CORE
Metadata, citation and similar papers at core.ac.uk
Provided by Juelich Shared Electronic Resources

FORSCHUNGSZENTRUM JÜLICH GmbH
Zentralinstitut für Angewandte Mathematik
D-52425 Jülich, Tel. (02461) 61-6402

Interner Bericht

Programming Shared Virtual Memory on the Intel Paragon^{TM} Supercomputer

R. Berrendorf, M. Gerndt
*M. Mairandres**

KFA-ZAM-IB-9509

April 1995
(Stand 13.04.95)

(*) European Supercomputer Development Center
Intel SSD
85622 Feldkirchen b. M., Germany

Programming Shared Virtual Memory on the Intel Paragon^{TM} Supercomputer

R. Berrendorf, M. Gerndt
Central Institute for Applied Mathematics
Research Centre Jülich (KFA)
52425 Jülich, Germany
{r.berrendorf,m.gerndt}@kfa-juelich.de

M. Mairandres
European Supercomputer Development Center
Intel SSD
85622 Feldkirchen b. M., Germany
martin@esdc.intel.com

Abstract

Programming distributed memory systems forces the user to handle the problem of data locality. With message passing the user has not only to map the application to the different processors in a way that optimizes data locality but also has to explicitly program access to remote data. Shared virtual memory (SVM) systems free him from the second task; nevertheless, he is still responsible to optimize data locality by selecting a well-suited work distribution. We describe a programming environment that is based on the Advanced Shared Virtual Memory system, a SVM implementation for the Paragon^{TM} Supercomputer, and on SVM-Fortran, a shared memory parallel programming language with language constructs for work distribution. Programming tools integrate program text and dynamic performance data to help the user in optimizing data locality.

Keywords: distributed memory multiprocessors, shared virtual memory, parallel programming tools, programming environments, parallel applications

1 Introduction

Programming distributed memory parallel computers with message passing is often considered to be a difficult task. Several approaches are being pursued such as HPF, shared virtual memory (SVM), or Linda to facilitate this task. Common to these approaches is the emulation of a global address space by software, either by the compiler, the operating system, or the language runtime system.

Shared virtual memory systems provide a global address space on the basis of the virtual addressing mechanisms in the operating system. Several research prototypes of SVM [1, 2] demonstrated the feasibility of that concept, but were not designed to support real-world scientific applications on state-of-the-art parallel systems.

The programming environment we are describing in this article is intended to support scientific programs utilizing SVM on the Intel Paragon^{TM} Supercomputer. It is based on the Advanced Shared Virtual Memory (ASVM) system, a prototype developed by the Intel European Supercomputer Development Center (ESDC). On top of ASVM, KFA is providing SVM-Fortran with a source-to-source compiler, a source code based optimizer and locality analyzer OPAL, and a trace visualization tool PARvis.

The key issues in providing an easy to use programming interface for SVM are: an efficient and scalable SVM implementation, fast barrier synchronization on processor sets, efficient monitoring support integrated into the SVM system, a language design oriented at the applications' needs, and analysis and optimization tools integrating program text and runtime performance information.

Section 2 presents the design concepts of ASVM and Section 3 introduces the language features of SVM-Fortran. In Section 4 we present the components of the programming environment and discuss their relationship. Early performance results of ASVM, of the code generated by the compiler and of an application are presented in Section 5. Subsequently, we discuss related work and future research issues.

2 ASVM on the Paragon^{TM} Supercomputer

The programming environment we are describing in this article is implemented on the Intel Paragon^{TM} Supercomputer, a distributed memory multicomputer that can accommodate more than thousand heterogeneous nodes connected in a two-dimensional rectangular mesh. Nodes are designed around Intel's i860 XP RISC processor. Multiprocessor (MP) nodes have three 75-MFLOPS Intel i860 XP processors — two to execute application code and a third processor for use as either a message coprocessor or as an application processor. General-purpose (GP) nodes have two i860 XP processors, one dedicated to applications and the other to messaging. An expansion port allows the addition of an I/O or networking interface. Nodes communicate by passing messages over an internal interconnect network providing high bandwidth and low latency.

The Paragon^{TM} Supercomputer's operating system [17, 18] is fully and transparently distributed across the system's nodes. A microkernel based on Mach 3.0 [19] resides on each node and implements core operating system functions. On top of the microkernel, an OSF/1 AD [20] server is running, providing full Unix semantics and a single system image. Apart from a single file name space and a single process space that spawn node boundaries, the single system image offers SVM functionality.

The SVM functionality is implemented in the Advanced Shared Virtual Memory system. ASVM is currently the prototype of a new SVM system for the Paragon^{TM} Supercomputer that substitutes the XMM layer [21] of the originally employed version of Mach. Our SVM system is integrated into the Mach microkernel and has interfaces to the local virtual memory system of every node on one side and to user or system level pagers on the other side. Similar to global shared memory, ASVM provides regions of virtual memory that can be shared among all nodes of a Paragon^{TM} Supercomputer. Actually, these memory regions are distributed between the nodes. In order to manage the SVM regions, ASVM uses page fault mechanisms similarly to standard virtual memory management.

ASVM supports strong coherency, i.e. if a processor reads from a location in a shared address range it always gets the value written in the last write operation to that location executed by any processor. This is achieved by keeping either only one copy of a page with write access or several copies of the page on different nodes having read-only access. When a read-only page is to be written on a node, all other copies of the page are invalidated first and then the node is granted write access to the page.

ASVM was developed with special focus on high performance and scalability to support real-world scientific applications. In order to accomplish this objective ASVM has some special features:

- **cache-based, distributed, multi-level management of SVM pages**
  A SVM system always has to know the location and status of all SVM pages. ASVM's strategy for management of SVM pages combines previously known methods and some new features into a cache-based, distributed, and multi-level page management protocol that was specifically designed for high scalability and efficiency. The basic idea is to keep only the recently used management information for different management strategies in a sort of cache and thus ensuring scalability by restricting the memory requirements of ASVM to a fixed amount of the node's memory, independent of the number of nodes and the size of the SVM regions. If one of the management strategies fails due to missing information in the cache, ASVM falls back to another strategy. The final management strategy is inefficient but always finds the required information about a page.

- **integration into the Mach microkernel**
  Most existing SVM systems on top of Mach are implemented on user level, i.e. external pagers. ASVM, however, is part of the Mach microkernel, thereby avoiding additional communication and task switches and improving performance considerably.

- **special SVM transport service**
  Messages for SVM management are sent via a special SVM transport service that is tuned to the size of SVM messages and directly uses the communication processor on every node of the Paragon^{TM} Supercomputer. The result is further improvement of performance.

- **paging to other nodes**
  If a node runs out of memory it first tries to send SVM pages to other nodes that are selected according to a certain strategy. Only if these nodes are also short on memory the pages are sent to disk. Sending pages to other nodes obviously is much faster than paging them to disk and avoids a possible bottleneck at the disk in case of heavy paging activity of the system.

For tuning of application programs as well as tuning of the ASVM system itself, ASVM provides extensive monitoring support. Since ASVM is integrated into the Mach microkernel all necessary monitoring information is available. Monitoring is supported on the levels of single applications and whole nodes. Especially on the application level, selective instrumentation (e.g. restriction to an arbitrary address region) and versatile counters can be used to keep the amount of monitoring data rather small. A more detailed description of monitoring support is given in Section 4.3.

3 SVM-Fortran

SVM-Fortran [4] is a shared memory parallel Fortran77 extension targeted mainly towards data parallel applications on shared virtual memory systems and distributed shared memory systems which provide hardware support for a global address space on top of physically distributed memory. It is based on HPF [5], Fortran-S [6], and KSR Fortran [7]. The main application area is broader than that of HPF and Vienna Fortran. SVM-Fortran supports coarse-grained functional parallelism where the parallel task itself can be data parallel.

The execution model of SVM-Fortran is an extension of the Single-Program-Multiple-Data (SPMD) model [8] which is well-known for its low-overhead parallel execution and is best-suited for hierarchical memory machines. Processors are allocated to the parallel application when it is started and are available until program termination.

SVM-Fortran supports nested parallelism. At program start, the entire computation forms a single task. A *task* is some computational work. Tasks can be dynamically decomposed into subtasks, e.g. via a parallel section or a parallel loop construct, i.e. each section and each loop iteration is an independent task. A task is either assigned to a single processor or to a set of processors which execute the task cooperatively. The set of processors executing a task is called the task's *active processor set (APS)*. The active processors execute the task either in *exclusive mode* or in *replicated mode*. In exclusive mode only one processor, the APS leader, performs the actual computation, whereas in replicated mode all processors execute the same code. The default execution mode is exclusive mode which facilitates incremental parallelization of applications.

SVM-Fortran provides the standard features of shared memory parallel Fortran languages, i.e. shared and private data, multi-dimensional parallel loops and parallel sections, classical synchronization operations as well as SVM-specific synchronization such as variable locking and atomic update.

SVM-Fortran provides specific features to determine the distribution of tasks onto processors. Similar to Fortran-S and KSR Fortran, loop annotations can be used to determine a static or dynamic work distribution scheme. Examples are *direct scheduling*, such as BLOCK and CYCLIC, as well as *dynamic scheduling*, e.g. self-scheduling.

Data locality is not a problem to be solved on the level of individual do-loops but is a global problem. Therefore, SVM-Fortran borrowed the concepts of processors arrangements and templates from HPF as tools to specify scheduling decisions globally via template distributions. Template distributions determine the work distribution for parallel loops in *predefined scheduling* and *semi-dynamic scheduling*. In predefined scheduling, loop iterations are assigned to processors according to the distribution of the appropriate template element. In semi-dynamic scheduling, templates are distributed according to dynamic scheduling decisions. Thus, the result of a dynamic scheduling decision can be applied to subsequent loops to exploit data locality.

Templates can be handled very flexible. They can be dynamically created, distributed and redistributed at any point in the program, and passed via the subroutine interface. SVM-

Fortran supports standard distributions like BLOCK, CYCLIC, and GENERAL_BLOCK, *indirect distributions* and *linked distributions.* The programmer can specify for each template element the target processor by an arbitrary integer expression within an indirect distribution. Linked distribution is a form of alignment where a distribution is described via the distribution of another template.

The example in Figure 1 illustrates the work distribution features of SVM-Fortran. The first loop uses direct scheduling. The block scheduling strategy gives a good load balance but no data locality since the iteration space is only a quarter of the problem domain.

In the second loop, data locality is optimized since predefined scheduling ensures that iterations are performed by the same processor as for loops running over the entire domain.

Creatable templates, such as T2, can be used if the iteration space is runtime dependent. The REDISTRIBUTE-directive allows to specify a distribution for such templates and can be used for dynamic load balancing.

The last parallel loop illustrates the use of semi-dynamic scheduling. When the loop is executed, the template's distribution is determined according to the dynamic scheduling decision. Unless the distribution is reset via the UNDEF-directive, subsequent executions of the parallel loop reuse the schedule to maintain data locality.

4 Integrated Programming Environment

Parallelization of real-world scientific applications requires the integration of the language, the SVM implementation, as well as of programming tools into a homogeneous programming environment.

For example, the programmer needs information about remote accesses to be able to optimize data locality. The very basic information about non-local behaviour is the individual page fault resulting from an access to a virtual address currently not residing in the node's memory. Such information has to be gathered in the kernel together with the faulting address and instruction. The information has to be presented in the form of program variables and source locations so that the user can easily understand it. Specific information is required from the compiler, the language runtime system, the kernel, and the program text to relate runtime data to the source text.

The following sections outline the components of the programming environment and describe their interaction.

4.1 SVM-Fortran Compiler

The SVM-Fortran compiler is a source-to-source compiler generating Fortran77 code with calls to a special runtime library.

At the beginning of the program, a shared segment is requested from the ASVM and shared common blocks are allocated from the shared segment. Shared variables need to be allocated at the beginning of every subprogram as several processor sets can execute independently a subprogram. For each processor set a private copy of these shared data has to be allocated. In contrast to shared data, private data needs no special handling as

```
        SUBROUTINE G(....,T,N,...)
CSVM$   TEMPLATE:: T(N)                ! fixed size template
CSVM$   TEMPLATE:: T1(N,N)             ! automatic template
CSVM$   TEMPLATE:: T2(:,:)             ! creatable template
CSVM$   PROCESSORS:: P(2,NUMPROC()/2) RESHAPE:: P1(NUMPROC())
CSVM$   DISTRIBUTE (BLOCK,BLOCK) ONTO P :: T1

        ...
        M= N/2
CSVM$   PDO(LOOPS(I,J),PROCESSORS(P),STRATEGY(BLOCK,BLOCK))
        DO I=1,M
              DO J=1,M
                    ...
              ENDDO
        ENDDO

CSVM$   PDO(LOOPS(I,J),STRATEGY(ON_HOME(T1(I,J))))
        DO I=1,M
              DO J=1,M
                    ...
              ENDDO
        ENDDO

CSVM$   CREATE:: T2(M,M)
CSVM$   REDISTRIBUTE (BLOCK,BLOCK) ONTO P :: T2
        ...

CSVM$   UNDEF:: T
15      CONTINUE
              ...
CSVM$         PDO (LOOPS(I),PROCESSORS(P1),
CSVM$*              STRATEGY(DISTRIBUTE_ONCE(T(I),GRAB(SELF_GUIDED))))
              DO I=1,M
                    ...
              ENDDO
              ...
              IF (...) THEN
CSVM$               UNDEF:: T
              ENDIF
        GOTO 15
```

Figure 1: SVM-Fortran example program

```
        DIMENSION A(N)
CSVM$   PDO(LOOPS(I), STRATEGY(ALIGNED(A(I))))
        DO I=1,N
              A(I) = ...
              ...
        ENDDO
```

Figure 2: Aligned Scheduling

it is mapped to the local memory of each processor.

As already mentioned, tasks can be executed either in exclusive or replicated mode. In exclusive mode it is necessary to suspend all but one processor and to reactivate those processors on parallel constructs. The suspended processors are waiting in a dispatcher loop for continuation messages of the APS leader.

An important issue for the compiler is the scheduling of parallel loops. For direct scheduling – e.g. block or cyclic distribution of loop iterations – code is generated which efficiently handles these special cases. For predefined and semi-dynamic scheduling with templates it is in general unknown at compile time which distribution strategy will be used at run time, e.g. how the template elements, and thus the loop iterations, are distributed to the processors. Therefore, for those loops a more general code is generated which is able to handle all work distribution strategies.

Task scheduling is not only implemented by the compiler and the language runtime system. Scheduling strategies such as *aligned scheduling*, which are applied to dynamically optimize data locality, require monitoring data. With aligned scheduling, loop iterations are assigned to the processors according to the current page distribution. In the example in Figure 2, ASVM page state information (Section 4.3) is used to determine the owner[1] of the page where `A(I)` is located.

SVM-Fortran provides three different types of synchronization. Barrier synchronization is the most important method as the semantics of our language requests that at certain points in the computation all processors in a processor set have to reach a point until one of them can continue. Barrier synchronization is done with a hierarchical algorithm based on message passing. As we support nested parallelism, only processors in the active processor set are involved in this operation. The other two synchronization methods are critical sections and page locks. To implement critical sections we use the underlaying SVM-mechanisms to obtain and set the lock for the critical section. As the implementation of page locks in the ASVM is not finished yet, currently we use a sub-optimal solution based on critical sections.

The compiler performs optimizations on different levels. The most important ones are:

- barrier removal: As barriers are expensive an important optimization is the removal of barriers where program semantics is not changed.

[1] In ASVM every page has a processor which owns the page, usually the processor which has written to that page lately.

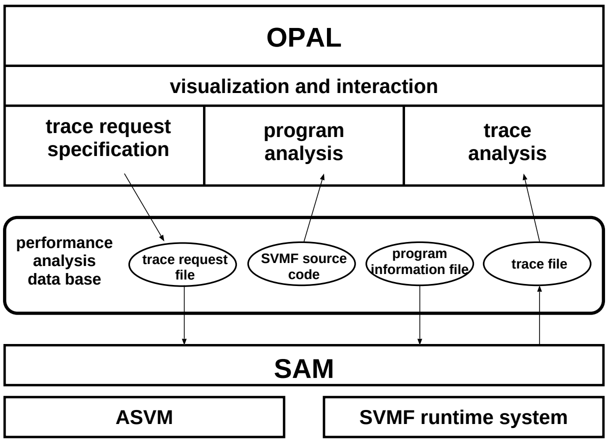

Figure 3: Performance analysis with OPAL

- avoidance of processor set computation: The overhead induced with the administration of processor sets can be reduced or even neglected for some common situations.

In addition to optimizations performed in the compiler, optimizations can be performed by an optimization tool taking monitoring information of individual program runs into account. This is a future extension of OPAL described in the next section.

4.2 OPAL: Optimizer and Locality Analyzer

OPAL is a performance analysis and optimization tool that integrates program text and monitoring information to support the user in optimizing data locality [9]. Figure 3 illustrates the performance analysis features of OPAL and its interfaces to the other components of the programming environment.

OPAL interacts with the monitoring subsystem via a performance analysis data base. The overall performance analysis process is incremental, i.e. starting from a coarse overview of the runtime behaviour, more detailed information for specific code regions is gathered in subsequent program runs.

The concept of incremental performance analysis was chosen to reduce the amount of monitoring data and thus to get around the performance impact on the application. Central to this concept is a hierarchically structured trace format and an efficient implementation of selective tracing.

The trace format provides events of different granularity. For example, on the coarsest level page fault sums are available. More information is provided by the page travel event that includes for each other process a counter for the page faults serviced by this process. On the finest level, events are generated for individual page faults.

Selective tracing is implemented via a *trace request file*. The program is compiled for performance analysis only once. This step inserts hooks to the SVM-Fortran Application Monitor (SAM). When the program is started, SAM reads the trace request file which includes trace requests such as

LOCAL foo.PDO_LABEL(150): RPF A, WPF A

specifying that read and write page fault events should be generated for array A during execution of the parallel loop with label 150 in subroutine FOO.

SAM translates program symbols like labels and variables with the help of the program information file generated by the SVM-Fortran compiler into runtime values. According to the trace requests SAM collects the information from the appropriate runtime data structures and requests SVM-specific information like page fault information from the ASVM. The ASVM provides a very flexible interface which supports the collection of data for arbitrary address regions.

All stages of the performance analysis process are supported by OPAL. In the current version the user can interactively generate trace requests for individual program regions. In contrast to trace visualization tools, OPAL presents gathered information based on the source code similar to profiling tools, e.g. page fault sums and execution times are presented as annotations to the source code. In addition, page faults are not presented in form of time line charts but for individual loops the faulting references and the faulting array elements can be inspected.

Requesting and analyzing information based on the program text is a natural approach for performance analysis. Of course, performance analysis can also benefit from visualization techniques. We plan to interface the graphical visualization tool PARvis [10] developed at KFA that includes state-of-the-art visualization techniques and allows trace analysis of different granularity via sophisticated zooming and flexible scrolling techniques.

4.3 Kernel Monitoring Support

Monitoring support is required for tuning of application programs as well as tuning of the ASVM system itself. On demand, ASVM provides extensive monitoring information in a raw form that has to be processed by appropriate tools to be easily understandable.

Major requirements for monitoring support are low perturbation of the application program, little memory consumption, and an efficient and easy to use interface. ASVM's monitoring system meets these requirements by providing selective instrumentation, information on different levels of granularity, and memory mapped monitoring data.

Monitoring support of ASVM consists of two major parts. First, coarse grained information on system level. A number of counters are implemented on each node, counting all SVM messages and the hits and misses of the ASVM caches. From these counters statistics about the system load (e.g. page faults and message traffic) caused by the ASVM system can be derived. Overhead of the counters is negligible and behaviour of the application program is not influenced because the counters are always switched on. Only one page of physical memory that is mapped into the address space of each task is required. The counters are read by a simple memory access.

The second major part of monitoring support of ASVM is fine grained information on application level. This part is more extensive and can be divided into three groups:

- **Counters**
  Counters for read or write page faults sum up the number of page faults on a specified

region of shared virtual memory and the time ASVM needs to supply the page. Additionally, page travel counters sum up the number of pages that were received from other nodes, defining a separate counter for every node.

- **Traces**
  Specific trace records store information for read or write page faults, for page invalidation messages, or when the access permission for a page is reduced from writable to readable. Data collected in traces is more detailed than counters and reflects the chronological order of the traced events. The disadvantage of detailed traces is the enormous consumption of memory. In order to keep the traces as small as possible and to allow user interaction, selective instrumentation is provided. Moreover, when a trace event occurs, a part of the user address space can be copied into the trace buffer. This feature allows to assign a trace event to the value of the selected user data (e.g. a loop counter) when the event occurred. A special function is provided for generating a consistent copy of the trace buffer.

- **Page State Information**
  For each page of a given virtual address range the state of the page (e.g. read only, owned, writable, paged out, locked) is provided. Besides performance monitoring, this information can also be used for tuning of applications, e.g. it is used to implement aligned scheduling (Section 4.1).

All monitoring data (i.e. counters, trace buffers, trace definitions, and page state information) is mapped into the address space of the application task and thus can be easily accessed by the application program or a monitoring library linked with the application. Counters and traces can be restricted to a specified region of shared virtual memory and a code region of the application program. They can be specified for a single thread or all threads of a task. The monitored regions of shared virtual memory may overlap and the boundaries of the regions need not be aligned with page boundaries. Thus, counters and traces can be defined for single variables or for a procedure or even a single statement that is executed by a selected thread.

This flexibility enables selective monitoring and keeps memory consumption and perturbation of the application program small. The overhead increases proportionally to the amount of monitoring support defined and thus is adjustable by the user. As long as no monitoring is defined, there is virtually no overhead.

5 Early Performance Results

First measurements show that basic SVM operations (e.g. obtaining a page for read or write access, or invalidation of pages) on the ASVM system are about two to ten times faster compared to external server implementations on the Paragon^{TM} Supercomputer [16]. The main reasons for this difference are the features of ASVM described in Section 2. We expect to improve these results by further tuning of ASVM. The final paper will contain basic performance data.

In Table 1 we give performance numbers for the implementation of template operations. The numbers given in the table are independent of the number of processors since each operation is performed independently on each processor. Performance ranges for template operations show execution times for 1-dimensional to 7-dimensional templates. If the operation requires a synchronization and the barrier cannot be eliminated by the compiler the execution time of the barrier has to be added to the execution time.

operation	execution time (in μsec)
create template (1-7 dimensions)	12.8-14.0
destroy template (1-7 dimensions)	10.9-14.1
undefine template (1-7 dimensions)	12.6-18.7
distribute a template by block (1-7 dimensions)	176-259

Table 1: Execution time of template operations

Table 2 shows the loop overhead for a sequential loop and for parallel loops scheduled with direct scheduling and predefined scheduling using a block distribution. Again, these numbers do not include the barrier overhead. Although predefined schdeuling is more expensive than direct scheduling the loop overhead is rather small. As irregular distributions are stored as a list of blocks of template elements assigned to the processor, this very flexible distribution format can be handled with similar efficiency.

loop type	loop overhead (in μsec)
sequential do	0.2
direct scheduling of PDO (block distr.)	0.8
predefined scheduling of PDO (block distr.)	29.6

Table 2: Loop overhead

Barriers are implemented with an $O(\log n)$ algorithm using message passing. As the communication startup time on the Paragon^{TM} Supercomputer dominates the execution time of barriers, barriers limit the overall application performance. For applications with a large number of small parallel loops it can be the most important performance bottleneck. As already pointed out, we try to eliminate as many barriers as possible in the optimization phase of our compiler.

One of the codes ported with the help of SVM-Fortran is a Crystal Growth Simulation [22]. It is an application developed at KFA for the optimization of the silicon production process. For the quality of the silicon crystal a constant convection in the melt is very important. The convection results from the heating, the rotation of the crucible, and the rotation of the crystal. The convection is modeled by a set of partial differential equations and determined by an explicit finite difference scheme. The selected work distribution assigns a part of the crucible to each processor. Due to the finite difference scheme page faults occur mainly for accessing objects at the boundary of a neighbouring domain. Table 3 shows the performance results. Due to the size of the domain the code could not be executed on less than four processors without paging to disc.

This example shows good speedups up to 16 processors. For more processors the problem

procs	time (sec)	speedup	barrier time (sec)	page faults per processor	page fault time (sec)
4	104.8	1.0	5.9	822	2.5
8	56.2	1.9	6.1	948	3.4
16	31.2	3.4	6.0	1018	3.4
32	20.4	5.1	6.2	1075	4.0
64	16.2	6.5	6.4	1136	4.1

Table 3: Performance of Crystal Growth Simulation

size of 42 × 92 × 202 (60 MB shared data) is too small. The main performance losses are the barrier synchronization and also the service time for page faults.

6 Related Work

The most well-known approach for providing a global address space on distributed memory systems is HPF which relies on compiler technology to generate explicit message passing. Although promising results are available for regular grid applications, considerable problems exist in compiling irregular grid applications. In such applications, the access patterns are unknown at compile time and expensive runtime techniques have to be applied.

The MYOAN/Fortran-S project at INRIA/IRISA [14] is a research project with similar goals than ours. MYOAN uses the external pager interface of MACH to implement SVM on the Paragon^{TM} Supercomputer in software. Fortran-S is a Fortran77 extension which supports a SPMD execution model. Previously it was developed for KOAN, a SVM implementation on the Intel iPSC/2, and had a great influence on the design of SVM-Fortran. In addition to features also available in Fortran-S, SVM-Fortran provides nested parallelism, dynamic loop scheduling, work distribution templates, and the two execution modes.

An implementation of an external server called MaX which is similar to MYOAN is done at the Technical University München [15].

There are some research projects ongoing as well as commercial products available which implement a single address space on a distributed memory computer using dedicated hardware. Among them are Flash at Stanford [11], Alewife at MIT [12], SPP1000 of Convex Computer Corp. [13], and KSR-1 of Kendall Square Research [7].

7 Status and Future Work

Currently, the SVM-Fortran compiler and ASVM are stable prototypes which have been used to implement several application codes. Performance analysis for these applications is done with a very early version of OPAL.

In the near future, ASVM will be tuned and enhancements like prefetching will be integrated. The performance analysis tools OPAL and PARvis will be extended to support full

SVM-Fortran and will be based on SAM for gathering performance data. Extensive tests with large scientific applications will increase stability of the programming environment and will give insight in the performance bottlenecks and potential optimizations.

In the long run, the programming environment will be ported to DSM systems providing a global address space supported by hardware. On these machines, access to remote data will be much faster than on SVM systems, but will be still considerably slower than access to local memory. Thus, locality analysis and optimization will remain the key issues for efficient programming.

Performance analysis is currently done by visualization either based on the source code or graphically. Future analysis tools should automatically identify performance bottlenecks and therfore need a knowledge base about types of potential performance bugs.

Optimizations developed for codes running on SVM systems will be of similar importance for DSM systems. Currently, optimizations such as shared to private coercion and selection of a well-suited work distribution are carried out manually. In the future, such optimizations have to be triggered and performed automatically. Static program information is not sufficient to select such optimizations since their outcome frequently depends on runtime properties. Therefore, we have to implement an automatic optimization cycle on top of trace-based performance analysis tools and optimizing transformation systems.

References

[1] F. Bodin, T. Priol, *Overview of the Koan Programming Environment for the iPSC/2 and Performance Evaluation of the BECAUSE Test Program 2.5.1*, Proc. of BECAUSE European Workshop, 1992, Sophia-Antipolis

[2] K. Li, *IVY: A Shared Virtual Memory System for Parallel Computing*, Proceedings of 1988 International Conference on Parallel Processing, Vol II, pp. 94-101, 1988

[3] K. Li, *Shared Virtual Memory System on Loosely Coupled Multiprocessors*, Ph.D. Thesis, Yale University, Technical Report YALEU-RR-492, 1986

[4] R. Berrendorf, M. Gerndt, W. Nagel, J. Prümmer, *SVM-Fortran*, Interner Bericht KFA-ZAM-IB-9322, Zentralinstitut für Angewandte Mathematik, Forschungszentrum Jülich, 1993

[5] HPFF, *High Performance Fortran Language Specification*, High Performance Fortran Forum, May 1993, Version 1.0, Rice University Houston Texas

[6] F. Bodin, L. Kervella, T. Priol, *Fortran-S: A Fortran Interface for Shared Virtual Memory Architectures*, Proceedings of Supercomputing 93, Portland, 1993

[7] Kendall Square Research, *Technical Summary*, Waltham, Massachusetts, 1992

[8] F. Darema-Rogers, V.A. Norton, G.F. Pfister, *Using a Single-Program-Multiple-Data Computational Model for parallel Execution of Scientific Applications*, Research Report RC 11552 (#51726) 11/19/85, IBM Watson Research Center Yorktown Heights, 1985

[9] M. Gerndt, *Performance Analysis Environment for SVM-Fortran Programs*, Interner Bericht KFA-ZAM-IB-9417, Zentralinstitut für Angewandte Mathematik, Forschungszentrum Jülich, 1994

[10] W.E. Nagel, A. Arnold, *Performance Visualization of Parallel Programs - The PARvis Environment -* , Caltech Technical Report, CCSF-47, +1994

[11] J. Kuskin et al., *The Stanford FLASH Multiprocessor*, Proc. 21st Int. Symposium on Computer Architecture, pp. 302-313, 1994

[12] A. Agarwal et al., *The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor*, MIT/LCS Memo TM-454, MIT, 1991

[13] Convex Computer Corp., *An Overview of the Exemplar Series SPP1000 System*, Richardson, Texas, 1994

[14] G. Cabillic, T. Priol, I. Puaut, *MYOAN: an Implementation of the KOAN Shared Virtual Memory on the Intel Paragon*, Internal publication 812, IRISA, Rennes, France, 1994

[15] R.G. Hackenberg, *MaX – Investigating Shared Virtual Memory*, Proceedings of HPCN Europe, LNCS 797, pp. 308-315, 1994

[16] J. Reinhold, *Funktionalitätserweiterung des MaX-SVM-Servers*, Diploma Thesis, Technische Universität München, Institut für Informatik, 1994

[17] *Paragon OSF/1 Operating System*, Specification Sheet, Order No. 206/10-92/RJ/GA, Intel Corporation Supercomputer Systems Division, 1992

[18] R. Zajcew, P. Roy, D. Black, C. Peak, P. Guedes, B. Kemp, J. LoVerso, M. Leibensperger, M. Barnett, F. Rabii, D. Netterwala, *An OSF/1 Unix for Massively Parallel Multicomputers*, Proceedings of the Winter 1993 USENIX Conference

[19] D. Golub, R. Dean, A. Forin, R. Rashid, *Unix as an Application Program*, Proceedings of the Summer 1990 USENIX Conference

[20] *Guide to OSF/1: A Technical Synopsis*, O'Reilly & Associates, Inc., 1991

[21] A. Forin, J. Barrera, M. Young, R. Rashid, *Design, Implementation, and Performance Evaluation of a Distributed Shared Memory Server for Mach*, Carnegie-Mellon University, CMU-CS-88-165, 1988

[22] M. Mihelcic, H. Wenzl, K. Wingerath, *Flow in Czochralski Crystal Growth Melts*, Report No. 2697, Research Centre Jülich, ISSN 0366-0885, 1992