View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by INRIA a CCSD electronic archive server

SVMview: a Performance Tuning Tool for DSM-based Parallel Computers

Didier Badouel, Thierry Priol, Luc Renambot

▶ **To cite this version:**

Didier Badouel, Thierry Priol, Luc Renambot. SVMview: a Performance Tuning Tool for DSM-based Parallel Computers. [Research Report] RR-2774, INRIA. 1996. inria-00073918

HAL Id: inria-00073918

https://hal.inria.fr/inria-00073918

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

SVMview: a Performance Tuning Tool for DSM-based Parallel Computers

Didier Badouel, Thierry Priol, Luc Renambot

N 2774

Janvier 1996

PROGRAMME 1

ISSN 0249-6399

SVMview: a Performance Tuning Tool for DSM-based Parallel Computers

Didier Badouel, Thierry Priol, Luc Renambot

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

Rapport de recherche n˚2774 — Janvier 1996 — 37 pages

Abstract: This paper describes a performance tuning tool, named SVMview, for DSM-based parallel computers. SVMview is a tool for doing a post-mortem analysis of page movements generated during the execution of Fortran-S programs. This tool is able to analyze some particular phenomena such as false-sharing which occurs when several processors write to the same page simultaneously. Such behavior entails badly the performance of DSM systems, and thus are particularly important to be detected.

Key-words:

(Résumé : tsvp)

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

SVMview: un outil de “performance tuning” pour MVP

Résumé : Ce papier décrit un outil de “performance tuning”, appelé SVMview, pour des architectures parallèles supportant une mémoire virtuelle partagée. SVMview est un outil permettant une analyse post-mortem des mouvements de page générés durant l'exécution d'un programme Fortran-S. Cet outil est capable d'analyser quelques phénomènes particuliers comme le faux-partage qui se produit lorsque plusieurs processeurs écrivent simultanément dans une même page. Il est particulièrement important de détecter ces comportements car ils induisent une dégradation rapide du système de mémoire virtuelle partagée

Mots-clé :

1 Introduction

Since 1985, the design of High Performance Parallel Processing Systems (HPPPS) is based mainly on distributed memory parallel computers (DMPCs). Such computers are made of several hundred of processors with their own local memories. However, exploiting the performance of such computers is not an easy task since the programming model is often seen as a breaking-through comparing to the one associated with sequential computers. Programmers have to manage explicitly message-passing between processors as well as to distribute data over the local memories which is considered as a low level programming similar in some respect to assembly language comparing to imperative language.

Several alternatives have been already investigated to program DMPCs using a high level programming model. Most of them aim at to provide a global address space implemented either by a compiler (Fortran-D [17], Vienna-Fortran [4], Pandore [1] and HPF [11]) or by an operating system (Shiva [21], KOAN [19], MYOAN [8], ASVM [27], MAX [15] or Treadmarks [22]). This latter approach extend the virtual memory concept to parallel architectures and is known under various names such as DSM (Distributed Shared Memory) or SVM (Shared Virtual Memory).

This idea came from the K. Li's work in his PhD thesis [20], since then a lot of works have been carried out in that area either for distributed systems (networks of workstations) or parallel computers (DMPCs). However, at the present time, none of the existing DMPCs provide a shared virtual memory layer even if the usage of such concept has been clearly demonstrated for some applications [3]. There are various reasons to explain such limited impact of innovating technology. Among them, the availability of a shared memory abstraction makes the programmer not to pay attention to data locality thus entailing bad performances. Contrary to the message passing, communication are implicit through page movements. Therefore, a programmer does not see effortlessly where communications occur in his program. Two other reasons have been pointed out recently by J. Carter in [9]. The first reason is that "*existing DSM systems are not well integrated with the rest of the software environment such as the compiler*". The second one is because "*there is a dearth of tools for debugging and tuning the performance of DSM programs*". These two reasons are closely related since tools for tuning performance of DSM systems often depend on the availability of programming environment. However, there are several on-going research works dealing with the design of programming environment for DSM systems: among them one can cite Fortran-S [6] and SVM-Fortran [5, 12].

This paper aims at describing a performance tuning tool for DSM systems named SVM-view. SVMview has been designed to analyze page movements generated by the execution of Fortran-S programs. It has been targeted to various parallel computers such as the iPSC/2 running the KOAN DSM [23] and the Paragon XP/S running either ASVM [27] or MYOAN[8]. These three DSM systems provide a monitoring support such as maintaining counters or generating events related to DSM activity.

The layout of this paper is as follows. Section 2 gives an overview of a programming environment we designed for DMPCs. Section 3 discusses several issues related to the per-

formance tuning of programs running with a DSM. Section 5 describes SVMview. Finally, section 8 draws some conclusions and perspectives to this work.

2 Background

This section gives an overview of a programming environment we designed for the Paragon XP/S. It is constituted of a DSM system for the OSF/1 MK-AD operating system and a parallel Fortran-77 compiler. In addition, we describe briefly the monitoring support of both the DSM and the code generator.

2.1 MYOAN: a DSM for the Paragon XP/S

MYOAN is a shared virtual memory system designed for the Intel Paragon supercomputer. It provides the same functionalities as KOAN which was originally designed for the Intel iPSC/2 [19]. An interesting feature of MYOAN is its ability to support multiple consistency protocols; both strong (atomic) consistency and a relaxed form of consistency, that permits multiple writers on the same page, are supported. MYOAN is an external pager of the MACH micro-kernel. It is implemented using a set of threads running concurrently with the user program. These threads communicate together using either NORMA IPC if communication is local to a node or NX/2 otherwise.

The default memory consistency protocol for memory regions in MYOAN is strong consistency. Each read at a given address returns the last value written to the same address. Strong consistency is maintained by implementing an invalidation-based protocol. A given page may be replicated in the nodes' physical memories only if it is protected against writes, *i.e.* its access mode is READ-ONLY, while pages in READ-WRITE access mode cannot be replicated. Should a process attempt to write to a READ-ONLY page, its replicas are invalidated before the write can proceed. In order to reduce the performance degradation due to the well-known false-sharing effect, MYOAN offers a relaxed form of consistency. This *multiple-writers* cache coherence-protocol allows processors to concurrently modify their own copy of a page. At a synchronization point, all the copies of a page which have been modified are merged into a single page that reflects all the changes. In addition, MYOAN provides a page broadcast mechanism to avoid contention when several processors request the same pages (producer-consumer scheme).

MYOAN has a monitoring support to provide a detailed feedback of the execution of parallel programs to the users. It provides both counters and events related to DSM management. Counters give a global overview of the DSM activity which have the advantage not to consume a lot of memory space. However, such information are not adequate if a pinpoint analysis is required. For such detailed analysis, the best way seems to gather DSM events. However, the storage of all events occurring during a parallel execution is highly memory consuming. MYOAN provides the ability to trace events related to the management of pages such as the activation of the page fault handler and servers that process incoming page requests and page invalidations. These events are stored in a temporary user buffer in

each node. The flushing of buffers into files has to be done explicitly by the programmer via a call to a subroutine. Such operation has to be carried out frequently to avoid buffer overflow.

2.2 Fortran-S: a code generator for DSM systems

Fortran-S [6] is a code generator targeted for DSM parallel architectures. A set of directives provides the programmer with a simple programming model based on shared array variables and parallel loops. One of the features of Fortran-S is its SPMD (Single Program Multiple Data) execution model that minimizes the overhead due to the management of parallel processes. The Fortran-S code generator creates a unique process code to be run by every processor for the entire computation duration. There is no dynamic creation of processes during the execution. Shared variables are mapped into distributed shared memory regions provided by a DSM system. Parallel loops can be distributed among the available processors using several predefined scheduling policies. One of them is the affinity scheduling to allow global distribution of loops within the application code in order to exploit data locality.

In addition, Fortran-S provides some directives to handle specific behaviors such as false-sharing or producer-consumer scheme. Such directives take benefit of the functionalities offered by the DSM system. During the execution of a parallel loop, a weak cache coherency can be selected, using an appropriate directive, to avoid the *false-sharing* effect. Likewise, a producer-consumer scheme can be efficiently handled through the use of a pair of directives which delimit the code section where pages have to be broadcast. In like manner of other control-oriented parallel fortran, Fortran-S offers several directives to specify reduction operations or synchronization such as critical section, atomic update or events.

Fortran-S provides several directives for monitoring support. Among them, a pair of directives can define a code section where an analysis has to be done. During the execution of this code section, events are generated both from the Fortran-S runtime library and the underlying DSM. Concerning the runtime, the following events are considered: shared variable mapping, starting and ending for both sequential and parallel code section and synchronization.

3 Analyzing parallel programs running with a DSM

The combination of a DSM system with a "high-level" programming model, such as MYOAN and Fortran-S, allows an incremental approach of the parallelization step. Since the parallel programming model is based on the use of directives, the programmer can simply modify or add new directives either to parallelize a new part of the code or to optimize a part which have been already parallelized. The optimization is a loop-based analysis process (see figure 1) where a programmer modifies his source code, runs it on the parallel computers, and then locates where bottleneck occurs using a performance tuning tool. During the execution, events are gathered to be visualized using a dedicated tool in a post-mortem manner. Depending of what kind of bottlenecks have been discovered, a programmer may apply some

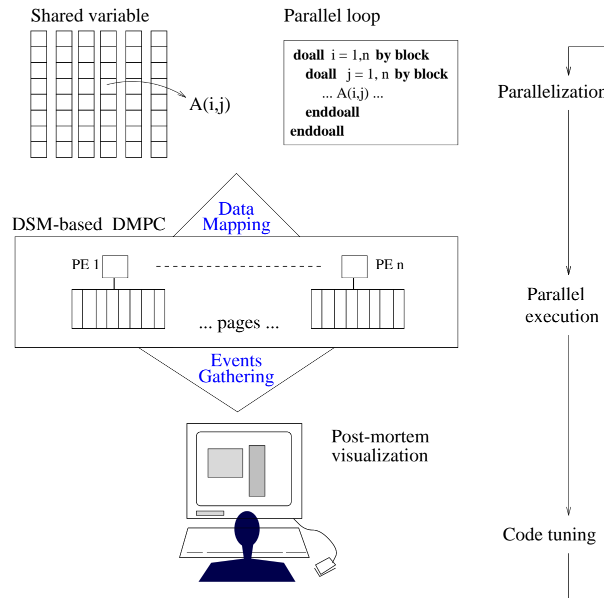


Figure 1: Performance tuning of DSM-based parallel programs.

modifications and then execute again his code. In order to ease the program optimization task, a performance tuning tool for DSM systems has to focus mainly on three aspects: communication, load balancing and synchronization. These issues are discussed in the following sections.

3.1 Communication bottlenecks in DSM system

Optimizing a parallel program on a DSM system requires a suitable knowledge of the page movements which occurred during its execution. As stated in the introduction, a DSM system hides the communication for the programmer. If such feature is desirable since it simplifies the writing of parallel programs, it entails a more complex task when analyzing the execution. Considering irregular problems such as sparse computation, this task is even more complicated. Improving the performance need to keep the amount of communication as low as possible. Such communication, which is based on message-passing for software-based DSM, are generated when a processor request a page from another processor or when it has to invalidate stall copies of a page located in some other processors. Each of these operations requires the sending of messages between the faulting processors, the manager and the owner of the requested page. Therefore, reducing page movements may improve tremendously the performance of parallel programs, especially if the underlying communication layer is slow or when the DSM implementation is made outside the OS kernel (like the MYOAN DSM).

Several data access patterns may increase the amount of communications generated by the DSM system. Cache management, due to the limited size of the cache, may decide to remove a copy of a page that is still needed for further computations. This well-known problem in virtual memory management is called page trashing. Another phenomenon is false-sharing. It occurs when several processors are writing simultaneously to the same page. Due to the strong cache coherence protocol, such page will move back and forth between processors. This will increase enormously the number of communications between processors. Several solutions can be applied to solve this problem such as array padding, specific loop scheduling strategy, compiler optimizations [14] or the use of specific cache coherence protocols [18]. Depending on the degree of false-sharing and the way the program has been parallelized, one of these optimizations is probably better than the others. In order to decide which of these optimizations has to be used, a fine analysis of this phenomenon is required.

Another severe drawback of a DSM system is its lack of support to manage efficiently a producer-consumer scheme. Our experiments have shown that such data access behavior can entail very bad performances when the number of processors reaches a given value. Such value depend both of the target architecture and the implementation of the DSM system. To tackle this problem, we implemented a page broadcast mechanism in our DSM systems (KOAN and MYOAN) as well as a language support within Fortran-S. A performance tuning tool has to offer a way to look at such problem.

3.2 Load balancing

The existence of a DSM system encourages the use of a control-oriented parallelization. An iteration space of a loop, with no data dependence, can be distributed among the available processors. Static or dynamic scheduling strategies can be used to balance the load. However, some of the scheduling strategies may generate more communication traffics between processors than the others. As for instance, a scheduling strategy based on cyclic distribution of the iteration space, is sometimes well suited to distribute the load when the amount of computations depends on the iteration value. However, such strategy may have the potential drawback of increasing the false-sharing effect. A block distribution strategy may badly distribute the load whereas it allows to keep low the amount of false-sharing. Another strategy based on affinity scheduling may have the advantage of reducing the page movements by exploiting the underlying temporal locality of the programmer code. However, this may lead to a poor load balance. A trade-off has often to be found in order to select the scheduling strategy offering the maximum performance (i.e. a few page movements with a good balance of the load). A performance tuning tool has to provide the programmer with the possibility to analyze page movements and load balance in connection with a scheduling strategy.

3.3 Synchronization

When a SPMD execution model is considered, barrier synchronizations play an important role to synchronize processor activities. The Fortran-S compiler adds automatically barrier synchronizations before and after the execution of a parallel loop to ensure the correctness of

Event	**Description**
MYOAN-read-page-fault	page fault when reading
MYOAN-write-page-fault	page fault when writing
MYOAN-invalidation	invalidation received
MYOAN-change-perm	change permission from write to read
MYOAN-page-dist	initial page distribution in local memory

Table 1: List of MYOAN events.

the program execution. Removing useless barrier synchronizations is part of the optimization process since some of them are not really needed depending of data dependencies. Therefore, a performance tuning tool must inform the programmer about the synchronization cost. The programmer may then take the decision to remove some barrier synchronizations if they are unnecessary. For this purpose, Fortran-S provides several directives to inform the code generator to suppress barrier synchronization before or after a parallel loop or both.

4 Event data collection

In our programming environment, three software components are involved during the execution of a parallel program: a DSM system (MYOAN), a runtime library which implements some services for the SPMD code generated by Fortran-S and the SPMD code itself. Each of these software components have to be instrumented to log specific events. Such events provide their contribution to the analysis of communication bottlenecks, load balancing or synchronization issues as discussed in the previous sections.

4.1 DSM events

Among the different software components, the DSM system is the one that will generate most of the communication during the execution of the program to be analyzed. During the solving of a page fault, various communications are needed between several processors (the faulting processor, the manager and the owner of a page). A first approach could be the generation of events for every communication activity. Such approach will generate a lot of events that are not always useful for a typical programmer which is not aware of the low-level details of the DSM protocol. Our approach focuses on the logging of events which are meaningful for the programmer. Such events correspond either to a page fault when reading from or writing to a page or to a change in the access right. Table 1 lists the most important events collected by the DSM system (a complete list of events is provided in appendix A).

For the page fault event, several information are recorded such as the processor identifier where the event occurred, the shared memory region identifier that contains the faulting address, two time stamps to specify the time when the page fault occurs and the one when

Event	Description
RT-wait-barrier-start	start waiting at a barrier
RT-wait-barrier-stop	stop waiting at a barrier
RT-wait-lock-start	start waiting at a lock
RT-wait-lock-stop	stop waiting at a lock
RT-wait-crit-sec-start	start waiting at a critical section
RT-wait-crit-sec-stop	stop waiting at a critical section
FS-parallel-loop-start	start the execution of a parallel loop
FS-parallel-loop-stop	stop the execution of a parallel loop
FS-sequential-sec-start	start the execution of a sequential code section
FS-sequential-sec-stop	stop the execution of a sequential code section
FS-set-proc-seq	proc. number executing the sequential code section
FS-shared-region-spec	shared memory region specification

Table 2: List of Fortran-S events.

the page fault has been processed, the faulting address, the faulting page number and the processor identifier that sends the page.

To make possible the analysis in any part of the program, it is necessary to have a pinpoint view of the page distribution at the beginning of the execution of the code section to be analyzed. A special event (MYOAN-page-dist), can be generated on request on every processor to know exactly the page distribution for a particular shared memory region.

Despite the fact that we decided not to generate an event for every communication, the execution of a parallel program may generate a huge number of events. This is especially true when an inexperienced programmer is parallelizing his sequential code for the first time. Events have to be managed carefully in order not alter the program behavior and not to lose the collected events. Our design choice consists in letting the programmer to allocate a buffer in the user space of the local memory associated with each processor. Buffer address is then communicated to the DSM for storing the events. Buffers are managed using a round-robin strategy. In case of buffer overflow, a special event is stored in the buffer. The programmer is responsible to allocate a sufficient buffer size to avoid buffer overflow, otherwise the analysis will then be impossible.

4.2 Runtime events

The SPMD code generated by Fortran-S makes call to a runtime library for several services such as shared memory allocation, message-passing communication, synchronization, loop distribution, etc... The runtime library has been instrumented to collect information needed to know the mapping of Fortran shared variables to shared memory regions, to estimate the cost of synchronization as well as to provide information about load balancing.

Table 2 gives an overview of the principal events that are generated by the runtime library (a complete list of the events are provided in appendix B).

One of the main motivation in designing a performance tuning tool for DSM was to integrate it within the environment. It means that a relationship between DSM events and the programming objects (shared variable, parallel loop) has to be shown to the programmer. Considering this objective, we included in the trace an event which specifies the memory mapping of a shared variable to a shared memory region managed by the DSM. Such event has the following parameters: variable name, variable type, virtual mapping address, number of dimensions and the parameters for every dimension. With such information, a DSM event, like a page fault, can be identified to the access of an element in the shared variable which has made such page fault. Moreover, it can be located within a sequential code section or within a parallel loop where it has occurred since events are generated when entering/leaving a sequential code section or a parallel loop.

As for the MYOAN events, runtime events are stored into buffers which are managed using a round-robin strategy. In case of buffer overflow, a special event is stored in the buffer.

4.3 SPMD code instrumentation

Fortran-S provides two directives (C$ann[StartTrace()] and C$ann[EndTrace()]) to specify a code section where an analysis has to be done. The code generator makes very few modification to the SPMD code. The main modifications concern the adding to some function calls to open and to close trace files, to inform both the runtime and the DSM system to generate events. Since the code generator is able solely to provide the information concerning the variable name and its associated type, several function calls are also added for each shared variable allocation. Before the code section to be analyzed, we add a function call to make a snapshot of the contents of the cache located in the local memory of every processors. An event will thus be generated indicating, for each shared memory region, the access rights for every pages.

5 SVMview: an DSM-based performance tuning tool

5.1 Rationale

Tuning the performance of a parallel code requires a precise knowledge of its execution. Such knowledge can be taken out from the trace files which contains specific events like the ones described in the previous section. Unfortunatly, the number of events are so huge that a regular programmer is unable to extract useful information from the traces to take proper actions. Visualization techniques can help the programmer to have a better understanding by mapping the events into graphical views. To be pertinent, such graphical views cannot be independant from the programming model which has been used to write the parallel code. They have to be adapted to a particular parallel programming model.

Several performance tuning tools have been designed in the past, but most of them focused mainly on performance tuning for message-passing parallel codes like Paragraph [16] or AIMS [26]. None of these tools can provide graphical views adapted to our requirements (*i.e.* able to show the DSM activity with a relationship to the high-level programming source code). Moreover, few of them are really available. Instead of modifying an existing tool, another approach consists in using a generic tool which can be targeted for a specific need. Pablo [24] is one of such tool. Contrary to other tools, Pablo is easily extensible. It provides a set of standard graphical views which can be used through a graphical language. This language describes the processing of the trace file through a graph where each node represents a specific operation or visualization to be applied to the input data coming from ancestors in the graph. Pablo reads trace files using the Self-Defining Data Format (SDDF) [2] format which is a trace description language. Despite its programmability, we found Pablo difficult to use since the processing of our trace files require often complex computations which cannot be described easily with its graphical language. For these reasons, we decided to build a new tool, named SVMview, instead of modifying or using an existing one. The next sections give a description of the tool.

5.2 Overview of SVMview

SVMview has been designed as a platform to experiment with *post-mortem* visualization tools for DSM-based architectures. SVMview takes as input two SDDF trace files using the *DSM Trace Format* and the *Runtime trace format* described in the appendix. Events taken out from the traces are passed to several graphical displays according to their types. SVMview allows the programmer to analyze DSM events linked to Fortran-S programming objects such as shared variables, sequential code section and parallel loops. As for instance, at any time, SVMview is able to inform the programmer that an event related to a DSM activity is related to a shared variable and where these events occurred in the code.

Figure 2 shows different graphical views from SVMview. A *monitor* window (at the upper left) allows to play back the execution and dispatches the events to the graphical views. SVMview has a specific graphical view to display the Fortran-S source code. This graphical view (at the upper right) aims at linking the execution steps to the Fortran-S source code. It allows the programmer to know exactly from what part of the Fortran-S source code the events, which are displayed in other graphical views, have been generated. Within the editor window, several areas are active, which means that the programmer can select a directive specifying a shared variable or a parallel loop to get some detailed information such as, for instance, the number of pages to map a given shared variable.

The *pages state* window provide the access rights (read-write, read-only, invalid) of every pages in the local memory of each processor using specific colors. During the execution playback, the programmer can see the access right modifications and page movements in an animated way. Several communication bottlenecks can be discovered using this view. However, specific bottlenecks such as false-sharing or producer-consumer can be identified using proper graphical views which are described in the following sections.

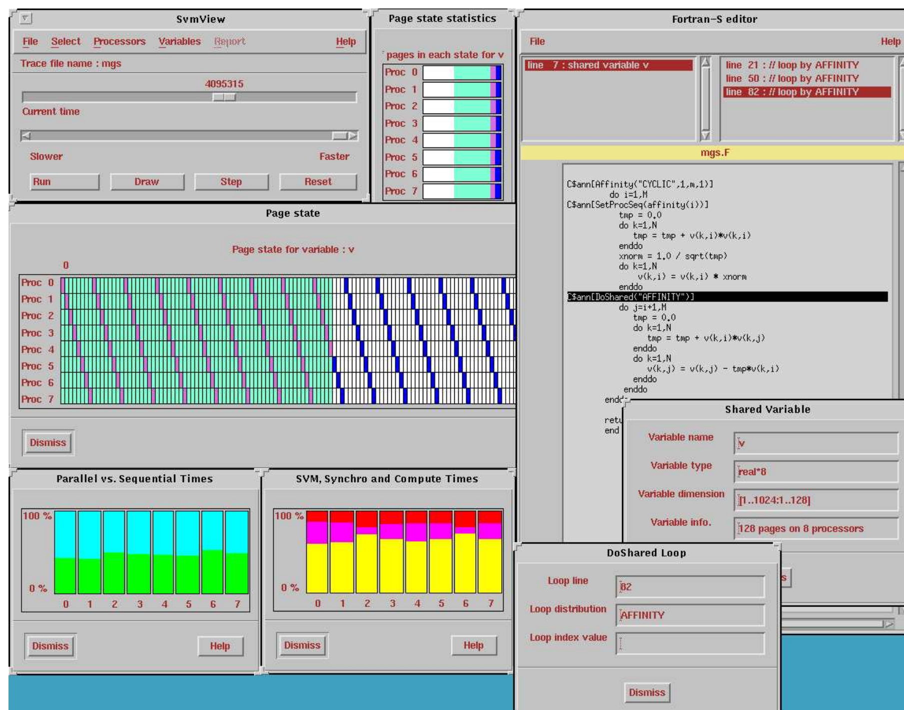


Figure 2: SVMview and some of its graphical views

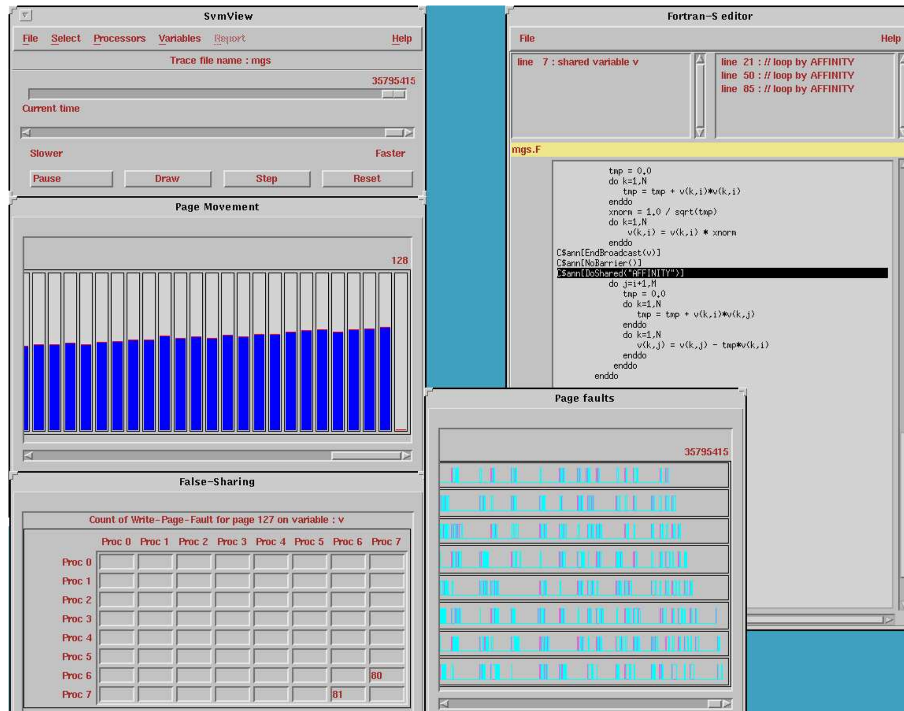


Figure 3: Detecting false-sharing with SVMview.

Many other possible views exist but are not displayed in figure 2. Generally, the views are based on classical graphical displays such as bar-graphs (to display information related to synchronization and load balancing issues), Gantt charts, Kiviat diagrams and spreadsheets tables. Most of the graphical components we used in SVMview comes from the Pablo environment [24] or from the Xbae toolkit. SVMview is also able to supply data to external applications such as MAPLE for specific processing.

5.3 False-sharing detection

A false-sharing effect occurs when several processors repeatedly attempt to write to the same page. In order to enforce the coherence for the modified data, the write operations are serialized by means of an invalidation-based protocol. The consequences of such protocol is that the incriminated page moves back and forth between the concurring nodes. SVMview can both quantify and graphically exhibit this phenomenon as illustrated by figure 3. The graphical views, shown here, correspond to different levels of interpretation offered to the programmer to analyze traces related to a parallel loop:

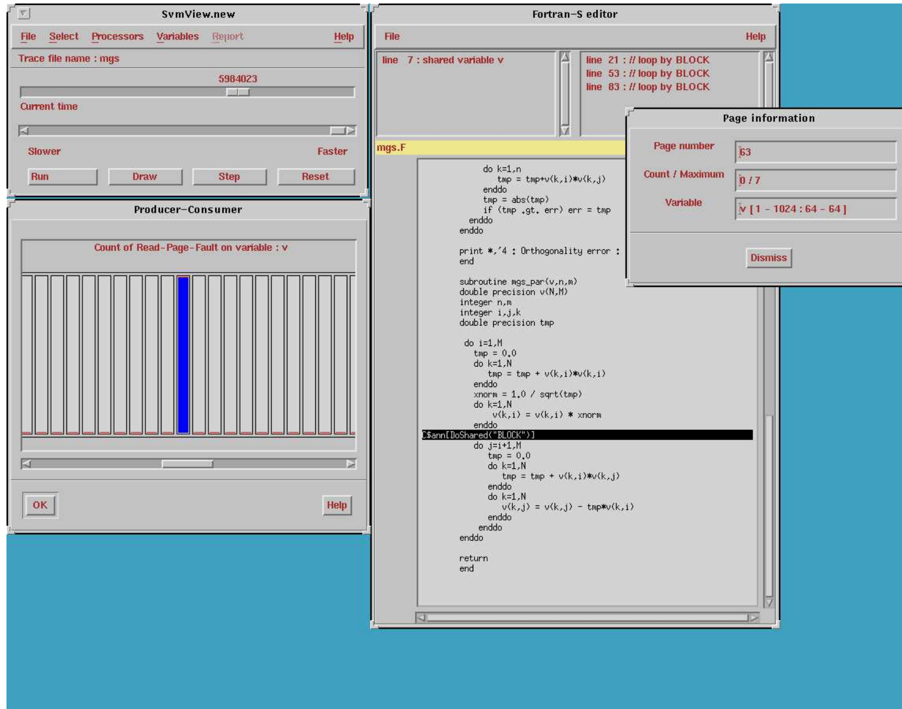


Figure 4: Detecting producer-consumer with SVMview.

1. the low-level **Page Faults** view only points out to some execution sections where the number of pages exchanged between nodes is important;
2. meanwhile, the **Page Movement** view allows to tell which pages are particularly involved in the traffic;
3. finally, the **False Sharing** view quantifies the false sharing effect for a given page.

The false sharing effect is measured by means of a two-dimensional array of counters. These counters are updated within a parallel loop to record the number of write page faults for a given page and for every processors involved in the communication. Each row corresponds to the number of times the selected page was sent; each column corresponds to the number of times the selected page was received. Due to the Fortran-S programming model, the write page faults, occuring during the execution of a parallel loop, are mostly due to write operations to different location inside a page. For this reason, these counters are well-suited to characterize an execution behavior relatively to the false-sharing effect.

5.4 Producer-consumer phase detection

The producer-consumer effect is a different phenomenon characterized by two phases that a barrier separates. During the production phase (a sequential code section), one particular processor produces values by means of one or more write operations. During the consumption phase (in a parallel loop), several processors attempts to read the produced value(s). The consequences are the sending of multiple messages requesting a copy of a particular page to the producer processor. The intensity of the phenomenon can be characterized by the number of page replicas produced during the consumption phase. As illustrated by figure 4, SVMview offers a graphical view to show off the producer-consumer effect on a variable.

SVMview characterizes a producer-consumer effect by counting for each page the number of consumers. This is done by counting the read requests for each page when stepping into a parallel loop.

6 Case study: the Modified Gram-Schmidt algorithm

To illustrate the use of SVMview to tune the performance of a particular code, we selected a simple numerical algorithm: the Modified Gram-Schmidt algorithm. Given a set of independent vectors $\{v_1, ..., v_n\}$ in R^m, the Modified Gram-Schmidt (MGS) algorithm produces an orthonormal basis of the space generated by these vectors. The basis is constructed in steps where each new computed vector replaces the old one. Vectors, to be orthonormalized, are stored in a matrix. The MGS algorithm proceeds as follow: an outer loop selects a vector which is normalized, an inner loop applies corrections to the remaining vectors which follow the normalized vector. Since there is no data dependence within the inner loop, it can be distributed on different processors.

The following sections describe several versions of the parallel MGS algorithm starting from a naive version to a more effective one. We illustrate the use of SVMview to explain the drawbacks of each of these versions. Experiments were carried out on a Paragon XP/S running either MYOAN or ASVM.

6.1 Version 1: BLOCK loop distribution strategy

The first version of the parallel MGS algorithm is presented in figure 5. Each vector has 1024 double precision floating-point numbers. One vector is stored exactly into one page (8192 bytes), consequently there will be no false-sharing during the execution of the parallel loop.

Figure 6 shows the *pages state* view before (a) and during (b) the parallel loop execution. The virtual address space is displayed for each processor where each rectangle represents a page. Its shading gives the associated access right (see table 3).

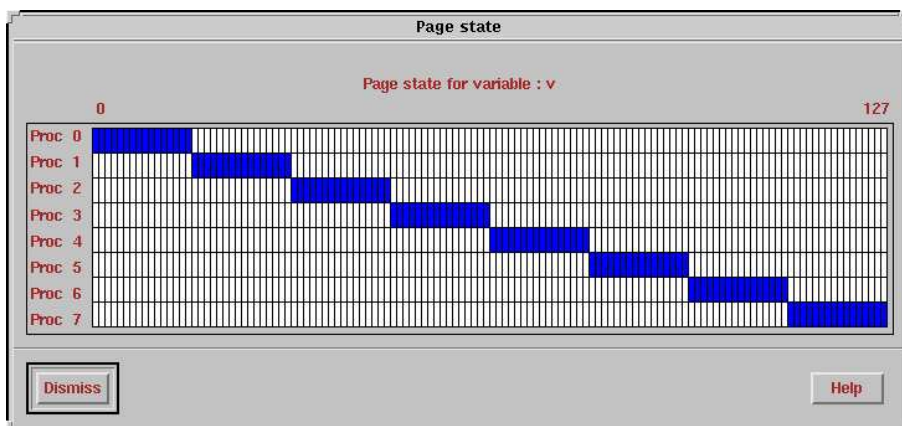
Before the execution (figure 6-a), pages are distributed among processors using a block strategy resulting from the matrix initialization. Each processor owns a block of contiguous pages where it can write to or read from. When running SVMview, the programmer can

```
      double precision v(1024,128)
C$ann[Shared(v)]
      ...
      do i = 1, m
         tmp = 0.0
         do k = 1, n
            tmp = tmp + v(k,i)*v(k,i)
         enddo
         xnorm = 1.0 / sqrt(tmp)
         do k = 1, n
            v(k,i) = v(k,i) * xnorm
         enddo
C$ann[DoShared("BLOCK")]
         do j = i + 1, m
            tmp = 0.0
            do k = 1, n
               tmp = tmp + v(k,i)*v(k,j)
            enddo
            do k = 1, n
               v(k,j) = v(k,j) - tmp*v(k,i)
            enddo
         enddo
      enddo
```

Figure 5: First version of parallel MGS.

Shading	State
White	Invalid
Light grey	Read-only not owner
Dark grey	Read-only owner
Black	Read-write owner

Table 3: Shading associated with page state.



(a) Page state before the parallel loop execution

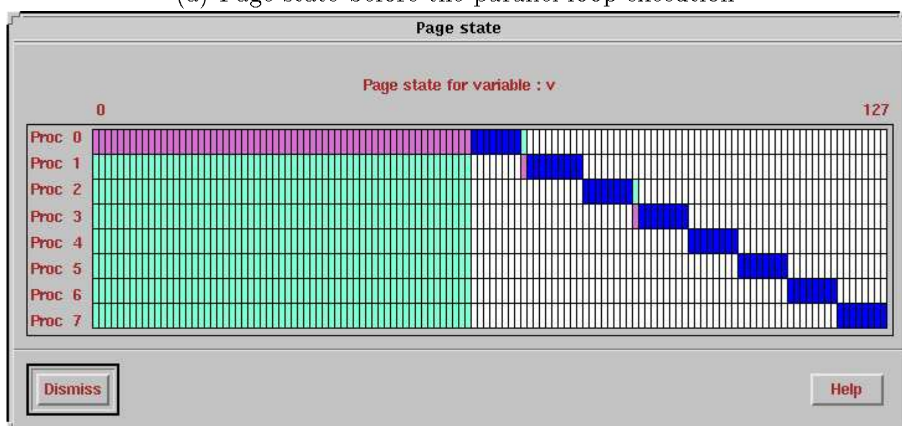


Figure 6: (b) Page state during the parallel loop execution (MGS1)

see the page movements as well as the access right modifications. Figure 6-b shows the pages state during the execution. A comparison of the two views shows clearly that the page ownerships have changed: for instance, processor 6 owns pages (with a read-write access right) that were previously located on processor 7. Such page movements can be seen for all the processors and they clearly reduce the performances. Such behavior can be explained by the block distribution strategy which has been used to spread the iteration space of the j loop. This iteration space is splitted each time the j loop is executed. As the lower bound depends on variable i, each processor does not process the same iterations when executing two successive iterations of the i loop.

As the execution proceeds, processor 0 becomes the owner of all pages. This is depicted in figure 6-b where processor 0 is the owner of roughly half the number of pages with a read-only access right. If the size of the virtual address space is greater than the size of the physical memory associated with processor 0, a page trashing will occur adding a large overhead in the execution time. This potential page trashing is due to the execution of the sequential code section (normalization) by processor 0.

The next section describes a new version of the MGS algorithm which solves the two problems which have been observed under SVMview.

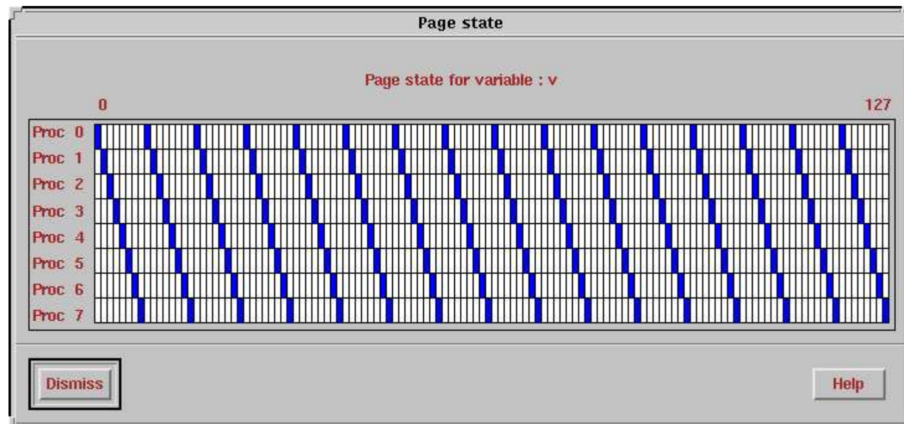
6.2 Version 2: AFFINITY loop distribution strategy

This second version (figure 7) uses an affinity scheduling to assign iterations to processors so that few page movements occur during the execution. Comparing to the previous version, several directives have been added. The Affinity() directive specifies the mapping of the iteration space onto processors (cyclic distribution). The SetProcSeq() directive fixes the processor number in charge of making the update to the shared variable v. Therefore, the normalization of vector i is carried out on the processor for which the iteration value i has been assigned by the affinity scheduling. Finally, the j loop is distributed using the affinity scheduling (DoShared directive). Whatever the value of i is, iterations of the j loop are always assigned to the same processor.

Figure 8 shows the impact of the new loop distribution strategy on page movements. It shows that the page ownership does not change during the execution indicating that the writings are local. However, there is still a drawback that was already present in the previous version. As shown in figures 6-b and 8-b, as the execution goes on, each processor get a copy of the page with a *read-only* access right. As for instance, in figure 8, the owner of page 0 is processor 0 whereas the other processors have a copy of that page. This is likely a producer-consumer effect. Figure 9 illustrates the use of SVMview to locate this producer-consumer phase in the MGS algorithm. In the lower left window, we can see that there is a producer-consumer effect on page 0 which means that a copy of a page has been requested by every processors making a memory contention on processor 0. The upper right window show where such phenomenon has been discovered. By looking into the code, such behavior can be explained as follow. Computations involved in the parallel loop require the access to the vector which has been normalized in the previous sequential code section. This memory contention can be removed using some specific directives as explained in the next section.

```
      double precision v(1024,128)
C$ann[Shared(v)]
      ...
C$ann[Affinity("CYCLIC",1,m,1)]
      do i = 1, m
C$ann[SetProcSeq(affinity(i))]
         tmp = 0.0
         do k = 1, n
            tmp = tmp + v(k,i)*v(k,i)
         enddo
         xnorm = 1.0 / sqrt(tmp)
         do k = 1, n
            v(k,i) = v(k,i) * xnorm
         enddo
C$ann[DoShared("AFFINITY")]
         do j = i + 1, m
            tmp = 0.0
            do k = 1, n
               tmp = tmp + v(k,i)*v(k,j)
            enddo
            do k = 1, n
               v(k,j) = v(k,j) - tmp*v(k,i)
            enddo
         enddo
      enddo
```

Figure 7: Second version of the parallel MGS algorithm.



(a) Page state before the parallel loop execution

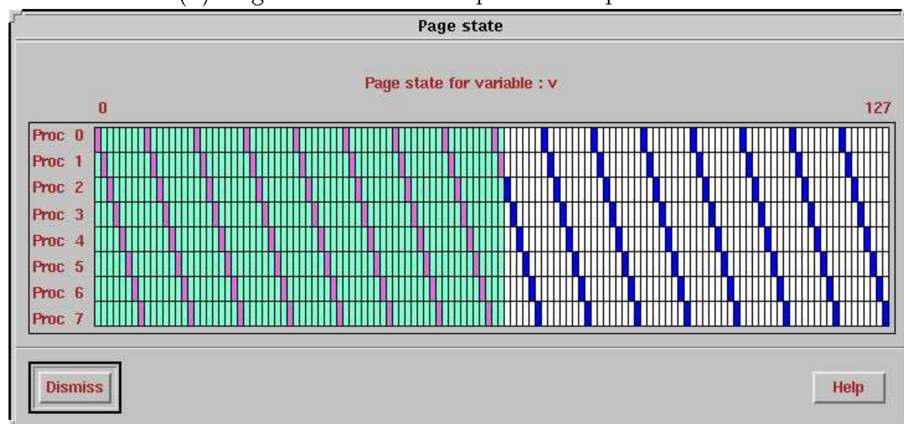


Figure 8: (b) Page state during the parallel loop execution (MGS2)

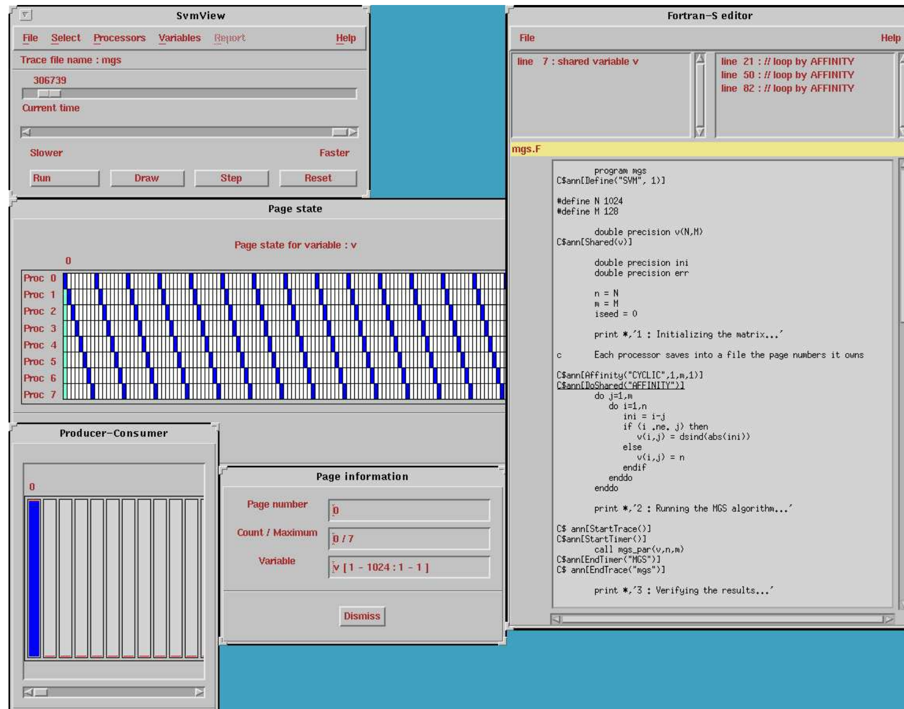


Figure 9: Detecting a producer-consumer scheme (MGS2).

```
        double precision v(1024,128)
C$ann[Shared(v)]
        ...
C$ann[Affinity("CYCLIC",1,m,1)]
        do i = 1,m
C$ann[SetProcSeq(affinity(i))]
C$ann[BeginBroadcast(v,v(1,i),v(N,i))]
           tmp = 0.0
           do k = 1,n
              tmp = tmp + v(k,i)*v(k,i)
           enddo
           xnorm = 1.0 / sqrt(tmp)
           do k = 1,n
              v(k,i) = v(k,i) * xnorm
           enddo
C$ann[EndBroadcast()]
C$ann[NoBarrier()]
C$ann[DoShared("AFFINITY")]
           do j = i+1,m
              tmp = 0.0
              do k = 1,n
                 tmp = tmp + v(k,i)*v(k,j)
              enddo
              do k = 1,n
                 v(k,j) = v(k,j) - tmp*v(k,i)
              enddo
           enddo
        enddo
```

Figure 10: Third version of the parallel MGS algorithm.

6.3 Version 3: Page broadcast

This new version (figure 10) is similar to the previous one in term of the loop distribution strategy (affinity). We added two directives in order to broadcast pages to processors to avoid the memory contention. These directives delimit the sequential code section where pages, in a given address range, have to be broadcast.

The page broadcast is carried out by message-passing either at a system level when this functionality is supported by the DSM (like MYOAN) or at a user level otherwise (like ASVM). Figure 11 shows the page states after the execution of MGS3. In this example, page broadcast has been carried out by message-passing at the user level. A temporary buffer is added to the generated code for storing the shared data to be broadcast. In addition, message-passing calls are added to broadcast the temporary buffer. In the remaining part of the program, the code is modified to let processors to use the temporary buffer instead of the shared variable until the latter is modified. Coming back to the MGS3 example, such technique explains why the *page state* view shows that there is no page movement during the execution.

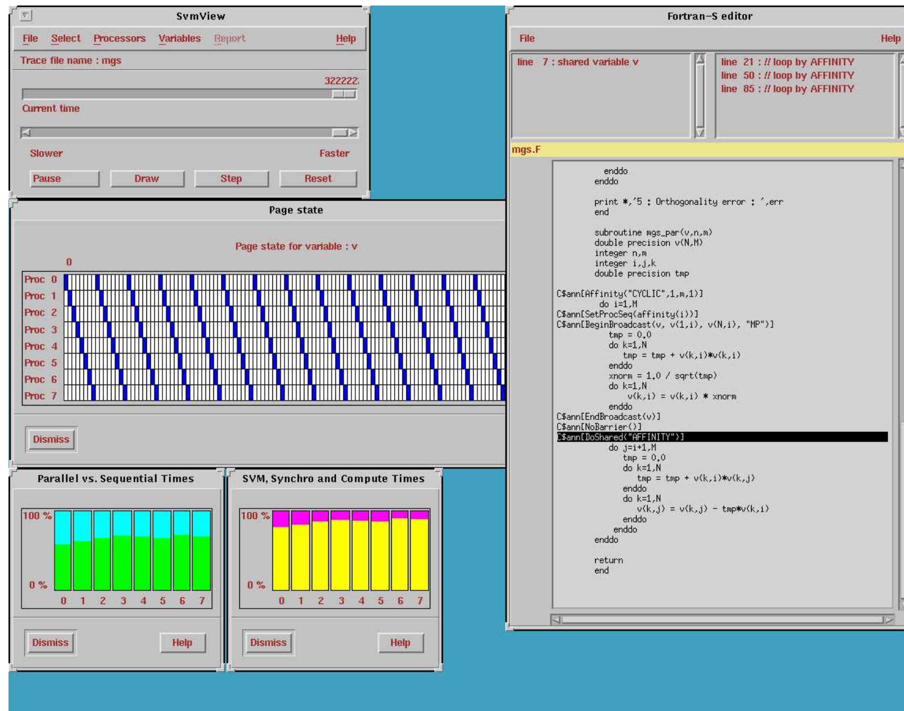


Figure 11: End of the parallel loop execution (MGS3)

The two lower views give some helpful information about the execution profile. The left view (*Parallel vs. Sequential Times*) shows, for each processor, the percentage of time spent in executing a parallel loop (dark grey) or a sequential code section (light grey). The right view (*SVM, Synchro, Compute Times*) supplies, for each processor, the percentage of time spent in waiting for pages (black), in executing barriers (dark grey) or in doing computations (light grey). These views show clearly that the load is well balanced among the processors and the synchronization cost remains low comparing to the computation. Moreover, the right view shows that there is no SVM activity (in fact, there is no black part in the displayed gauges).

In the three previous version of the MGS algorithm, the problem size has been set in such a way that the vector size is exactly identical to the page size. In the next section, we modified the problem size and run the third version of the MGS algorithm.

```
         double precision v(1025,128)
C$ann[Shared(v)]
         ...
C$ann[Affinity("CYCLIC",1,m,1)]
         do i = 1, m
C$ann[SetProcSeq(affinity(i))]
C$ann[BeginBroadcast(v,v(1,i),v(N,i))]
            tmp = 0.0
            do k = 1, n
               tmp = tmp + v(k,i)*v(k,i)
            enddo
            xnorm = 1.0 / sqrt(tmp)
            do k = 1, n
               v(k,i) = v(k,i) * xnorm
            enddo
C$ann[EndBroadcast()]
C$ann[NoBarrier()]
C$ann[DoShared("AFFINITY")]
            do j = i + 1, m
               tmp = 0.0
               do k = 1, n
                  tmp = tmp + v(k,i)*v(k,j)
               enddo
               do k = 1, n
                  v(k,j) = v(k,j) - tmp*v(k,i)
               enddo
            enddo
         enddo
```

Figure 12: Fourth version of the parallel MGS algorithm.

6.4 Version 4: Impact of problem size

When programming with a DSM, the problem size has a very sensitive influence on the performance of a parallel algorithm. By slightly modifying the problem size, the performance can decrease dramatically due to the false-sharing effect. To illustrate this problem, we added one element to each vector (figure 12). Therefore, a page stores two different vectors and this may generate a false-sharing effect when writing simultaneously to these two vectors. Figure 13 shows how to evaluate the impact of false-sharing using SVMview. The lower right view shows the counter values for page number 127. These values correspond to the number of write page faults that occured during the execution of the parallel loop for the total duration of the program execution. Page 127 has moved back and forth between processors 6 and 7. The lower left view show that the phenomenon occurs on every page except the last one. Despite the affinity scheduling to exploit data locality, jointly with a cyclic distribution to balance the load, false-sharing entails poor performance.

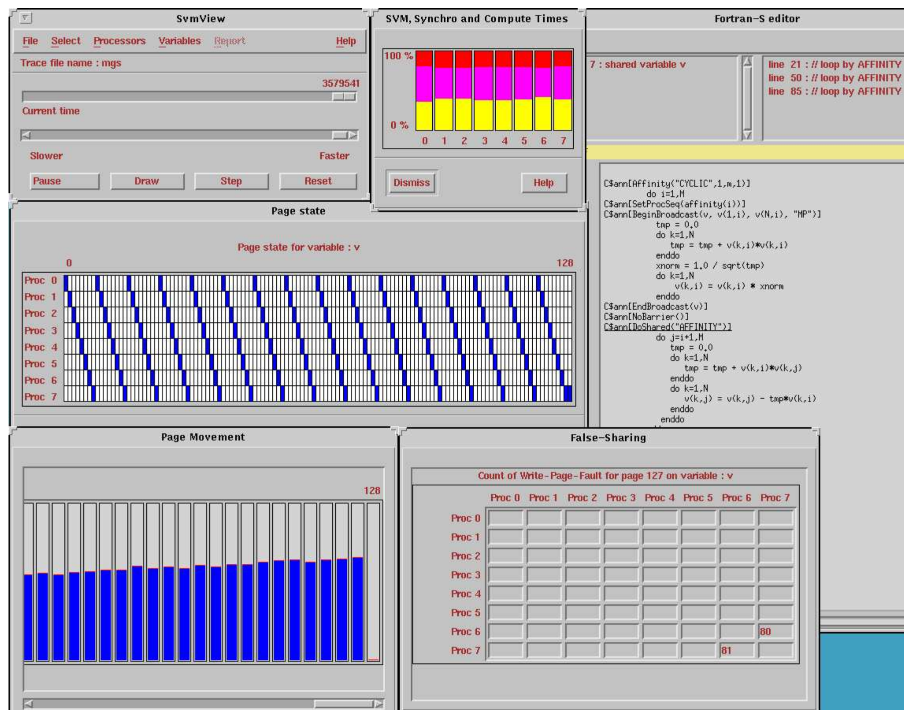


Figure 13: Visualizing the false-sharing (MGS4).

```
        double precision v(1025,128)
C$ann[Shared(v)]
        ...
C$ann[Affinity("BLOCK",1,m,1)]
        do i = 1,m
C$ann[SetProcSeq(affinity(i))]
C$ann[BeginBroadcast(v,v(1,i),v(N,i))]
           tmp = 0.0
           do k = 1,n
              tmp = tmp + v(k,i)*v(k,i)
           enddo
           xnorm = 1.0 / sqrt(tmp)
           do k = 1,n
              v(k,i) = v(k,i) * xnorm
           enddo
C$ann[EndBroadcast()]
C$ann[NoBarrier()]
C$ann[DoShared("AFFINITY")]
           do j = i+1,m
              tmp = 0.0
              do k = 1,n
                 tmp = tmp + v(k,i)*v(k,j)
              enddo
              do k = 1,n
                 v(k,j) = v(k,j) - tmp*v(k,i)
              enddo
           enddo
        enddo
```

Figure 14: Fifth version of the parallel MGS algorithm.

6.5 Version 5: Reducing false sharing

To increase the performance, we can either suppress the false-sharing effect or limit its impact by changing the parallelization strategy. The first approach consists in either padding the beginning of each vector on a page boundary or in using another cache coherence protocol which allows several processors to write to the same page simultaneously. For illustration purpose, we now investigate the second approach. Instead of using a cyclic distribution with the affinity scheduling, we use a block distribution strategy (figure 14). Therefore, each processor will be in charge of updating a set of contiguous vectors, thus making a false-sharing effect when updating the first and the last vectors within a parallel loop. Figure 15 shows the new situation: only few pages are subject to false-sharing: those pages which store two vectors updated by two different processors (pages with a black color in the *page movement* window).

However, there is another side of the picture: the load is not anymore well balanced among the processor as shown in the upper middle view. Processor 0 spends most of its time waiting at a barrier synchronization (dark grey) since it does not have enough computation.

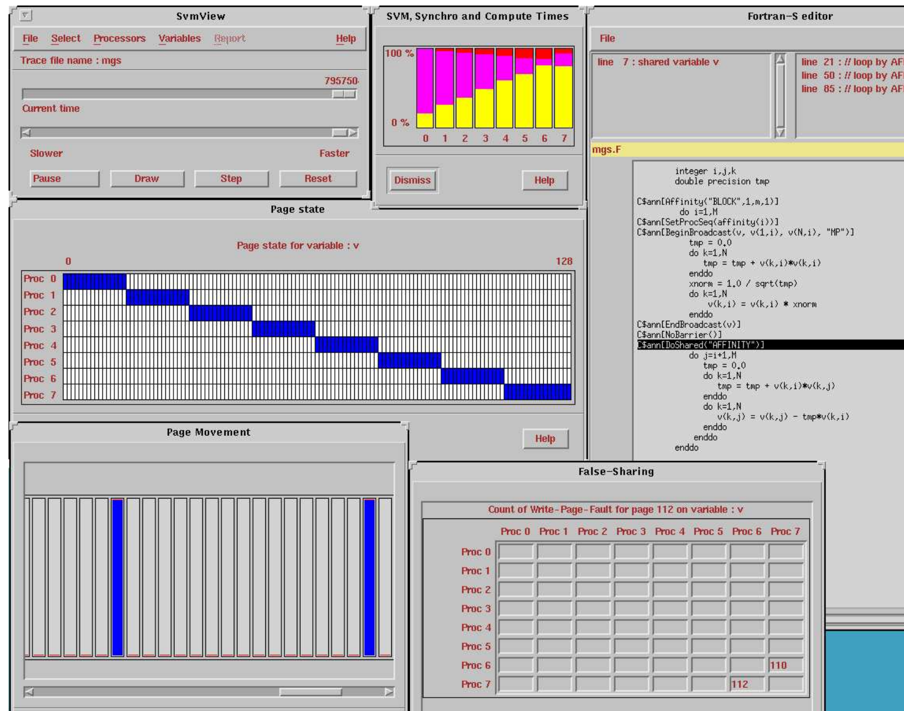


Figure 15: Reducing the impact of false-sharing (MGS5).

An affinity scheduling using a block loop distribution strategy assigned the first block of vectors to processor 0. When all these vectors have been orthonormalized (*i.e.* $i > block_size$), processor 0 has no more computation to do and this creates a load imbalance. This phenomenon also occurs for the other processors.

Consequently, the approach consisting in reducing the false-sharing effect, by using another scheduling strategy, does not give good performance. It has to be avoided using either a vector padding technique when using the ASVM DSM or a weaker cache coherence protocol as provided by the MYOAN DSM (C$ann[WeakCoherency(var)] directive). Performance study related to the different versions of the MGS algorithm can be found in [23].

7 Related work

There is few works dealing with the design of performance tuning tools for DSM systems. This is probably due to the lack of monitoring support provided by existing DSM systems. Another similar project to our own is currently in progress at KFA in Juelich. They designed a programming environment based on the ASVM DSM system [27] designed at Intel ESDC

in Munich. One component of this environment is the SVM-Fortran language which has some similarities with Fortran-S. A source-code-based optimizer and a locality analyzer tool, called OPAL [13], have been designed. Trace files can be visualized using the ParVIS tool. Comparing to our approach, ParVIS and OPAL are not yet integrated as a single tool. Consequently, programmers cannot analyze performance directly linked to its SVM-Fortran source code. Moreover, ParVIS was mainly designed to analyze message-passing parallel programs.

8 Conclusion and future orientation

SVMview has been developed from scratch and is targeted to the analysis of Fortran-S codes. It focuses mainly on data locality management to provide useful information to the programmer and allows to optimize their parallel codes. When programming with DSM architecture, the aim of most of the optimizations is to decrease page movements by exploiting the underlying locality. SVMview is also a testbed to carry out experiments on the interest of graphical displays in order to observe the behavior of parallel codes. Most of the displays we used came from existing tools such as Pablo or Maple. Adding additional graphical displays is very easy, they could be either part of SVMview or external.

However, SVMview has a limited scope in its current version: it focuses mainly on displaying page movements and cache contents. We are considering several extensions such as a control flow analysis of the parallel execution and a reflexive language able to describe what patterns have to be found in the trace files. The latter extension could extend the adaptability of the tool to be able to locate specific data accesses not restricted to false-sharing or producer-consumer schemes.

Our approach is mostly targeted to software DSM systems since we can add easily a monitoring support to catch specific events. Moreover, the page size restrains the number of events to be generated. However, in order to improve the performance of DSM, the current evolution is to implement a global address space by means of specialized hardware (KSR [7] or FLASH [10]) or using a standard interface such as the IEEE Scalable Coherent Interface (SCI) [25]. Most of the hardware DSM provides counters instead of events which cannot provide the same level of information than events. Consequently, a more aggressive program instrumentation is thus needed. Meanwhile, such instrumentation does not have to distort the behavior of the data access. It has also to cope with side-effects due to the operating system which can modify the counters value. Instrumentation techniques and performance tuning tools have to take into account these problems in order to provide an accurate analysis of the execution.

Acknowledgment

This work is supported by Intel SSD under contract no. 1 93 C 214 31318 01 2 and Esprit BRA APPARC. We thank S. Zeisset and M. Mairandres, from ESDC, for providing the ASVM DSM.

References

[1] F. André, J.-L. Pazat, and H. Thomas. PANDORE: A system to manage data distribution. In *Int. Conf. on Supercomputing*, pages 380–388, June 1990.

[2] R. A. Aydt. The Pablo Self-Defining Data Format. Technical report, University of Illinois, 1994.

[3] D. Badouel, K. Bouatouch, and T. Priol. Ray tracing on distributed memory parallel computers: Strategies for distributing computation and data. *IEEE Computer Graphics and Application*, 14(4):69–77, July 1994.

[4] S. Benkner, B. Chapman, and H. Zima. Vienna fortran 90. In *Scalable High Performance Computing Conference*, pages 51–59. IEEE Computer Society Press, April 1992.

[5] R. Berrendorf, M. Gerndt, W. Nagel, and J. Prümmer. SVM-Fortran. Internal Report KFA-ZAM-IB-9322, Central Institute for Applied Mathematics, Research Centre Jülich, 1993.

[6] F. Bodin, L. Kervella, and T. Priol. Fortran-S: A fortran interface for shared virtual memory architectures. In *Supercomputing'93*. IEEE Computer Society Press, November 1993.

[7] H. Burkhardt, S. Frank, B. Knobe, and J. Rothnie. Overview of the KSR1 computer system. Technical Report KSR-TR-9202001, Kendal Square Research, Boston, 1992.

[8] G. Cabillic, T. Priol, and I. Puaut. MYOAN: an implementation of the KOAN shared virtuel memory on the Intel Paragon. Technical Report 812, IRISA, March 1994.

[9] J.B. Carter, D. Khandekar, and L. Kamb. Distributed Shared Memory: Where we are and where we should be headed. In *Proc. of HOTOS'95*, 1995.

[10] M. Heinrich et al. The performance impact of flexibility in the stanford FLASH multiprocessor. In *6th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1994.

[11] HPF Forum. High performance fortran language specification. May 1993.

[12] M. Gerndt and R. Berrendorf. Parallelizing applications with SVM-Fortran. In *High-Performance Computing and Networking, Lecture Notes in Computer Science*, volume 919. Springer-Verlag, May 1995.

[13] M. Gerndt, A. Krumme, and S. Özmen. Performance analysis for SVM-Fortran with OPAL. Technical Report KFA-ZAM-IB-9519, KFA-ZAM, August 1995.

[14] E. D. Granston, T. Montaut, and F. Bodin. Loop transformations to prevent false sharing. *International Journal of Parallel Programming*, vol. 23(numéro 4):263–301, August 1995.

[15] R. G. Hackenberg. MaX: Investigating shared virtual memory. In *High-Performance Computing and Networking, Lecture Notes in Computer Science.* Springer-Verlag, April 1994.

[16] M. Heath and J. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, 8(5):29–39, August 1991.

[17] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C.W. Tsenh. An overview of the Fortran-D programming system. *Communication of the ACM*, 35(8), August 1992.

[18] A.H. Karp and V. Sarkar. Data merging for shared-memory multiprocessor. In *Proc. Hawaii International Conference on System Sciences*, pages 244–256, January 1993.

[19] Z. Lahjomri and T. Priol. KOAN: a shared virtual memory for the iPSC/2 hypercube. In *CONPAR/VAPP92*, September 1992.

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

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

[22] R. Mirchandaney, S. Hiranandani, and A. Sethi. Improving the performance of DSM systems via compiler involvement. In *Supercomputing'94*, pages 763–772. IEEE Computer Society Press, November 1994.

[23] T. Priol and Z. Lahjomri. Experiments with shared virtual memory on a iPSC/2 hypercube. In *International Conference on Parallel Processing*, pages 145–148, August 1992.

[24] D.A. Reed, R.A. Aydt, R.J. Noe, P.C. Roth, K.A. Shields, B.W. Schwartz, and L.F. Tavera. Scalable performance analysis: The PABLO performance analysis environment. In *Scalable Parallel Libraries Conference*, pages 104–113. IEEE Computer Society, 1993.

[25] ANSI/IEEE Standard. Scalable Coherent Interface (SCI). August 1993.

[26] J. Yan, S. Sarukkai, and P. Mehra. Performance measurement visualization and modeling of parallel and distributed programs using the AIMS toolkit. *Software, Practice and Experience*, 25(5):429–461, April 1995.

[27] S. Zeisset. Evaluation and enhancement of the Paragon multiprocessor's shared virtual memory system. Master's thesis, Munich Technical University, November 1993.

Appendix

A myoan trace format

```
SDDFA

/*
 *  Trace format for MYOAN Events
 *
 */;;

#1:
"MYOAN-read-page-fault" {
      int    "Node id";
      int    "Shared Region id";
      int    "Start.clock - microseconds";
      int    "End.clock - microseconds";
      int    "Faulting address";
      int    "Faulting page number";
      int    "Instruction address";
      int    "Node id that sends the page";
};;

#2:
"MYOAN-write-page-fault" {
      int    "Node id";
      int    "Shared Region id";
      int    "Start.clock - microseconds";
      int    "End.clock - microseconds";
      int    "Faulting address";
      int    "Faulting page number";
      int    "Instruction address";
      int    "Node id that sends the page";
};;

#3:
"MYOAN-invalidation" {
      int    "Node id";
      int    "Shared Region id";
      int    "Clock - microseconds";
      int    "Faulting address";
      int    "Faulting page number";
```

```
      int    "Node id that sends the invalidation";
};;

#4:
"MYOAN-change-perm" {
      int    "Node id";
      int    "Shared Region id";
      int    "Clock - microseconds";
      int    "Faulting address";
      int    "Faulting page number";
      int    "Node id that sends the change permission";
};;

/*
 * In 'Page status', for each page number, a value is provided
 * if the value is 0 : invalid state (not present in the local memory)
 *                 1 : read-only and not owned by the processor
 *                 2 : read-only and owned by the processor
 *                 3 : read-write and owned by the processor
 */;;

#6:
"MYOAN-page-dist" {
      int    "Node id";
      int    "Shared Region id";
      int    "Number of pages";
      int    "Page Status"[];
};;

#7:
"MYOAN-statistics" {
      int    "Node id";
      int    "Read fault";
      double "Min. time for a RF";
      double "Max. time for a RF";
      double "Med. time for a RF";
      double "Tot. time for a RF";
      int    "LRU read";
      int    "Write fault";
      double "Min. time for a WF";
      double "Max. time for a WF";
      double "Med. time for a WF";
```

```
      double "Tot. time for a WF";
      int    "LRU write";
      int    "Number of invalidation";
      double "MYOAN time";
};;

#8:
"MYOAN-page-fault-summary" {
      int    "Node id";
      int    "Shared region id";
      int    "Read fault";
      double "Min. time for a RF";
      double "Max. time for a RF";
      double "Med. time for a RF";
      double "Tot. time for a RF";
      int    "LRU read";
      int    "Write fault";
      double "Min. time for a WF";
      double "Max. time for a WF";
      double "Med. time for a WF";
      double "Tot. time for a WF";
      int    "LRU write";
      int    "Number of invalidation";
      double "MYOAN time";
};;
```

B Fortran-S trace format

```
SDDFA
/*
 *  Trace format for Fortran-S events
 *
 */;;

#101:
"RT-wait-barrier-start" {
      int    "Node id";
      int "Clock - microseconds";
      int    "FS line number";
      char   "FS file name"[];
};;
```

```
#102:
"RT-wait-barrier-stop" {
      int    "Node id";
      int "Clock - microseconds";
      int    "FS line number";
      char   "FS file name"[];
};;

#103:
"RT-wait-lock-start" {
      int    "Node id";
      int "Clock - microseconds";
      int    "Locking address";
};;

#104:
"RT-wait-lock-stop" {
      int    "Node id";
      int "Clock - microseconds";
      int    "Locking address";
};;

#105:
"RT-wait-crit-sec-start" {
      int    "Node id";
      int    "FS line number";
      char   "FS file name"[];
      int "Clock - microseconds";
};;

#106:
"RT-wait-crit-sec-stop" {
      int    "Node id";
      int    "FS line number";
      char   "FS file name"[];
      int "Clock - microseconds";
};;

/*
 * FS line number is the Fortran-S line number where
 * there is a C$ann[DoShared()]
 *
```

```
 */;;

#200:
"FS-parallel-loop-start" {
      int    "Node id";
      int "Clock - microseconds";
      int    "FS line number";
      char   "FS file name"[];
      int    "start value";
      int    "end value";
      int    "increment value";
      int "Startup time - microseconds";
};;

/*
 * FS line number if the Fortran-S line number where
 * there is a 'enddo' associated to a parallel loop
 *
 */;;

#201:
"FS-parallel-loop-stop" {
      int    "Node id";
      int "Clock - microseconds";
      int    "FS line number";
      char   "FS file name"[];
};;
#202:
"FS-shared-region-spec" {
      char   "Variable name"[];
      char   "Variable type"[];
      int    "Shared Region id";
      int    "Starting address";
      int    "Number of dimension";
      int    "Min"[];
      int    "Max"[];
};;

/*
 * FS line number is the Fortran-S line number of
 * an instruction that starts a sequential section
 *
```

```
 */;;

#203:
"FS-sequential-sec-start" {
      int    "Node id";
      int "Clock - microseconds";
      int    "FS line number";
      char   "FS file name"[];
};;

/*
 * FS line number if the Fortran-S line number of
 * an instruction that ends a sequential section
 *
 */;;
#204:
"FS-sequential-sec-stop" {
      int    "Node id";
      int "Clock - microseconds";
      int    "FS line number";
      char   "FS file name"[];
};;

/*
 * 'Processor number' is the processor number
 * in charge of updating shared variables within
 * a sequential code section
 *
 */;;

#205:
"FS-set-proc-seq" {
      int    "Node id";
      int "Clock - microseconds";
      int    "Processor number";
};;

/*
 * 'Number of processors' used during the execution
 *
 */;;
#206:
```

```
"FS-RT-stat" {
      int    "Number of processors";
};;

/*
 *  Value of the index of a parallel loop
 *
 *
 */;;

#207:
"FS-RT-index-value" {
      int    "Node id";
      int "Clock - microseconds";
      int    "FS line number";
      char   "FS file name"[];
      int    "Index value";
};;

#208:
"FS-RT-Coherence" {
      int    "Node id";
      int "Clock - microseconds";
      int    "FS line number";
      char   "FS file name"[];
      int    "Start address";
      int    "End address";
      int    "Merging time";
};;
```

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

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