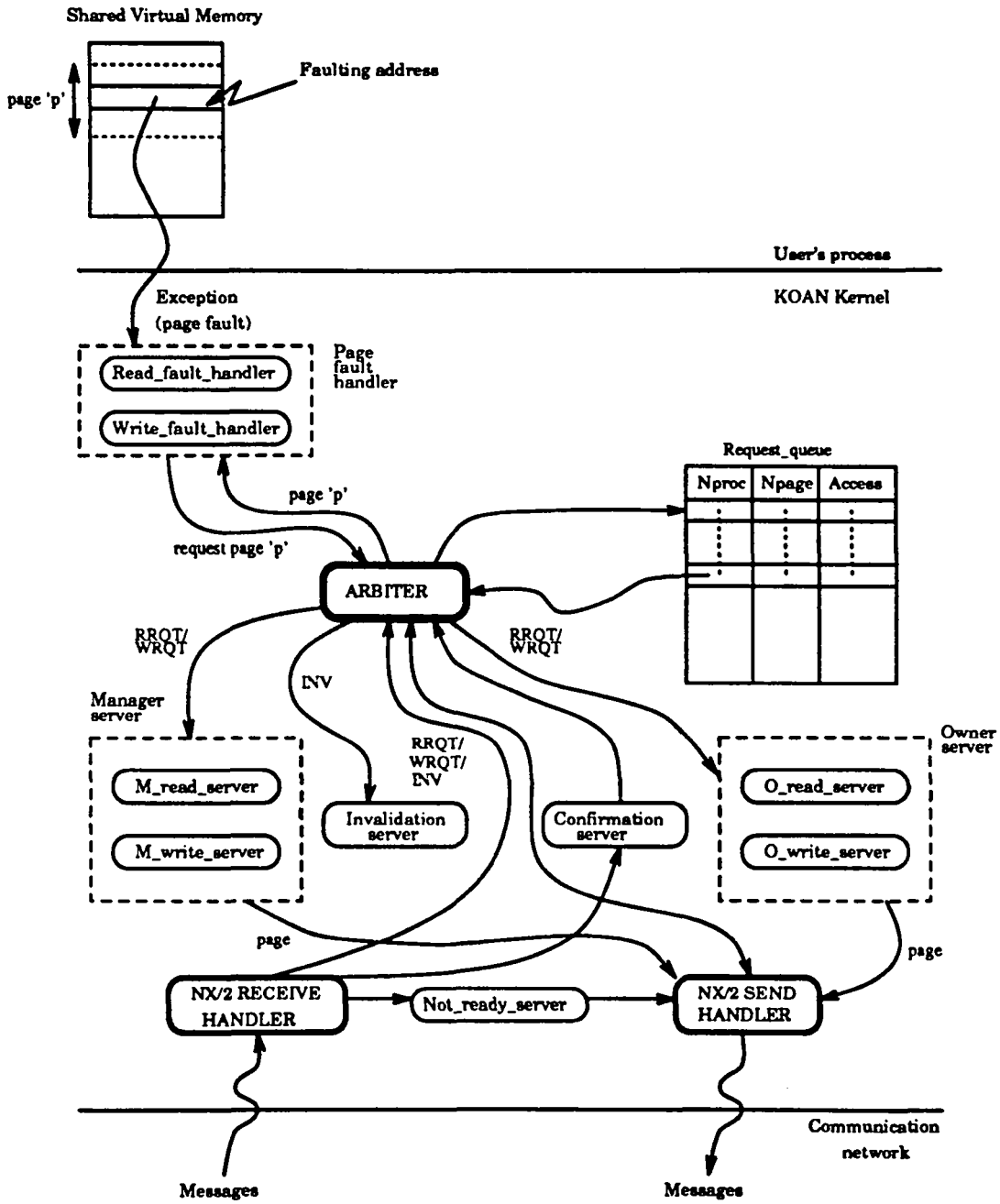


Figure 2: KOAN synchronization protocol.

Name	Description
Read_fault_handler	Called by the MMU when read page fault occures
Write_fault_handler	Called by the MMU when write page fault occures
M_read_server	Processes page requests by the manager (read access)
M_write_server	Processes page request by the manager (write access)
O_read_server	Processes page requests by the owner (read access)
O_write_server	Processes page requests by the owner (write access)
Invalidation_server	Change the access of a page to nil access
Confirmation_server	Processes confirmation messages
Not_ready_server	Resends a request rejected by a remote processor

Table 1: Servers and handlers

being processed by the arbiter, handlers must wait until the arbiter has finished processing the page request. Requests stored in the queue are processed when the arbiter has received a confirmation message indicating the completion of the previous page request.

Let us describe how requests are processed by each processor. In each processor, the arbiter manages a table called **busyout[]**. This table has an entry for each page assigned to the processor (manager). Initially, each entry of **busyout[]** is set to false. When the entry 'p' of **busyout[]** is true, it means that this page is currently being processed by one of the manager servers. This entry is set to false when the manager receives a *confirmation* message that tells it the end of the processing of the request.

A request for page 'p' coming from remote processors is first processed by the arbiter. Two cases may happen:

1. If page 'p' is not currently processed by one of the manager servers (**busyout[p]**=false), the arbiter sets **busyout[p]** to true and sends the request to one of the servers regarding to the required access mode.

2. If page 'p' is currently processed by one of the manager servers (**busyout[p]**=true), two cases may happen:

   (a) The manager is in page fault on the same page. If so, the arbiter rejects the request by sending it back. Indeed, request cannot be stored in the request queue because it is processed when a *confirmation* message is received. Since the manager does not send such message to itself, if the request is stored in the queue it will never be processed when the page fault will end. When a rejected request will be received by the processor which has sent it, the *not_ready_server* running in this processor will resend it.

   (b) Page 'p' is currently processed for another processor 'q' which is distinct from the manager. In that case, the arbiter stores the request in the request queue. When the arbiter will receive a *confirmation* message from processor 'q' for page 'p', it will get the oldest request on page 'p' from the request queue and will send it to the relevant servers.

Finally, if a page fault on page 'p' occurs in the manager, it is sent to the arbiter. If this latter is currently processing a remote request for page 'p' from a processor 'q', it stores the request (the page fault) in the request queue. This request will be processed later when the manager will receive a *confirmation* message from processor 'q'.

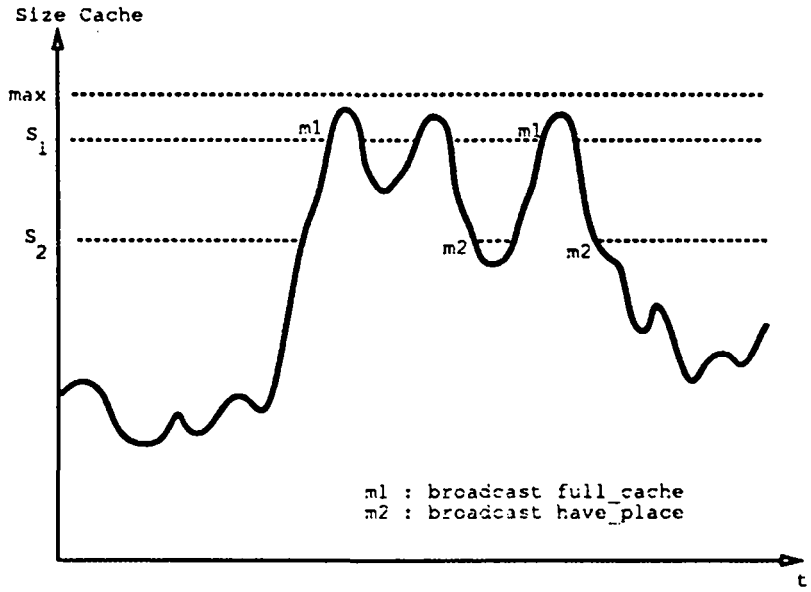


Figure 3: Cache management scheme.

3.3.2 A solution to the page swapping problem

In this section, we describe our solution to the page swapping problem. The physical address space of each processor contains different kinds of pages. Pages in *read-write* access mode, which are owned by the processor, belong to a set denoted P_{owner}, pages in *read-only* access mode belong to a set called P_{read}, and unused physical memory pages belong to a set named P_{free}. When a page fault occurs and there is no unused physical memory page available, the KOAN kernel invokes the following page replacement algorithm:

1. If P_{free} is not empty, delete a page p from P_{free} and return p.
2. Else, if P_{read} is not empty, delete a page p from P_{read} with a LRU policy (Least Recently Used) and return p,
3. Else, choose a page p with a LRU policy in the set P_{owner}.

A page chosen for replacement in the set P_{read} or P_{free} can be thrown away, but a page belonging to the set P_{owner} requires an ownership migration which consists of sending it to another processor. The following algorithm is used to find a processor that is able to receive a page in its local memory:

1. The faulting processor sends a request for ownership migration to the manager of the page.
2. When the manager receive this request, it checks if the requesting processor is still the owner of the page. If this is the case, the manager chooses one processor q with the procedure *Find_processor*, sends q to the requesting processor and waits for confirmation. Otherwise the manager sends a message named *resolved* to the requesting processor which means that the problem is solved because the requesting processor has already lost the ownership of the page.

Number of processors	Read page fault		Write page fault	
	best times (ms)	worst times (ms)	best times (ms)	worst times (ms)
2	2.995	3.170	3.110	3.133
4	3.059	3.560	3.144	3.742
8	3.104	3.615	3.274	4.701
16	3.255	3.783	3.355	6.654
32	3.412	3.955	3.447	10.110

Table 2: KOAN page fault times.

3. If the requesting processor receive the message *resolved*, it can replace the page immediately, otherwise it sends the page to processor q and then replaces the page.

4. The processor q receives the page, notes that it is the new owner of this page and sends a confirmation to the manager.

Let us describe how the manager finds a processor which has enough room in its local memory (procedure *Find_processor*). A local counter *count* is maintained on each processor which represents the number of pages owned by a processor at a given time. *count* is increased when the processor acquires the ownership of a page and is decreased when it relinquishes the page. When the counter *count* reaches the barrier S_1, the processor broadcasts a message named *full_cache* to all managers. When count reach the barrier S_2, the message *have_space* is broadcasted. Figure 3 illustrates this mechanism.

We can easily prove the correctness of this algorithm if the size of the shared memory address space, T, is bounded by:

$$T \leq N * (M_i - K + N - 1) \quad (1)$$

where M_i is the size of the memory space on processor i, N is the number of processors and K is equal to (S_1 - S_2).

4 Performance

We have performed measurements in order to determine the cost of various basic operations for both read and write page faults of the KOAN shared virtual memory. We used the *uclock* timer designed by P. Jacobson et al. [16] which has microsecond resolution. For each type of page fault (read or write), we have tested the best and worst possible situation on different numbers of processors. A page fault that occurs on the manager which is also the owner represents the best case for both read and write page faults. The worst case for read page fault occurs when the three entities are distinct processors: the faulting processor, the manager and the owner of the page are as far apart as possible. The worst case for write page fault is similar, but in addition all processors have a copy of the faulting page. The best and worst times for both read and write page faults are shown in table 2.

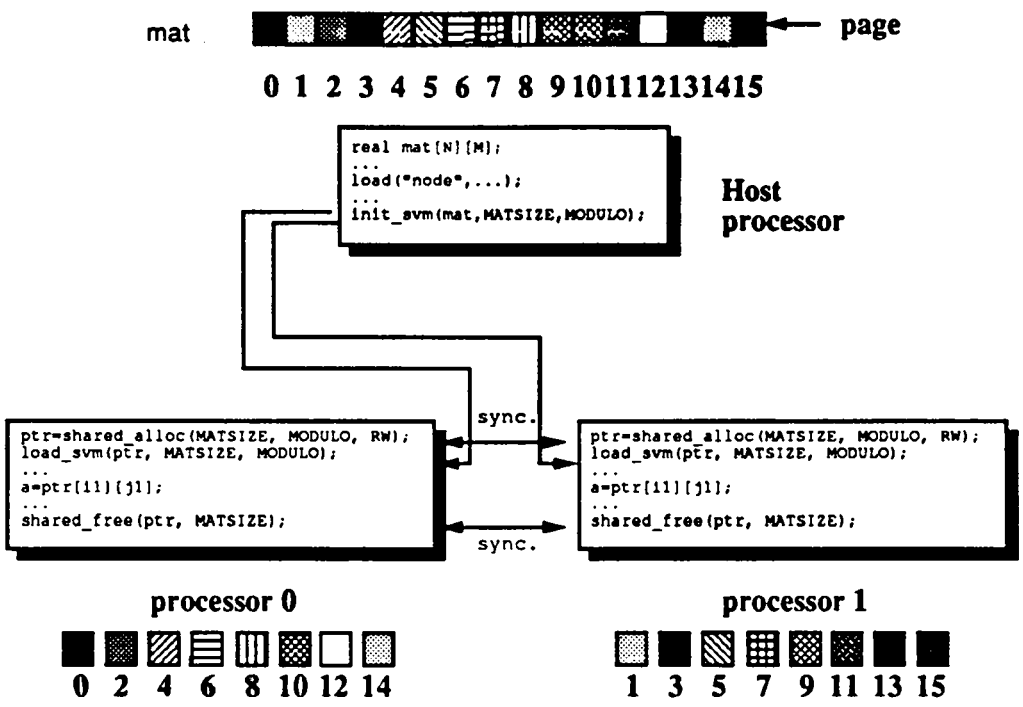


Figure 4: Using a shared region with KOAN.

5 Programming with KOAN

A library has been added to the C language in order to allow the user to manage the shared virtual memory. The C compiler has not been modified to support the KOAN shared virtual memory. The library is a set of low-level routines for allocating, freeing and initializing a shared region. Tools like barriers and critical section management allow the user to synchronize processes. KOAN provides both software event tracing and tools for performance analysis. The implementation of KOAN has some restrictions: it allows only one process per node and interrupt-driven communication is forbidden.

5.1 Allocating and freeing shared regions

To allocate a shared region, each processor calls the routine **shared_alloc()**. The behavior of this routine, when called by **all** the processors, is similar to a barrier synchronization. Each processor sets up its own system data and waits until all processors have finished. The **shared_alloc()** routine returns to each processor a pointer (a virtual address) to the shared region. Only one shared region is allowed. Therefore, the users is responsible for managing the shared region. He must compute the size of all data which are shared by the processors and maps the data in the shared region.

Synopsis:

ptr = **shared_alloc**(size, os, cc)

Parameter declarations:

int size : size of the shared region
int os : page ownership protocol
MODULO : page 'p' is managed by processor H(p)=p mod N
BLOCK : page 'p' is managed by processor H(p)=p/N
int cc : cache coherency protocol
RW : strong coherency
RO : no cache coherency

Return value:

char *ptr : pointer to the shared region

To free a shared region, each processor calls the routine **shared_free**(). All processors wait at a barrier before freeing the shared region.

Synopsis:

shared_free(ptr, size)

Parameter declarations:

char *ptr : pointer to the shared region
int size : size of the shared region

Return value:

nil

5.2 Initializing a shared region

In order to initialize the shared region, data located on the host processor can be loaded into the shared region, using the routine **init_svm()** on the host processor and **load_svm()** on the node processors. such data could be vectors, matrices, etc... Figure 4 shows an example using a shared region which contains a matrix. In this example, the host processor store matrix *mat* which is constituted of 16 pages. Each node allocates a shared region (by calling **shared_alloc**() for matrix *mat* using the MODULO page ownership protocol. When the host processor calls **init_svm**() and nodes call **load_svm**(), pages are sent to nodes according to the page ownership protocol. For instance, processor 0 receives and is the owner of pages $0, 2, 4, etc...$ and manages them.

Synopsis:

init_svm(ptr, size, os)

Proc.	Time (ms)
2	0.4523
4	0.8892
8	1.3737
16	1.8904
32	2.4052

Table 3: Timing results for the *gsync*() routine.

Parameter declarations:

char *ptr : pointer to the shared region
int size : size of the shared region
int os : page ownership protocol

Return value:

nil

Synopsis:

load_svm(ptr, size, os)

Parameter declarations:

char *ptr : pointer to the shared region
int size : size of the shared region
int os : page ownership protocol

Return value:

nil

5.3 Synchronization tools

5.3.1 Barrier

The NX/2 system provides barrier synchronization. **All** processors can be synchronized, using the routine **gsync()**. Table 3 shows timing results for a barrier synchronization depending of the number of processors.

Synopsis:

gsync()

Parameter declarations:

nil

Return value:

nil

5.3.2 Managing critical sections

In a shared memory system, critical sections can be managed by using semaphores [11], a mechanism based on the use of a shared variable. However, this approach is not appropriate when the shared memory is implemented as page-demand shared virtual memory because the page which contains the shared variable will be subject to the ping-pong effect [22]. Thus, we have chosen a message passing approach based on Susuki and Kasami [23] algorithm which works as follows.

In this algorithm, the process that is in the critical section has a virtual token which represents its privileged status: it can stay in the critical section as long as it holds that token without consulting the other processes. The token is requested by process P_i by using a timestamped request message broadcast to all other processes. P_i does not know which process has the token. The token contains the serial number or timestamp of the last visit it made to each of the P_k processes. Once process P_j, which holds the token, no longer needs to execute in the critical section, it looks for the first process P_k such that the timestamp of the last request from P_k is greater than the timestamp stored on the token during its last visit to P_k. Token is then sent to process P_k which can enter in the critical section. We provide the following calls for managing critical sections: **get_lock()** is used to set up a critical section.

Synopsis:

get_lock(id)

Parameter declarations:

int id : critical section identifier

Return value:

nil

A process wishing to enter a critical section must call function **lock()**.

Synopsis:

lock(id)

Parameter declarations:

int id : critical section identifier

Return value:

nil

A process wishing to leave a critical section must call function **unlock()**.

Synopsis:

unlock(id)

Parameter declarations:

int id : critical section identifier

Return value:

nil

5.4 Measuring performance

Measuring performance is an important issue since it can be difficult to understand the behavior of programs implemented using a shared shared virtual memory. In order to help the user in this task, KOAN provides both performance analysis and software event tracing.

5.4.1 Performance analysis

During the execution of the user's parallel algorithm, the KOAN kernel continuously updates a set of counters which represent the number of times each processor has activated a handler or a server as well as the time that each processor has spent managing the shared virtual memory. The values of these counters are available in each processor by calling routine **svm_info()**:

Synopsis:

svm_info(info)

Parameter declarations:

struct info_svm info : statistics

Return value:

nil

The data structure **info_svm** is as follow:

```
struct info_svm {
        unsigned long read_fault;     /* Number of read page faults */
        unsigned long write_fault;    /* Number of write page faults */
        unsigned long count_lru;      /* Cache miss */
        unsigned long not_ready;      /* Number of calls to the not_ready server */
        unsigned long invalidation;   /* Number of invalidation received */
        unsigned long min_write;      /* Minimum time for getting a page (write access) */
        unsigned long min_read;       /* Minimum time for getting a page (read access) */
        unsigned long max_write;      /* Maximum time for getting a page (write access) */
        unsigned long max_read;       /* Maximum time for getting a page (read access) */
        double med_time_read;         /* Average time for getting a page (read access) */
        double med_time_write;        /* Average time for getting a page (write access) */
}
```

5.4.2 Software event tracing

KOAN can keep track of all events which occurred during the execution of a user's parallel algorithm. Each processor maintains its own trace of events where each of them is stamped by a local clock. A post-processing step merges all the local traces in a coherent list of events. Since software event tracing is time and memory consuming, management of traces is left to the user. The KOAN library offers several routines which help the user to manage these traces. Calling the **set_trace()** routine sets up a memory region to store the local trace:

Synopsis:

set_trace(start, size)

Parameter declarations:

char *start : pointer to a memory region used to store the local trace
int size : maximum number of entries in the trace

Return value:

nil

Event tracing is enabled by calling the **enable_trace()** routine:

Synopsis:

enable_trace()

Parameter declarations:

nil

Return value:

nil

Event tracing is disabled by calling the **disable_trace**() routine:

Synopsis:

disable_trace()

Parameter declarations:

nil

Return value:

nil

Local traces can be saved on the disk managed by host by calling the **file_trace**() routine:

Synopsis:

file_trace(fd)

Parameter declarations:

FILE *fd : file descriptor

Return value:

nil

A tool is provided to process all the local traces in order to merge them. It uses the algorithm described in [17] which build a global time on distributed memory parallel computer without disturbing the running of the user's algorithm.

5.5 A programming example: a parallel matrix algorithm

Let us describe an example of a parallel matrix multiply running with KOAN. The matrix multiply computes $C = AB$ where A, B and C are stored in the shared virtual memory. In this example, we use the SPMD programming model (Single Program Multiple Data). Each processor runs the same code shown in figure 5. The first step allocates a shared region (**shared_alloc**) the size of which is the sum of the size of matrices A, B and C. Then, matrices A and B are initialized with data coming from the host processor (**load_svm**, **init_svm**). Each processor computes a part of matrix C. The last step consists in freeing the shared region (**shared_free**).

6 Testing KOAN with parallel algorithms

This section deals with some experiments we have done with KOAN. We have investigated the parallelization of both a numerical algorithm (the modified Gram-Schmidt algorithm) and a non-numerical one (a ray-tracing algorithm). In the first algorithm, the relations between computations and data are known statically whereas they are unknown in the second algorithm. This section is organized as follows: section 6.1 describes some parallel programming methodologies whereas section 6.2 and section 6.3 are devoted to the description of experiments.

6.1 Parallel programming methodologies

Several programming methodologies can be applied to sequential algorithms in order to parallelize them.

A first approach consists of partitioning the data domain of the algorithm into sub-domains, each of which is associated with a processor. Computations are assigned to processors which own the data used by these computations. They are sent to processors by mean of messages. This approach is called *data-oriented* parallelization.

The second approach focuses on a functional decomposition. The original algorithm is split up into a set of functions. Each function is associated with a processor. Data flows between the processors. This kind of parallelism is called *pipelining* or *systolic computing*.

A third approach focuses on the parallelization of loops. Loops are analyzed in order to discover dependencies. A set of tasks are created that represent a subset of iterations. This approach requires that each task has access to the data. Operating systems associated with DMPCs do not offer shared memory services. Therefore, the user is responsible for adding communication primitives to allow a task to access remote data. This approach is called *control-oriented* parallelization.

Data-oriented and *systolic* parallelization are generally used for parallelizing algorithms on a DMPC since it requires only the exchange of messages. A *control-oriented* parallelization is not often used on DMPC because users think that emulating a shared memory is time intensive. The next sections show some results obtained by using *control-oriented* parallelization on both a numerical application (Modified Gram-Schmidt algorithm) and non-numerical application (ray-tracing).

6.2 Modified Gram-Schmidt

6.2.1 The Modified Gram-Schmidt algorithm

```
#include <svm.h>

#define N 100
#define M 100
#define MATSIZE N*M
typedef float vec[M];
vec *A, *B, *C;

main()
{
  int start, end, part;

  A = (vec *) shared_alloc(3*MATSIZE*sizeof(float), MODULO, RW);
  B = (vec *) ((unsigned int) A+MATSIZE*sizeof(float));
  C = (vec *) ((unsigned int) B+MATSIZE*sizeof(float));

  load_svm(A, 3*MATSIZE*sizeof(float), MODULO);

  part = (N / numnodes())
  start = part*mynode();
  end = start+part;

  for (i=start; i<end; i++)
      for (j=0; j<M; j++)
          for (k=0; k<M; k++)
              C[i][j] += A[i][k]*B[k][j];

  shared_free(A, 3*MATSIZE*sizeof(float));
}
```

Figure 5: Parallel matrix multiply running with KOAN.

```
for i = 1 : n
   A(1 : m, i) = A(1 : m, i) / || A(1 : m, i) ||_2
   for j = i + 1 : n
      A(1 : m, j) = A(1 : m, j) - A(1 : m, i)^T A(1 : m, j) A(1 : m, i)
   end
end
```

Figure 6: Modified Gram-Schmidt algorithm (MGS).

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

Figure 7: Simplified MGS algorithm.

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

Figure 8: Parallel MGS algorithm using control distribution.

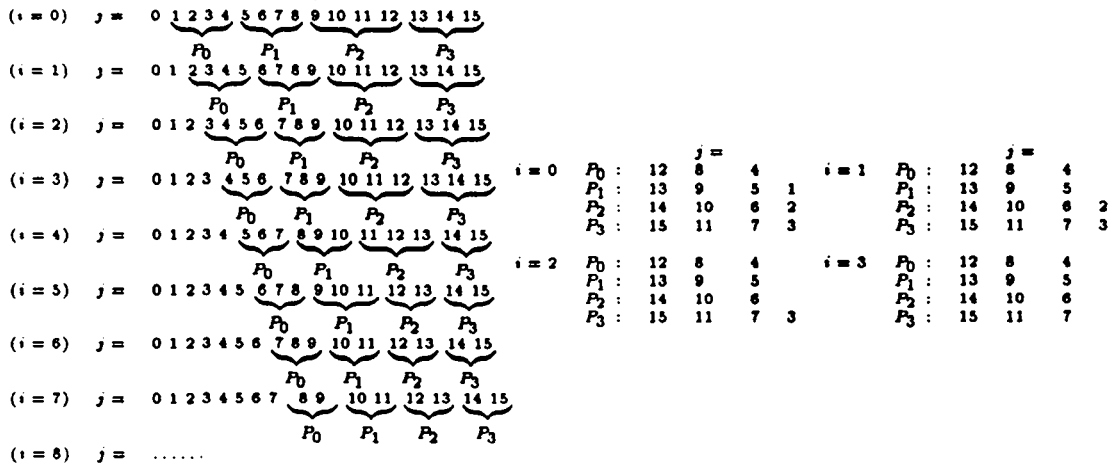
The Gram-Schmidt algorithm creates an orthogonal set of vectors of a given matrix [13]. As shown in figure 6, the algorithm is set up of two nested loops. The outer i-loop scans all the vectors of a given matrix, each vector is normalized and the subsequent vectors are updated by the j-loop. This algorithm can be be simplified by using the following notation given in [25] (figure 7)). $[i, i]$ means that vector i is normalized ($v_i = v_i/\|v_i\|$) and $[i, j]$ means that vector j is corrected with vector i ($v_j = v_j - (v_i, v_j)v_i$).

6.2.2 Parallelizing the MGS algorithm

To obtain a parallel implementation of the algorithm by *control-oriented* parallelization and running with a virtual shared memory, it is necessary to analyze its precedence graph. 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 8 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 $[i + 1, n - 1]$. Since the matrix is read and written from the virtual shared memory, the distribution strategy must take care to 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 $[i+1, n-1]$ as shown in figure 9-a. However, one can ascertain that between two successive steps of



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

Figure 9: Two distribution approaches.

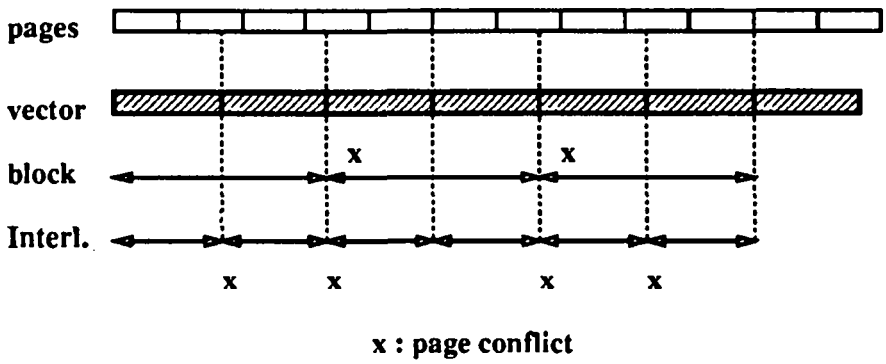


Figure 10: Page conflicts vs loop distribution

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 they have been used by processor P_1 at step $i = 0$. We can choose a second approach for improving temporal locality. It consist to interleave each element of the loop domain $[i + 1, n - 1]$ as shown in figure 9-b.

If we do not take care on 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 vectors. As shown in figure 10, loop distribution by interleaving may increase the number of page conflicts between processors. Therefore, in our first experiments, best results were obtained by distributing the j-loop by blocks. Figure 11 shows both the number of page faults and invalidations using the two loop distribution technics. In this case, the matrix size is 256 x 1892.

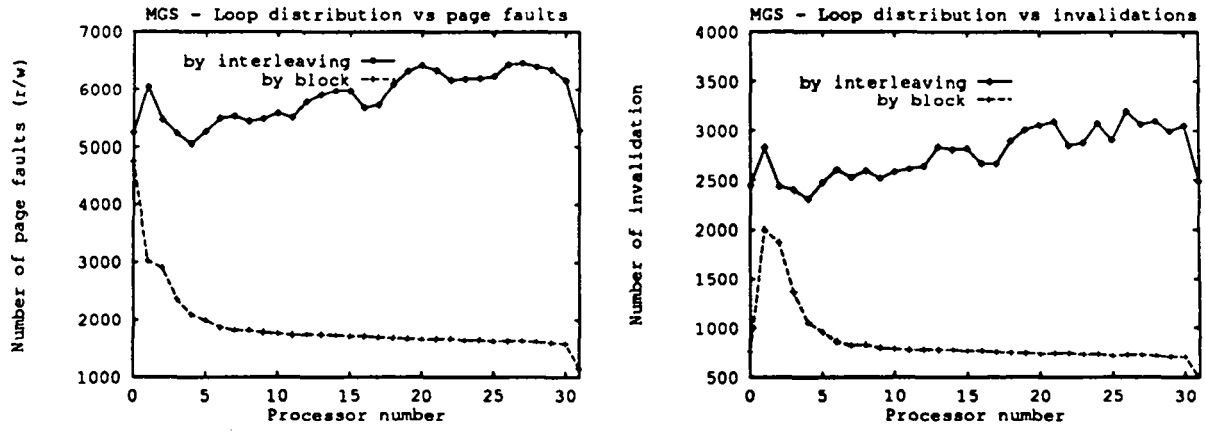


Figure 11: Page conflicts vs loop distribution

Vector size	Time (ms)	Speedup	Invalidation	Write Page faults	Read page faults	Total Page faults
1024	55956	12.32	4432	4432	11856	16288
1028	86684	7.98	24546	16323	31830	48153
1032	86504	8.03	24548	16202	31898	48100
1036	86975	8.02	24998	17046	32314	49360
1088	88190	8.31	24322	17424	31958	49382
1152	90094	8.61	23077	16746	31136	47882
1280	91183	9.45	20055	14433	29130	43563
1536	91864	11.26	15417	11693	26422	38115
1792	105598	11.43	21218	15169	34137	49306
1892	114944	11.09	27558	19450	41221	60671
2048	98415	14.03	8865	8865	23713	32578

Table 4: Benchmark results.

6.2.3 Results

First experiments have been done on matrices which allow us to evaluate different cases. Matrices are all square with size 512 x 512, 1024 x 1024 and 2048 x 2048. The size of the matrix element are 4 bytes. Concerning the storage of matrices in the shared virtual memory, there are either two rows into a page (512 x 512), either one rows into a page (1024 x 1024) or one row into two pages (2048 x 2048). Figure 12 shows that speedup in the 20 to 30 range are possible for large matrices. They have been obtained by using a loop distribution by block. Results are very encouraging except for the matrix with size 512 x 512. This is due by the fact that there are two rows in a page which can be used simultaneously by two different processors. This increases the number of page conflicts and adds a lot of overhead due to the invalidation protocol.

Next, we have studied performances on matrices where the number of rows is fixed ($N = 256$) and the row length is in the 1024 to 2048 range. Table 4 shows results according to the row length. We can ascertain that the speedup decreases when the end of rows does not fall in with the end of a page (for instance, when the row length varies from 1024 to 1028. In this case, several rows are stored in the same page. If two different processors modify simultaneously these rows, page

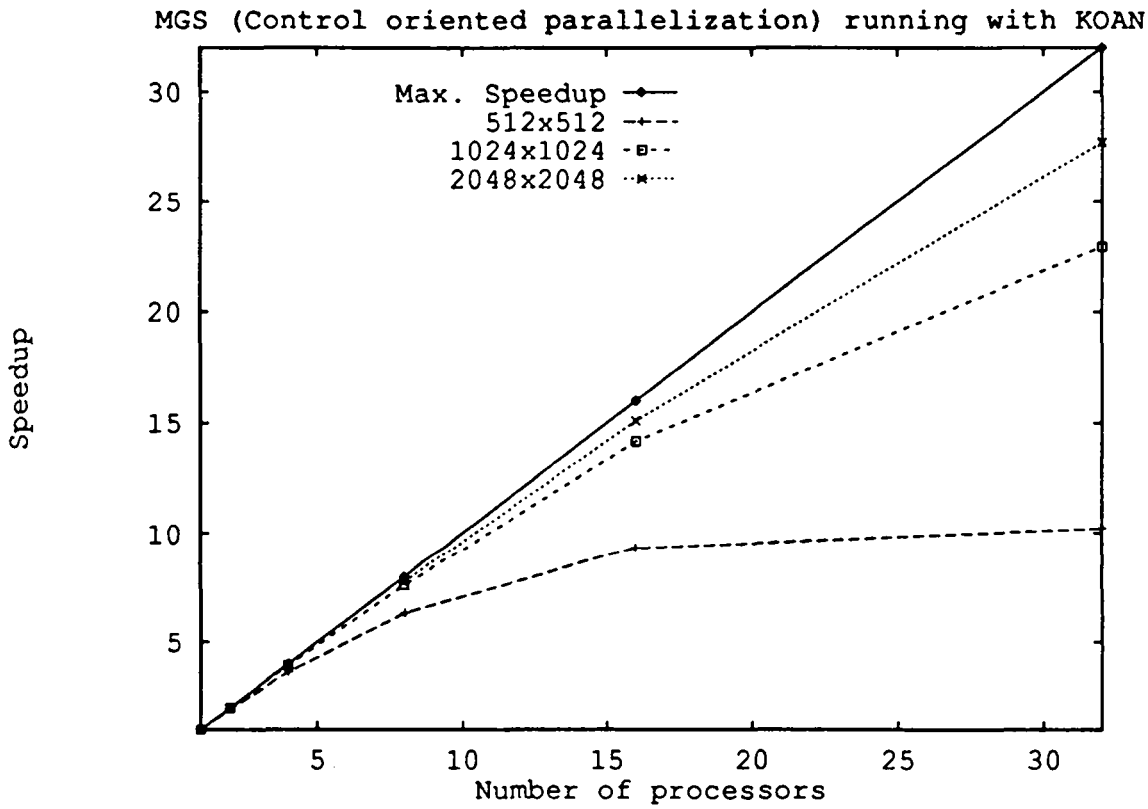


Figure 12: Speedup with different size of the matrix.

	Without code restructuring	with code restructuring
Speedup	7.98	10.7
Read page faults	31830	23714
Write page faults	16323	8865
Invalidations	24546	8865

Table 5: benefits of code restructuring.

faults and invalidations increase. Therefore, the number of messages increases in the network and adds a lot of overhead. This problem can be solved by using either a code restructuring or weak coherency.

The first solution consist in padding the matrix in order to put the beginning of each row of the matrix on a beginning of a page. This technic avoids page conflicts since there are not several rows in a page. However, a part of the virtual shared memory is lost. Another solution consists in using a weak coherency protocol as described in [12]. Since several processors have to write into different locations of the same page, we can let them to modify concurrently their own copy of a page. Giloi et al. have suggested to use two new constructs: **begin_weak** and **end_weak** which delimits 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 change. The user has to modify its parallel program in order to add these new construct at right places.

We have evaluated the benefits of code restructuring when the row length is equal to 1028. Results are given in table 5. They show that the number of page faults and invalidations decrease significantly and improve the speedup of the parallel algorithm.

6.3 Ray-tracing

6.3.1 The ray tracing principle

The ray tracing algorithm is used in computer graphics for rendering high quality images. It is based on simple optical laws which take into account effects such as shading, reflection and refraction. It acts as a light probe, following *light rays* in the reverse direction (Figure 13). The basic operation consists of tracing a ray from an *origin* point towards a *direction* in order to evaluate a light contribution. The closest intersection (*impact* point) between the ray and the scene determines the object, which contributes to this evaluation. The computation of each pixel of a simulated screen plane consists of shooting a ray from an *observer* through this pixel (*primary* rays). When an impact point is found, the contribution of various light sources to the intensity of the pixel are computed by shooting rays (*light* rays) from this point to each light source to determine if the relevant point is shadowed. According to the photometric properties of the intersected object, new rays are shot from the impact point, in order to take into account the contribution of neighboring objects [8, 15, 26]. If the object is transparent (reflective) a ray is shot in the refracted (reflected) direction (*secondary* rays).

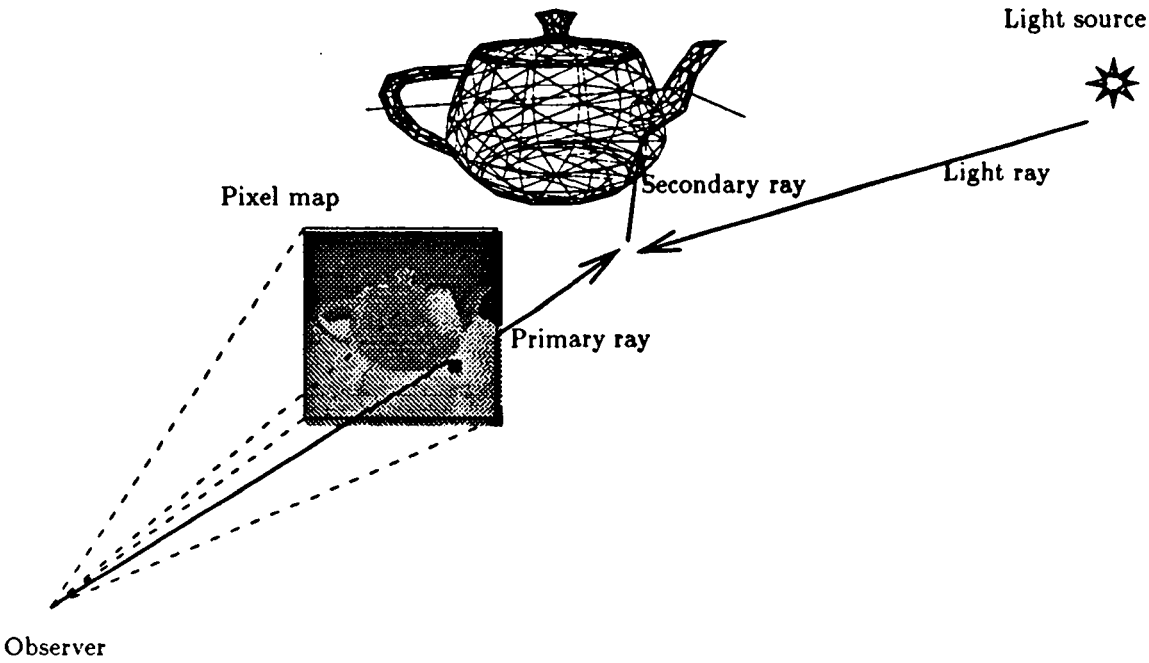


Figure 13: The ray tracing principle.

6.3.2 Parallelizing the ray-tracing algorithm

Ray tracing is intrinsically parallel since the evaluation of one pixel is independent of the others. The difficulty in exploiting this parallelism is to simultaneously ensure that the load be balanced and that the database be distributed evenly across the memory of the processors. The parallelization of such an algorithm raises a classical problem when using distributed parallel computers: how to ensure both data distribution and load balancing when no obvious relation between computation and data can be found. This problem can be illustrated by figure 14.

The computation of one pixel $pixel[i,j]$ is the sum of various light contributions $contrib()$ depending on the lighting model. Indeed, the recursive nature of the lighting model induces dependencies between the the various contributions to one pixel. $space[...]$ is the data structure associated with the space subdivision. The values of f_x, f_y and f_z are unknown before the computation. Searching for all the data $space[a,b,c]$ used for the evaluation of one pixel is equivalent to the ray tracing itself. Therefore, relationships between computations and data are unknown. Using *data-oriented* parallelization does not allow the computation of a data domain decomposition that ensures both correct data distribution and a balanced load [4]. This leads us to consider another kind of parallelism (*control-oriented*) which requires a shared virtual memory. This paradoxical approach for DMPCs can ensure both the data and the workload are evenly distributed. As described in the next section, the workload can be balanced dynamically during the running of the algorithm.

Distributing computations must ensure that each processor does roughly the same amount of work. This can be done by distributing pixels among processors. Two approaches can be used. The first (called *static scheduling*) consists in subdividing the screen into as many parts as

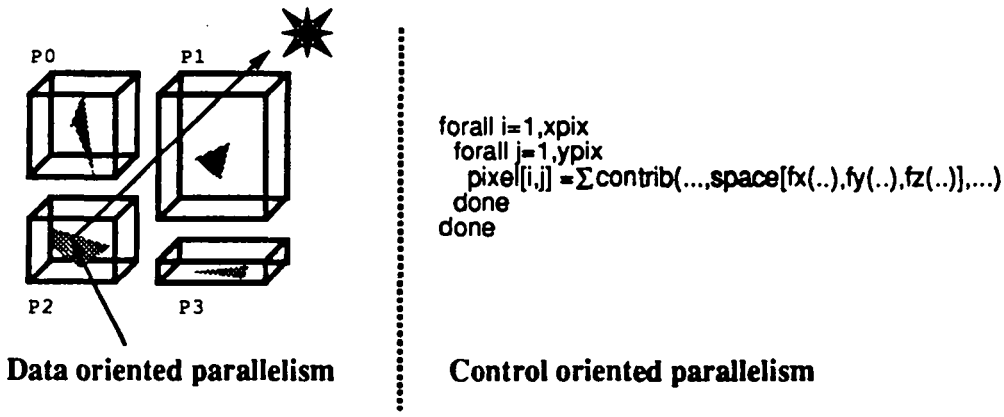


Figure 14: Parallelizing the ray-tracing algorithm.

Database	# polygons	# rays	Shared memory size
Teapot	3 754	1 397 473	793 Kbytes
Mountain	9 920	1 722 415	2 031 Kbytes
Rings4	18 002	1 872 991	4 632 Kbytes

Table 6: Databases characteristics and the rendering result.

processors. Each part is assigned to one processor. This approach is not satisfying because the computation times required by different pixels are not similar. Consequently, the workload is not equally distributed over the processors. The other approach (called *dynamic scheduling*) consists of assigning pixels to idle processors. As soon as a processor has completed the computation of a pixel, it asks a server for a new pixel. This solution ensures a balanced load but does not take into account the ray coherence property which can be used to reduce the number of remote data accesses. Ray coherence means that two rays shot from the observer through two adjacent pixels have a high probability of intersecting the same objects. This property is also true for all the rays spawned from these two primary rays. The dynamic scheduling technique does not ensure that the same processor treats neighboring pixels. Consequently we advocate the use of a mixture of the two techniques for reasons of efficiency. The static scheduling technique reduces the cost of remote data accesses whereas the dynamic scheduling technique solves the load balancing problem.

6.3.3 Results

Our experiments have been performed by using a set of scenes called *Standard Procedural Databases* (SPD) provided by Eric Haines [14] and the well-known *Teapot* from the University of Utah. These databases are presented in Table 6. Because of their different geometric and photometric properties, they constitute a representative test set.

Results in table 7 show that we obtain linear speedup and confirm results which has been achieved by emulating a virtual shared memory inside our parallel ray-tracing algorithm (VM_pRAY) [3]. Times decreased by a factor from 1.51 to 2.11 depending on the scene to compute. Translating virtual address physical one was done by software in the first experiment whereas it is now computed

1	2	4	8	16	32	
5927	2964	1483	743	372	187	*Teapot*
11042	5525	2766	1384	697	351	*Mountain*
		6075	2858	1413	704	*Rings4*

Table 7: Synthesis times (in seconds) with KOAN.

by the MMU with KOAN. Results obtained by using a shared virtual memory can be compared with those published in [20, 4]. They allow us to compare two different approaches: *data-oriented* parallelization and *control-oriented* parallelization. For scenes *rings4* containing several thousands of polygons, the first technique has a $O(log\, n)$ speedup while the second is in $O(n)$. These results can be explained by the fact that *data-oriented* parallelization has three major drawbacks: Load balancing is difficult to achieve since data accesses are unknown. Objects which are shared by several processors imply repeated intersection and decrease the efficiency of the algorithm when the number of processors increases. The last defect concerns the light source. A processor that possesses a light source in its associated region, receives a lot of messages from the other ones. This causes a network congestion and decreases dramatically the efficiency of the algorithm. As for *control-oriented* parallelization, it efficiently handles this situation since it uses cache memories. Indeed, a region that contains a light source will be frequently used by a processor and therefore it will be stored in the cache memory with a low probability to be overwritten. These drawbacks allow us to say that ray-tracing algorithms based on *data-oriented* parallelization (by messages) are not well suited for DMPCs.

7 Conclusion and future works

The works that are described in this paper represents a first step which consists in evaluating a shared virtual memory on distributed memory parallel computers. The first results are encouraging. Ongoing efforts are targeted at the comparison of various parallel implementations (data and control parallelism, functional decomposition) on DMPC for a representative set of numerical and non-numerical algorithms. We are currently in the process of designing a new release of KOAN which will allow us to compare different cache coherence protocols as well as different code restructuring technics. Finally, we are designing a high level software interface to KOAN in order to simplify the use of a shared virtual memory on the iPSC/2 hypercube.

Acknowledgements

We owe special thanks to D. Durand for her careful reading of this manuscript.

References

[1] V. Abrossimov, M. Rozier, and M. Gien. Virtual Memory Management in Chorus. In *LNCS*

: *Progress in Distributed Operating Systems and Distributed Systems Management*, Springler-Verlag, April 1989.

[2] J.K. Archibald. *The Cache Coherence Problem in Shared-Memory Multiprocessors.* PhD thesis, University of Washington, 1987.

[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] D. Badouel 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.

[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] John K. Bennett, John B. Carter, and Willy Zwaenepoel. *Munin: Shared Memory for Distributed Memory Multiprocessors.* Technical Report Rice COMP TR89-91, Rice University, April 1989.

[7] L.M. Censier and P. Feautrier. A new solution to coherence problems in multicache systems. *IEEE Trans. on Computers.*, C-27(12):1112–1118, Dec 1978.

[8] R.L. Cook and K.E. Torrance. A reflectance model for computer graphics. *ACM transactions on graphics*, 1(1):7–24, January 1982.

[9] P. Dasgupta, R. LeBlanc, and W. Appelbe. The CLOUDS Distributed Operating System: Functional Description, Implementation Details and Related Work. In *IEEE International Conference on Distributed Computing System*, 1988.

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

[11] E.W. Dijkstra. The structure of the multiprogramming system. *Comm. ACM*, 11(5):341–346, May 1968.

[12] 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.

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

[14] E. Haines. A proposal for standard graphics environments. *IEEE Computer Graphics and Applications*, 7(11), November 1987.

[15] A. Roy Hall and Donald P. Greenberg. A testbed for realistic image synthesis. *IEEE Computer Graphics and Applications*, 3(8):10–20, November 1983.

[16] P. Jacobson and E. Tarnvik. *A High Resolution Timer for the Intel iPSC/2 Hypercube.* Technical Report UMINF 91-04, University of Umea, Institute of Information Processing, Department of Computer Science, March 1991.

[17] J.M. Jézéquel. Building a Global Time on Parallel Machines. In 3^{rd} *International Workshop on Distributed Algorithms*, Springler Verlag LNCS no 392, 1989.

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

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

[20] T. Priol and K. Bouatouch. Static load balancing for a parallel ray tracing on mimd hypercube. *The Visual Computer*, 5():109–119, March 1989.

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

[22] Christoph Scheurich and Michel Dubois. Dynamic page migration in multiprocessors with distributed global memory. *Proceedings of the 8th International Conference on Distributed Computing Systems*, 162–169, 1988.

[23] I. Suzuki and T. KASAMI. An optimality theory for mutual exclusion algorithms in computer networks. In *Conf on Distributed Computing Systems*, oct 1982.

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

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

[26] T. Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 23:343–349, June 1980.

[27] 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.

LISTE DES DERNIERES PUBLICATIONS PARUES EN 1991

PI 582 PROGRAMMING REAL TIME APPLICATIONS WITH SIGNAL
Paul LE GUERNIC, Michel LE BORGNE, Thierry GAUTIER, Claude LE MAIRE
Avril 1991, 36 Pages.

PI 583 ELIMINATION OF REDUNDANCY FROM FUNCTIONS DEFINED BY SCHEMES
Didier CAUCAL
Avril 1991, 22 Pages.

PI 584 TECHNIQUES POUR LA MISE AU POINT DE PROGRAMMES REPARTIS
Michel ADAM, Michel HURFIN, Michel RAYNAL, Noël PLOUZEAU
Mai 1991, 10 Pages.

PI 585 TOWARDS THE CONSTRUCTION OF DISTRIBUTED DETECTION PROGRAMS, WITH AN APPLICATION TO DISTRIBUTED TERMINATION
Jean-Michel HELARY
Michel RAYNAL
Mai 1991, 24 Pages.

PI 586 OPAC : A COST-EFFECTIVE FOATING-POINT COPROCESSOR
André SEZNEC, Karl COURTEL
Mai 1991, 26 Pages.

PI 587 ON FAILURE DETECTION AND IDENTIFICATION : AN OPTIMUM ROBUST MIN-MAX APPROACH
Elias WAHNON, ALbert BENVENISTE
Mai 1991, 24 Pages.

PI 588 BOUNDED-MEMORY ALGORITHMS FOR VERIFICATION ON THE FLY
Claude JARD, Thierry JERON
Mai 1991, 14 Pages.

PI 589 UNE APPROCHE MULTIECHELLE A L'ANALYSE D'IMAGES PAR CHAMPS MARKOVIENS
Patrick PEREZ, Fabrice HEITZ
Juin 1991, 32 pages.

PI 590 THE IDEMPOTENT SOLUTIONS OF THE SEMI-UNIFICATION PROBLEM
Pascal BRISSET, Olivier RIDOUX
Juin 1991, 16 pages.

PI 591 AVARE UN PROGRAMME DE CALCUL DES ASSOCIATIONS ENTRE VARIABLES RELATIONNELLES
Mohamed OUALI ALLAH
Juin 1991, 32 pages.

PI 592 SCHEDULING IN DISTRIBUTED SYSTEMS : SURVEY AND QUESTIONS
Yasmina BELHAMISSI, Maurice JEGADO
Juin 1991, 36 pages.

PI 593 APPLICATION OF BELLEN'S PARALLEL METHOD TO ODE's WITH DISSIPATIVE RIGHT-HAND SIDE
Philippe CHARTIER
Juin 1991, 24 pages.

PI 594 PROGRAMMATION D'UN NOYAU UNIX EN GAMMA
Pascale LE CERTEN, Hector RUIZ BARRADAS
Juillet 1991, 48 pages.

PI 595 CALCULATING THE BUSY PERIOD DISTRIBUTION OF THE M/M/1 QUEUE
Louis-Marie LE NY, Gerardo RUBINO, Bruno SERICOLA
Juillet 1991, 11 pages.

PI 596 EFFICIENT CODE GENERATION FOR DISTRIBUTED MEMORY MACHINES*
Françoise ANDRE, Olivier CHERON, Jean-Louis PAZAT, Henry THOMAS
Juillet 1991, 14 pages.

PI 597 KOAN : A SHARED VIRTUAL MEMORY FOR THE iPSC/2 HYPERCUBE
Zakaria LAHJOMRI, Thierry PRIOL
Juillet 1991, 32 pages.

PI 598 KOAN : A VERSATILE TOOL FOR PARALLELIZING REALISTIC RENDERING ALGORITHMS
Didier BADOUEL, Kadi BOUATOUCH, Zakaria LAHJOMRI, Thierry PRIOL
Juillet 1991, 28 pages.

ISSN 0249-6399