INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Shared Virtual Memory and Message Passing Programming on a Finite Element Application

Rudolf Berrendorf and Michael Gerndt, Zakaria Lahjomri and Thierry Priol

N° 2355

Mars 1995

PROGRAMME 1

ISSN 0249-6399

Shared Virtual Memory and Message Passing Programming on a Finite Element Application *

Rudolf Berrendorf and Michael Gerndt**, Zakaria Lahjomri and Thierry Priol ***

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

Rapport de recherche n°2355 — Mars 1995 — 14 pages

Abstract: This paper describes the methods used and experiences made with implementing a finite element application on three different parallel computers with either message passing or shared virtual memory as the programming model. Designing a parallel finite element application using message-passing requires to find a data domain decomposition to map data into the local memory of the processors. Since data accesses may be very irregular, communication patterns are unknown prior to the parallel execution and thus makes the parallelization a difficult task. We argue that the use of a shared virtual memory greatly simplifies the parallelization step. It is shown experimentally on an hypercube iPSC/2 that the use of the KOAN/Fortran-S programming environment based on a shared virtual memory allows to port quickly and easily a sequential application without a significant degradation in performance compared to the message passing version. Results for recent parallel architectures such as the Paragon XP/S for message-passing and the KSR1 for shared virtual memory are presented, too.

(Résumé : tsvp)

*This work is supported by the Esprit BRA APPARC
**Zentralinstitut für Angewandte Mathematik, Forschungszentrum Jülich (KFA), D-52425 Jülich - Germany
***IRISA-INRIA

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

Comparaison des techniques de mémoire virtuelle partagée et d'échange de messages à l'aide d'une application par élément finie

Résumé : Ce papier présente les résultats d'une mise en oeuvre d'une application par élément finie sur trois architectures parallèles en utilisant soit l'échange de messages soit une mémoire virtuelle partagée. La conception d'une application par élément finie sur une architecture parallèle à mémoire distribuée nécessite une décomposition de domaine afin de placer les données dans les mémoires locales des processeurs. Les accès aux données étant irréguliers, la parallélisation est une tâche complexe. Nous montrons que l'utilisation d'une mémoire virtuelle partagée simplifie grandement cette tâche. Nous montrons, à l'aide de résultats expérimentaux sur un hypercube iPSC/2, que l'utilisation de l'environnement de programmation KOAN/Fortran-S permet le portage rapide d'une telle application sans constater une perte significative. Nous présentons également des résultats sur des machines plus récentes comme le Paragon XP/S et la KSR-1.

1 Introduction

Parallelizing applications with irregular data access, such as those using finite element techniques, on distributed memory parallel computers is a complex task due the distributed nature of the memory. Porting such applications on distributed memory parallel computers requires to handle arbitrary data distributions as the data accesses are unknown at compile time. The PARTI subroutine package developed at NASA/ICASE by J. Saltz et al. [5] has been designed for that purpose. It allows the computation of processor-local indices, the analysis of communication patterns and the communication of non-local array elements. By using PARTI, few modifications to the sequential application are necessary to get a parallel version. However, the user is still responsible to select those arrays to be distributed, to specify the distributions, and to carefully analyze and transform references to distributed arrays. The specification of the distributions as well as the code analysis requires deeper knowledge of the application and is a time consuming and error-prone task.

To avoid this tedious task, one can use a shared virtual memory (SVM) concept so as to program distributed memory parallel computers like conventional shared memory parallel architectures. A SVM hides the physical local memories and provides to the user a virtual address space made of pages that move on demand among processors. Each local memory acts as large cache. This concept can be implemented within the operating system such as the KOAN SVM [6] or by specialized hardware devices as done in the KSR1 of Kendall Square Research. However, using an SVM will add additional overhead to the parallel execution caused by several factors like a distribution overhead, cache coherency, etc. The aim of this paper is to provide a comparison between the two programming models with respect to programming aspects and performance. A comparing experiment is carried out using the same machine: an hypercube iPSC/2. Since this machine does not represent the state of the art in the design of parallel architectures, we present additionally the results for two recent parallel architectures: the Intel Paragon XP/S and the KSR1.

This paper is organized as follows. Section 2 describes the ParFEM application we used in the test. Section 3 addresses the parallelization of ParFEM using a message-passing programming model whereas section 4 and 5 deals with the shared virtual memory programming model. Section 6 discusses several issues for the two programming models. We conclude in section 7.

2 ParFEM: A Finite Element Application

The ParFEM application has been developed by Harry Vereecken et al. (Institute for Petrol and Organic Geochemistry) [9] at KFA. This application models transport and chemical processes in heterogeneous 3D porous media. Accounting for the heterogeneity of the porous medium results in grid size of more than 10^6 nodal points. In combination with the strong nonlinearity of the partial differential equations such problems can only be handled on high performance computer systems in combination with appropriate numerical solution techniques for the linearized set of equations. These equations are obtained by applying

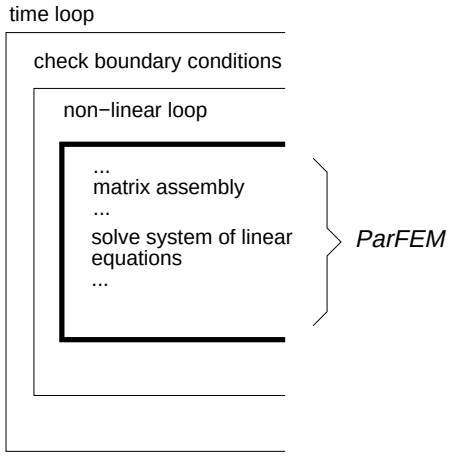

Figure 1: Overall program structure of ParFEM.

Galerkin's finite element method to the water and solute transport equation. The program is based on a code originally developed by Yeh [10]. Fig. 1 shows the overall program structure. The outer loop models the steps in time, the boundary loop varies certain boundary conditions, and with the non-linear loop the system of non-linear equations is reduced to a system of linear equations. Inside this non-linear loop are the compute-intensive parts of the application, the assembling of the finite element matrix and the solution of the linear equations. The assemble matrix, which is set up in the assemble part and passed to the solver, is a sparse matrix with 18 (2D-problem) or 27 (3D-problem) non-zero elements per row.

From the whole application we have picked the most time consuming part of the inner loop, the assemble of the finite element matrix and the solution of a set of equations with a conjugate gradient method. For our experiments we have used two data sets, a small data set with 1254 nodal points (900 elements; small 3D problem) and a larger data set with 17368 nodal points (8325 elements; large 2D problem). The assemble part mainly is a loop over all elements. The loop iterations can be run in parallel, but update operations on shared variables have to be done atomically. The conjugate gradient method is an iterative algorithm for finding the solution for a set of equations; the algorithm iterates until a convergence criterion is reached.

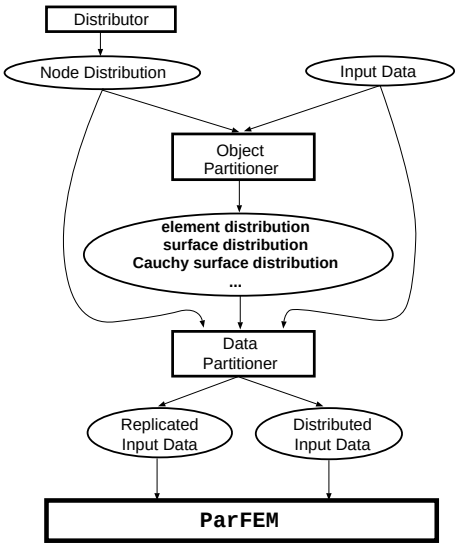

Figure 2: Data distribution tools.

3 Message-Passing

3.1 Data Domain Decomposition

The most performance critical decision is the selection of the data decomposition strategy. The parallel code handles arbitrary distributions correctly but the communication overhead can be quite different. Therefore a tool was developed, the *distributor*, which computes from a specification of the problem domain, a specification of the global node numbering scheme, and a user specified decomposition strategy the distribution of the nodes. From the node distribution several distributions of other objects are derived automatically by a tool called *object partitioner*. For example, the partitioner computes a distribution of the grid elements based on the majority rule: an element is assigned to that processor which owns most of its nodes. If there is no unique processor with this property an arbitrary processor is selected. This step requires to understand the indexing scheme in the application. Fig. 2 shows the global organization of the data distribution tools.

According to the distributions, the input data is rearranged and copied to a parallel file system such that each processor can read the information for its local array elements efficiently. The developed tool, the *data partitioner*, is executed on one node of the parallel system. It reads the different distributions and the sequential data, and generates one file containing the data replicated in all processors and one file for every processor in the parallel file system containing the information of distributed arrays.

3.2 PARTI Subroutines

The implementation is based on a subroutine package developed at NASA/ICASE by Joel Saltz et al. called PARTI [5]. It supports arbitrary distributions of arrays, computation of processor-local indices, analysis of communication patterns, and communication of non-local array elements.

Distributions are specified in each processor via a list of the global array indices assigned to the processor. Based on the distributions, communication patterns and local indices are computed. Pre-computed communication patterns are then used in communication operations to perform the actual exchange of array element values. Since the problem topology is fixed, the expensive analysis of communication patterns need to be done only once.

PARTI supports gather, scatter, and scatter add operations to fetch, distribute and combine information for non-local elements. Besides these operations on one-dimensional arrays also operations on two-dimensional arrays were needed and partly developed in cooperation with ICASE during the parallelization of this application.

3.3 Matrix Assembly

The assembling step is made up of three different phases. First, all information of non-local nodes of elements owned by a processor is gathered from the owners. Then the computation of the individual element matrices is performed. The components of the element matrices are then combined in each processor for local as well as non-local nodes of these elements. Afterwards, the partial information for non-local nodes computed by the owners of elements is combined in the owners of the nodes via a scatter add operation.

3.4 Conjugate Gradient

The linear equation system determined by the global matrix is solved with the Conjugate Gradient method. During each step of this iterative solver the values of array elements of the current solution have to be exchanged. In addition, the termination condition has to be computed by global reductions.

4 Shared Virtual Memory on KOAN/Fortran-S

4.1 Overview of KOAN/Fortran-S

The KOAN/Fortran-S programming environment allows the user to program a distributed memory parallel architecture without explicit message passing. This programming environment is constituted of two components: a Shared Virtual Memory (KOAN) and a Fortran code generator (Fortran-S). This programming environment has been ported on the Intel hypercube iPSC/2 but work is under progress to port this programming environment on the new Intel Paragon XP/S.

```
C  gather information for non-local node
   call dfgather(sched1,x(nnp+1),x(1))
   ...
   call ifm2dgather(sched1,lrn(1,nnp+1),
  +                 lrn(1,1),1,jband)
C  perform computation for local elements only
   do m=1,nel
     ... local computation ...
     do iq=1,8
       ni = iem(iq)
       rld(ni) = rld(ni) + ...
       do jq=1,8
         ...
         rld(ni) = rld(ni) + ...
         i = ...
         cmatrx(i,ni) = cmatrx(i,ni) + ...
       enddo
     enddo
   enddo
C  combine partial sums of matrix elements
   call dfmdscatter_add(sched1,cmatrx(nnp+1,1),
  +                     cmatrx(1,1),maxnp,jband)
   call dfscatter_add(sched1, rld(nnp+1), rld(1))
```

```
C$ann[DoShared("BLOCK")]
C$ann[VGlobal(DSUM, tmprld, 1, nnp)]
   do m=1,nel
   ... local computation ...
     do iq=1,8
       ni = iem(iq)
       tmprld(ni) = tmprld(ni) + ...
       do jq=1,8
         ...
         tmprld(ni) = tmprld(ni) + ...
         i = ...
C$ann[AtomicUpdate()]
         cmatrx(i,ni) = cmatrx(i,ni) + ...
       enddo
     enddo
   enddo
C$ann[DoShared("BLOCK")]
   do i=1,nnp
     rld(i) = tmprld(i)
   enddo
```

Figure 3: Matrix assembly: message-passing (left) and KOAN/Fortran-S (right).

```
C  Gather information of non local node
   call dfgather(sched2,vec(nnp+1),vec(1))
   do j=1, nnp
     temp=0.
     do i=1, eintrz
       temp=temp+cmatrx(i,j)*vec(gnojcn(i,j))
     enddo
     axvec(j)=temp
   enddo
```

```
C$ann[DoShared("BLOCK")]
   do j=1, nnp
     temp=0.
     do i=1, eintrz
       temp=temp+cmatrx(i,j)*vec(gnojcn(i,j))
     enddo
     axvec(j)=temp
   enddo
```

Figure 4: Matrix-vector multiply: message passing (left) and KOAN/Fortran-S (right).

KOAN is a Shared Virtual Memory (SVM) embedded in the operating system of the iPSC/2 [6]. It provides to the user an abstraction from an underlying memory architecture [8]. It provides a virtual address space that is shared by a number of processes running on different processors of a distributed memory parallel computer (DMPC). In order to distribute the virtual address space, the SVM is partitioned into pages which are spread among local processor memories according to a mapping function. Each local memory acts as a large software cache for storing pages. Since the size of the physical memory on a processor is much less than the size of the SVM, the part of the local memory, which acts as a cache, is managed according to a LRU (Least Recently Used) policy. A memory management unit (MMU) is needed to provide the user with a linear address by translating virtual addresses to physical ones. KOAN provides several functionalities such as different cache coherence protocols and synchronization mechanisms

Fortran-S is a code generator targeted for shared virtual memory parallel architectures such as the iPSC/2 running KOAN or the KSR1. It respects the Fortran-77 standard since it is widely used in the scientific community. Therefore no extension to the language syntax has been made. A set of annotations provides the user with a simple programming model based on shared array variables and parallel loops. One of the main features of Fortran-S is its SPMD (Single Program Multiple Data) execution model [4] that minimizes the overhead due to the management of parallel processes. The Fortran-S code generator creates a process for each processor for the entire duration of the computation. There is no dynamic creation of processes during the execution. A description of Fortran-S is given in [3]. A parallelizer, called PARKA, can generate Fortran-S code from a sequential fortran-77 code.

4.2 Matrix Assembly

This section outlines the parallelization of the matrix assembly using the KOAN/Fortran-S programming environment. A simplified version of the matrix assembly algorithm is shown in Fig. 3. The outer loop is used to scan each element of the mesh. This loop cannot be parallelized without adding adequate synchronization. Due to indirect access to shared variables *cmatrx* and *rld*, there are some potential data dependencies. Two techniques have been exploited to overcome these dependencies. First a temporary variable, that is not shared and thus replicated on every processor, allows to make the update to *rld* independently. By using a proper annotation (C$ann[Vglobal(DSUM,tmprld,1,nnp)]), a global sum operation is performed on each element of vector *tmprld* and each processor has the same value of each element of vector *tmprld*. Later the shared variable *rld* is updated by the private temporary sums. (see Fig. 3). The main advantage of this technique is to avoid *false-sharing* when updating a shared variable. *false-sharing* appears when several processors write into different addresses located in the same page. The page will move back and forth between the two processors and thus increasing the communication time. This optimization seems to be burdensome but could be carried out automatically by the Fortran-S code generator.

Similarly, this technique could be used with the *cmatrx* variable however this would be very memory consuming since the variable has to be replicated on every processor. There is another solution based on an explicit synchronization scheme to avoid several processors

to update the same element. For this purpose, Fortran-S provides two synchronization mechanisms: critical section and atomic update. Critical section is not well suited for synchronizing the update since it is implemented with a distributed algorithm using message passing [6]. It is mainly targeted for synchronizing large grain computations which is not our case. Atomic update is an efficient synchronizing mechanism based on the locking of pages into the cache of each processor. A C$ann[AtomicUpdate()] annotation has to be added before the assignments to ensure that updates to *cmatrx* are done atomically: the processor in charge of updating *cmatrx* locks the page that contains the element that have to be updated. Requests to this page, coming from other processors, will not be processed until the page is unlocked. As the work in each iteration is nearly equal, we have used a block scheduling for the loop distribution.

4.3 Conjugate Gradient

The parallel version is very similar to the sequential one. Some parts have been modified in order to be able to use reduction operations such as global sum or global maximum. The algorithm iterates until a convergence criterion is reached. For each iteration step, a matrix-vector multiply subroutine (Fig. 4) is called. In the CG algorithm, there are mainly two kinds of loops that can be distributed: loops that update vectors and reduction loops. For these latter loops (five in the code), annotations such as C$ann[SGlobal(DSUM,var)] and C$ann[SGlobal(DMAX,var)] have been inserted into the code. In four cases, such optimizations can be carried out automatically by the PARKA parallelizer. The Fortran-S code generator will add extra code to call necessary message-based reduction functions at the end of the execution of the parallel loop. For performance reasons this is implemented with message-passing rather than using the SVM.

The parallelization of the matrix-vector multiply assigns a part of the result vector to each processor. Due to the same blocking distribution technique applied for every loop in the CG algorithm, the resulting vector does not need to be stored in a shared memory region. However, the input vector has to be stored in a shared variable since it is accessed indirectly. Fig. 4 shows the parallel version of the matrix vector multiply.

5 Shared Virtual Memory on the KSR

5.1 Overview of the KSR

The KSR1 of Kendall Square Research is a parallel machine with hardware-embedded SVM, called $ALLCACHE^{TM}$, implementing sequential consistency [7]. The unit of coherence is a subpage of size 128 bytes. Parallel constructs are available to the user on several levels. On the most basic level are POSIX pthreads. On the next higher level are PRESTO routines, which dynamically evaluate runtime decisions to improve performance. Available constructs to specify parallelism are parallel regions, parallel sections, tile families (parallel loops), and affinity regions for multiple loops. Tiling in Fortran can be done automatically with the

KAP preprocessor, semi-automatically with programmer hints to the preprocessor which inserts missing information to the best of its knowledge, or manually. While automatic parallelization is the easiest way to use, best performance is reached usually with manual parallelization. To control the assignment of work to pthreads and eventually to processors, the user can group a number of pthreads to *teams* such that subsequent parallelism is partitioned the same way. This guarantees that memory regions are accessed by the same processors thus reducing page conflicts. Affinity regions have the same aim for loop tiling over several loops. For a full description of the hardware and software details see [1].

5.2 Parallelizing for the KSR

As the general parallelization strategy for the KSR and KOAN/Fortran-S is similar we will describe only the relevant differences for the KSR compared to the work done for KOAN/Fortran-S.

The basic programming model for the KSR is the fork-join model in contrast to Fortran-S which uses the SPMD-model. To reduce the overhead on forking with parallel constructs we allocated at the start of the program a team of threads which is given as an additional argument to parallel constructs. With this technique the overhead for fork and join was reduced.

The original program version was written for Cray-like machines with little attention to data locality. While porting the application to one processor of the KSR it has shown up that it was necessary to optimize the program with respect to locality to utilize the processor cache. All optimizations applied would be important for other (sequential) machines with a memory hierarchy (e.g. DEC Alpha, IBM RS/6000, etc.), too. It is remarkable, that all applied cache optimizations have influenced the performance with respect to SVM in a positive way, and that after this work only few modifications were necessary to optimize for the specifics of SVM.

Synchronizing accesses to shared variables in the assemble part was done with page locks to ensure exclusive access to parts of a data structure while other processors still have the possibility to work on other parts. A first implementation with critical sections (as it would be done on a shared memory machine with a few processors only), i.e. exclusive access to a code section, has shown a significant bottleneck.

We used two alternatives to reduce false sharing on the KSR: introduction of private data which has to be done with care to keep the used memory small, and alignment of variables on page boundaries if the false sharing happens between different variables.

KSR offers several possibilities for parallelizing an application. The easiest to use approach are parallel loops marked with special comments. While the advantage of this model is its ease of use, the disadvantage is the relative high overhead for initiating a parallel loop; only if enough work is available inside a loop the startup costs and the costs of the barrier synchronization at the end of a loop can be compensated. This is especially a problem for small loops, e.g. to initialize data structures, where the high overhead of a parallel loop has to be weighted against the disadvantage of a concentration of all accessed subpages on one

node if the loop is executed sequentially by one processor. Parallelizing the program with parallel loops resulted only in small modifications to the sequential code.

The other approach is to run a large program section in parallel and to manage loop iteration splitting, synchronization, reduction operations, etc. by your own using low-level primitives of the operating or run-time system, i.e. using an SPMD-model for that part of the program. The advantage is that the parallel overhead can be kept smaller, especially if the program has several small parallel sections. The disadvantage with this approach is, that the code has to be rewritten (sometimes substantially) and temporary data structures for each thread have to be managed. With ParFEM the performance gain over the parallel loops approach was relatively small (within 10 percent) such that this type of parallelization was be useful only on restricted sections of code rather than a general parallelization strategy.

6 Discussion

Table 1 summarizes the performance results we got from the experiments described in the previous chapters. More results can be found in [2]. In some columns for the large data set, results are missing due to insufficient memory for all the data. *KOAN/Fortran-S* stands for an iPSC/2 with KOAN/Fortran-S, *MP+PARTI* for an iPSC/2 with message-passing, *Paragon* for message-passing implementation on the Paragon XP/S, and *KSR* for the KSR1. *small* gives the results for the small data set (1254 nodes) and *large* gives the results for the large data set (17368 nodes).

Comparing the SVM version with the message passing version with respect to performance can be done only for the two versions running on the same hardware (iPSC/2). The SVM version for more than 1 processor is within a factor of 1.1 (small number of processors) to 1.4 (32 processors) compared to the message-passing version. One reason for the loss of performance on a large number of processors is the large page size (4 K) resulting in data access conflicts. The KSR scales well on the large data set, one reason is the small (sub-) page size of 128 bytes. The KSR results seems to be similar to the ones obtained with the Paragon XP/S except for the small data set. This can be explained by the greater data cache size of the KSR processor (256 KB) comparing to the one in the i860 micro-processor (16 KB). With the large data set, the cache size has few impact. The performance results mentioned for the message passing version will show a much better efficiency for a parallelized version of the entire application. The overhead for handling the distributions and for computing the communication patterns is neglectable for more realistic runs of the entire application since the setup time is constant. This setup phase takes almost 45 % of the total execution time on 32 nodes and dominates the total time for the small data set. For the faster machines, Paragon and KSR1, the small data set had not enough work to compensate the parallel overhead.

All program versions of the application are very similar to the sequential program. The main modifications we made for the SVM-versions, KOAN/Fortran-S and KSR, dealt with

[1] Due to restrictions only 27 processors were used really.

Proc.	KOAN/Fortran-S		MP+PARTI		Paragon		KSR	
	small	large	small	large	small	large	small	large
1	86.9		92.4		6.5	30.8	1.75	38.6
2	59.1		51.0		4.8	18.2	0.94	20.2
4	30.9		26.6		2.8	9.8	0.59	10.3
8	18.3		16.7		3.9	5.5	0.43	5.4
16	14.4	134.5	11.8		3.7	3.8	0.41	3.1
32	12.6	73.5	8.9	53.1	3.3	3.3	0.56[1]	2.3[1]

Table 1: Performance results (in seconds).

the exploitation of spatial locality, avoidance of false sharing (mainly by array privatisation), and handling of reductions. Although Fortran-S and the KSR use different programming models (SPMD vs. fork-join) the modifications we made were similar. Both versions could be executed without modifications on any number of processors, and the code still runs on sequential machines. With the message passing version, three pre-processing steps, i.e. domain decomposition specification, rearranging input data, and re-compilation, have to be performed prior to the parallel execution on a different processor configuration. Although only few changes were made to the sequential code during the parallelization, a deep understanding of the application was needed. To successfully parallelize that application also the subroutine evaluating the boundary conditions had to be transformed since distributed arrays are accessed. Although this subroutine only requires 0.5 % of the sequential execution time it was the most difficult to handle. Most of the object distributions are related to arrays used only in this subroutine. Several different indexing schemes had to be analyzed and almost all references to arrays in this subroutine implied a specific analysis and code adaptation. In the SVM version this subroutine has not to be dealt with since it can be executed sequentially.

One big advantage of the SVM programming model was the ability to do incremental parallelization i.e. starting with the sequential version we began to parallelize some kernels only, running the rest of the program sequentially. With the message passing version all debugging had to be performed on the fully restructured code thus making it very difficult to detect indexing failures and communication failures. These result only in numerical differences between the sequential and the parallel solution.

7 Conclusion

We have described our experience and results with porting an application with irregular data access to different parallel systems with either shared virtual memory or message passing. To conclude the experiments, SVM can be efficient if the size of the problem is large enough. Beside finding parallel portions in a program, the major task on machines with SVM is to

avoid page access conflicts. Privatizing data structures has the obstacle that the demand of memory on each node grows considerably. Another possibility is the use of a weaker cache coherence protocol. Two major benefits found in parallelizing the application for an SVM system was the ability to do incremental parallelization starting from a sequential program version, and the parallelization by local transformations. The message-passing version needed a deep understanding of the whole application and it was not possible to parallelize it incrementally. On the other side, in a direct comparison on the iPSC/2, we got better performance results with the message passing version. Our work on parallelizing the code has shown that utilities to accurately monitor system activities and to report the results to the user in a reasonable way is essential for doing a good job on program optimization on this type of machines.

8 Acknowledgments

We would like to thank Harry Vereecken of the Institute for Petrol and Organic Geochemistry at KFA who gave us his application program for our parallelization tests. Achim Basermann (Central Institute for Applied Mathematics, KFA) has contributed the sequential conjugate gradient solver and helped us in many discussions. The Centre for Novel Computing (CNC) at the Manchester University gave us access to their Kendall Square Machine, we thank them very much, especially Mark Bull and Graham Riley. We would like to thank Prof. A. Bode and R. Hackenberg at the TU München for providing access to their iPSC/2 system.

References

[1] *Technical Summary.* Kendall Square Research, Waltham, Massachusetts, edition, 1992.

[2] Rudolf Berrendorf, Michael Gerndt, Zakaria Lahjomri, Thierry Priol, and Philippe d'Anfray. *Evaluation of numerical applications running with shared virtual memory.* Internal Report KFA-ZAM-IB-9315, KFA Research Centre Juelich, 1993.

[3] F. Bodin, L. Kervella, and T. Priol. Fortran-s: a fortran interface for shared virtual memory architectures. In *Supercomputing'93*, pages 274–283, IEEE, November 1993.

[4] F. Darema-Rodgers, V.A. Norton, and G.F. Pfister. *Using A Single-Program-Multiple-Data Computational Model for Parallel Execution of Scientific Applications.* Technical Report RC11552, IBM T.J Watson Research Center, November 1985.

[5] R. Das and J. Saltz. *A Manual for Parti Runtime Primitives - revision 2.* Internal Research Report, ICASE, 1992.

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

[7] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(6):313–348, September 1979.

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

[9] H. Vereecken, G. Lindenmayr, A. Kuhr, D. H. Welte, and A. Basermann. *Numerical Modelling of Field Scale Transport in Heterogeneous Variably Saturated Porous Media.* Internal Report KFA/ICG-4 No. 500393, Forschungszentrum Jülich, 1993.

[10] G. T. Yeh. *3DFEMWATER, a Three Dimensional Finite Element Model of Water Flow Through Saturated-Unsaturated Media.* ORNL-6386, Oak Ridge National Laboratory, 1987.

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