Compiling SVM-Fortran for the Intel Paragon XP/S

Rudolf Berrendorf, Michael Gerndt
Zentralinstitut für Angewandte Mathematik
Forschungszentrum Jülich GmbH (KFA)
D-52425 Jülich, Germany
{r.berrendorf,m.gerndt}@kfa-juelich.de

Abstract

SVM-Fortran is a language designed to program highly parallel systems with a global address space. A compiler for SVM-Fortran is described which generates code for parallel machines; our current target machine is the Intel Paragon XP/S with an SVM-extension called ASVM. Performance numbers are given for applications and compared to results obtained with corresponding HPF-versions.

Keywords: massive parallelism, global address space, compiler, template, control flow.

1 Introduction

Today, several programming models compete as the adequate model to program highly parallel systems with up to several thousands of processors. In the message passing approach – often used with an SPMD execution model [5] – the distributed memory is visible to the programmer; non-local data access has to be specified explicitly in the program with message passing operations for the requesting side as well as for the service side. Most programmers agree that message passing programs are difficult to write. Other programming models try to hide the distributed memory from the programmer and emulate a global address space either with special language constructs, through the compiler, as an operating system extension, or in hardware. On top of this global address space, an adequate execution model needs to be defined.

In this paper we describe a compiler for SVM-Fortran; the language is designed to program highly parallel systems with a global address space. Currently, we use the Intel Paragon XP/S as our target machine. The global address space on this distributed memory computer is emulated by a shared virtual memory implementation called ASVM [15] which is integrated into the MACH-3 kernel. Shared virtual memory (SVM) [8] emulates a global address space on top of the distributed real memory on a cache line or page base, similar to a virtual memory. Special protocols ensure a consistent view on all processors.

The paper is organized as follows: The following section relates our work to other research in that field. In section 3, an overview of SVM-Fortran is given. Section 4 gives details on the compiler. Finally, performance results for applications written in SVM-Fortran are given in section 5.

2 Related Work

Closely related to implementation aspects of the compiler is the underlying programming model and the language constructs the compilers translate. We will give an overview of relevant work on this field.

HPF[7] is an extension to Fortran 90 with which the programmer specifies a distribution for data and the compiler generates appropriate message passing operations, usually following the owner-compute-rule. Parallelism is available in HPF in form of vector statements and forall-loops. With the current HPF language version there are considerable problems to handle irregular data accesses. Ongoing efforts try to extend HPF in several directions. There are several compilers for (a subset of) HPF available. Similar to our SVM-compiler, compilers for HPF need to handle templates to store either data distributions (HPF) or work distributions (SVM-Fortran).

Fortran-S [3] is a Fortran 77 extension which supports a SPMD execution model. In addition to features also available in Fortran-S, SVM-Fortran provides nested parallelism, dynamic loop scheduling, work distribution templates, and two execution modes. In Fortran-S, assignments to shared variables or variables of unknown sharing type need barrier synchronizations in front of and after the assignment. A

0-8186-7177-7/95 $04.00

major task for the compiler is therefore to reduce the number of barriers through intra- and interprocedural analysis.

In the CRAFT Model of Cray Fortran [11], shared data is distributed to processors via user-specified distribution directives; a global address space is implemented by the compiler based on the data distribution information. The execution model is based on a SPMD-model with work sharing. Inside a sequential region, there is no way to switch to parallel mode; thus, a real nesting of language constructs is not available. Locality within parallel loops can be reached with an aligned iteration distribution based on distribution information for a variable.

KSR-Fortran [12] has several language constructs to specify parallelism based on a fork-join execution model: parallel loops, parallel sections, and parallel regions. As on other machines, data locality is a main issue on the KSR. As this is a problem global to a program rather than for a small section of code, work scheduling can be done on the KSR with thread teams and affinity regions. Thread teams are a named set of threads with which the user can tell the compiler or runtime system that only those threads belonging to the team should work on some piece of code. Affinity regions establish for a larger section of code a relationship between threads and iteration scheduling of parallel loops, i.e. the same iterations of different loops should be handled by the same threads within an affinity region. Due to the fork-join model, startup times on parallel constructs are quite high on the KSR.

The ANSI X3H5 Technical Committee on Parallel Processing Constructs for High Level Languages [6] has worked on an language-independent model for parallel processing. They proposed a general programming model conceptionally similar to PCF-Fortran [10] which is based on a fork-join execution model.

3 SVM-Fortran

SVM-Fortran is an extension of Fortran 77 providing annotations for shared memory parallel programming on highly-parallel machines with a global address space. It supports the SPMD model and, moreover, provides nested parallelism and work distribution directives.

Nested parallelism is useful for programming applications consisting of multiple coarse-grained parallel tasks which are themself data parallel codes. The work distribution directives allow to specify the mapping of parallel work onto processors. This mapping is responsible for the data locality of the parallel applications.

Execution Model

A SVM-Fortran program is executed by a set of abstract processors which are available from the beginning of the application. SVM-Fortran uses the term abstract processors instead of processes to reflect the assumption that there is a one-to-one correspondence of abstract and physical processors assigned to an application.

A *task* is defined by a *program region*, either a whole subroutine, a section of a parallel section construct, or the body of a parallel loop. A task is executed by a set of abstract processors, called the task's *active processor set* (APS). At program start the whole program forms a single task with an APS consisting of all abstract processors.

Tasks are either executed in exclusive or in replicated execution mode. In *exclusive mode*, only one processor of the APS executes the code while the others are suspended until the *master processor* encounters a parallel construct and the other processors resume execution at this construct. In *replicated mode*, all processors in the APS execute the task redundantly.

Parallel constructs in SVM-Fortran are parallel loops and parallel sections. When the master processor encounters a parallel construct, the APS is split up into new *derived processor sets* according to the work distribution. A parallel construct defines new parallel tasks, i.e. loop iterations and individual sections, that are assigned to the derived processor sets. After all new tasks have been executed, the abstract processors continue the execution of the original task within the task's APS.

Work Distribution Directives

Explicit distribution of tasks onto abstract processors is the main concept for creating data locality. The specification of a work distribution strategy in SVM-Fortran is based on processor arrangements and templates. This section outlines the basic features, details can be found in [2].

Processor arrangements are rectangular index spaces that are used to name abstract processors. Processor arrangements are declared with the `PROCESSORS` directive. Figure 1 illustrates the distribution of individual sections onto sets of abstract processors. Since multiple processors are assigned to each section, data parallelism can be exploited in each subroutine.

```
CSVM$ PROCESSORS:: P(10),P1(10)

CSVM$ PSECTION
CSVM$ SECTION ON P
        CALL F()
CSVM$ SECTION ON P1
        CALL G()
CSVM$ PSECTION_END
```

Figure 1: Explicit distribution of parallel sections.

```
CSVM$    PROCESSORS:: P(NUMPROC())

CSVM$    TEMPLATE:: NODES(17000)
CSVM$    DISTRIBUTE (BLOCK) ONTO P:: NODES
CSVM$    TEMPLATE:: BOUNDARY_NODES(1000)
CSVM$    DISTRIBUTE (I) ONTO
CSVM$+    HOME(NODES(INDIRECT(I))):: BOUNDARY_NODES
```

Figure 2: Using linked distribution for irregular grids.

The *template concept* of SVM-Fortran is used to specify the work distribution for parallel loops globally. A template is a rectangular index domain which reflects the iteration space of the parallel loops. Templates can be either declared statically or created dynamically according to the actual problem size of an application. Similar to templates in HPF, SVM-Fortran templates can be explicitly distributed onto abstract processors. In contrast to HPF, where it is very important that the compiler knows the distribution of templates, SVM-Fortran templates can be redistributed everywhere in the program.

SVM-Fortran provides the *standard distribution formats* BLOCK, CYCLIC, GENERAL_BLOCK, and *indirect distribution*, and *linked distribution*. While the standard distribution formats assign blocks of template indices to processors via a predefined strategy, both other strategies specify the target processor for each template element individually. The main application area for these two strategies are irregular-grid applications where the numbering of grid nodes does not reflect their spatial relation.

Figure 2 illustrates the use of linked distribution. The computation for the boundary nodes of the finite element grids shall be distributed to the processors responsible for the finite element nodes in the regular numbering scheme.This construct does not introduce a permanent relationship, i.e. redistribution of NODES does not imply automatic redistribution of BOUNDARY_NODES.

Figure 3 outlines global loop scheduling via templates called *predefined scheduling*. The ON_HOME clause assigns an iteration to the home processor of the appropriate template element. Due to our implementation described in subsequent sections, predefined scheduling is not much more expensive than directly annotating loops with a work distribution strategy as described in the following. Thus, the comfort of programming can be exploited without severe performance penalty.

```
CSVM$ PROCESSORS:: P(NUMPROC())
CSVM$ TEMPLATE:: T(N)
CSVM$ DISTRIBUTE (BLOCK) ONTO P:: T

C     -- direct scheduling --
CSVM$ PDO(LOOPS(I),STRATEGY(BLOCK))
      DO I=1,N
         ...
      ENDDO

C     -- predefined scheduling --
CSMV$ PDO (LOOPS(I),STRATEGY(ON_HOME(T(I))))
      DO I=1,N
         ...
      ENDDO
```

Figure 3: Template distribution and loop scheduling.

In addition to predefined scheduling, loops can be annotated individually where the work distribution can be either *direct* or *dynamic*. Direct strategies are simple strategies like BLOCK and CYCLIC that assign the iterations to the processors in the APS. With dynamic strategies such as *self scheduling*, loop iterations are assigned to processors according to the current work load.

Further, a combination of dynamic work distribution strategies and template distributions called *semi-dynamic scheduling* is available. This combination allows to store a schedule resulting from a dynamic strategy in form of a template distribution.

4 Compiling SVM-Fortran

Currently, our primary target system is the Intel Paragon XP/S with a shared virtual memory extension called ASVM[15]. ASVM is integrated into the MACH-3 micro-kernel of the operating system. There are other external servers available for the Intel Paragon XP/S as well, e.g. MYOAN [4].

The SVM-Fortran compiler is a source-to-source compiler generating intermediate Fortran 77 SPMD code with calls to a special runtime library. Currently, we are able to generate code for the Intel Paragon

XP/S as well as for single and multi-processor Unix systems. The intermediate code is then translated by the native compiler of the target system.

Processor Control

A central point for the implementation of the language is the concept of a *processor set* which is a set of nodes working together on a part of the program. Processor sets and derived processor sets form a tree of processor sets with active processor sets on leave nodes. Often used operations on processor sets are:

- Generate a new processor set, suspend the old one, and mark the new one as active, e.g. this is done on entering a parallel section. As our execution model is SPMD, the generation of processor sets can be done efficiently without the need to generate new processes or threads, or to signal other processors that the generation has to be done.
- Delete a processor set and re-activate the old one. An example is the end of a parallel section.
- Perform operations on a processor set, e.g. do a barrier on a processor set.

Data structures necessary for processor sets are stored and locally administrated on every processor, i.e. the above mentioned operations are done locally on every processor without the need to synchronize or to exchange information. Exceptions to this asynchronous operation mode are the generation of processor sets where local information is not sufficient to determine which processors belong to a processor set e.g. if dynamic loop scheduling is used and more processors than iterations are available.

As already pointed out in the description of the language, there are two execution modes possible for a part of a program: *exclusive mode* and *replicated mode*. Dependent on this execution mode either all or only one processor of a processor set execute the code.

In *replicated mode* all processors of a processor set execute code with independent flow of control. As we generate SPMD-code, the original code needs no special handling concerning control flow for these sections of the program. The only exception are non-coordinated subprogram calls. Each such call is executed within a new processor set consisting of one processor only.

In *exclusive mode*, a master processor of the active processor set executes the code. The master processor of a processor set is always the processor for which the function MYPROCSET() returns 0. Other processors of the active processor set are suspended and are re-activated on some language constructs (e.g. parallel loop, template distribution) to share work. We implement this with dispatch-operations (see Figure 4) which let the master processor execute the code while the other processors of the processor set wait for a special message from the master.

As can be seen in the figure, non-master processors are blocked at the begin of the subroutine by the call to SVM_DISPATCH_RECV(). This function returns an integer value which is interpreted as a jump target where the non-master processors should continue their execution. Code for the master processor is inserted into the program to send appropriate messages (CALL SVM_DISPATCH_SEND()) to non-master processors where the latter one should continue their work. Language constructs where non-master processors are re-activated are:

- parallel loop, parallel section (R)
- exclusive region, replicated region (R)
- coordinated call
- create, destroy, undef, redistribute
- subprogram calls

For those language constructs marked with an R, the code generation for control flow has to be done recursively. For example, inside a parallel section non-master processors wait on an additional dispatch loop, and for each of the above language constructs inside the section, continuation messages are sent to the non-master processors by the master processor. At the end of the section, the master sends a continuation message such that all processors of the processor set reach that point and can continue in the control flow of the upper hierarchical control flow level.

Compared to active messages [14] our approach has the advantage that stack frames are set up correctly. The disadvantage compared to active messages is the overhead involved with dispatch operations.

Data Handling

SVM-Fortran distinguishes between shared and private data. *Private data* is replicated in a SPMD-model to each processor's local memory. *Shared data* is allocated from a shared segment; the shared segment is requested from the SVM-subsystem during the initialization phase. Shared variables are allocated at the

original code:

```
      PROGRAM dispatch
CSVM$ EXCLUSIVE_REGION
      i = 1
      CALL sub
      i = 2
CSVM$ EXCLUSIVE_REGION_END
      END
```

intermediate code:

```
      PROGRAM dispatch
C***  dispatcher-loop for non-master processors
45002 IF(SVM_MYPROCSET.NE.0)
     +   GOTO(45003,45004),SVM_DISPATCH_RECV()
      I = 1
C***  master processor re-activates other processors
      CALL SVM_DISPATCH_SEND(2)
45004 CONTINUE
      CALL SUB
c***  non-masters go back to dispatcher loop
      IF(SVM_MYPROCSET.NE.0) GOTO 45002
      I = 2
C***  master processor re-activates other processors
      CALL SVM_DISPATCH_SEND(1)
45003 CONTINUE
      END
```

Figure 4: Dispatch operations in exclusive mode.

beginning of every subprogram and freed before a return is executed. Incarnations of the same subroutine by different processor sets therefore do not conflict. Common blocks are allocated at the beginning of the main program and freed at the end of the main program. In general, freeing shared memory requires a barrier synchronization between all processors of the active processor set. *Partially shared data* is shared between processors of one processor set only and can be introduced with parallel loops and parallel sections. This type of data can be handled like shared data with a data scope of the language construct where the data was introduced, e.g. a parallel loop. Alignment of variables and common blocks on page boundaries is handled with compiler-generated padding.

Templates

Data structures for templates are stored locally. Every template gets from the compiler a unique identifier which is an index to a run-time descriptor. Templates used as a dummy argument get a unique identifier, too. Template descriptors contain among others fields the rank, dimensions, internal state (empty, declared, defined), and the distribution for every dimension. On a template definition, the SVM-Fortran compiler writes the invariable information concerning the template to the compiler information file: template name, subprogram name of declaration environment, and rank of template. During the initialization phase of the run time system, those information is read and stored in the run time descriptor for each template. Similar to processor arrangements, constant dimension values are passed by the compiler to the run time system through the compiler information file while non-constant values are passed through special run time calls. The internal state of a template changes after certain operations:

- After a create-directive a template has the state *declared.*
- Also, after a template-directive in which the rank and all dimension values are specified the state is *declared.*
- After a template-directive in which only the rank was specified the state is *empty.*

Passing a template as an argument to a subprogram is handled similar to processor arrangements: An integer variable gets assigned the unique identifier for that template, and in the called subprogram a check is made if actual and formal argument agree in rank and dimensions.

Template operations like *define*, *create*, *undef*, and *destroy* are local operations with respect to every processor, i.e. every processor does not need any information from other processors. *Distribute* and *redistribute* operations may need coordination between all processors in a processor set to exchange necessary data, e.g. with a linked distribution.

To distribute or redistribute a template, it is necessary that the template has the state *declared* or *defined*; after such an operation the template has the state *defined.* Additionally, the current processor set is stored in the template descriptor.

Dependent on the distribution type (e.g. block distribution or linked distribution), different parameters describing the specific distribution are stored; they are

chosen for every distribution strategy such that the runtime overhead for loop scheduling is kept small. The values for the describing parameters are determined at distribution time.

Block distribution: For a block distribution with N template elements the lower and upper bound assigned to a processor p ($1 \leq p \leq P$ = number of processor in a processor set) is calculated according to:

$$lower\_bound = \left\lceil \frac{N}{P} \right\rceil \cdot (p-1) + 1$$

$$upper\_bound = \min(lower\_bound + \left\lceil \frac{N}{P} \right\rceil - 1, N)$$

Then, to decide if a template element e is assigned to processor p, that processor has to determine only if e lies between the lower and upper bound (this test is necessary if the template is used in a linked distribution of another template).

Cyclic distribution: For a cyclic distribution with a segment size l the number of segments assigned to a processor p:

$$num\_seg(p) = \left\lceil \frac{N - (p-1) \cdot l}{P \cdot l} \right\rceil .$$

The lower bound and upper bound for segment i ($1 \leq i \leq num\_seg(p)$) on processor p is then:

$$lower\_bound(p,i) = \underbrace{(p-1) \cdot l + 1}_{A} + (i-1) \cdot \underbrace{P \cdot l}_{B}$$

$$upper\_bound(p,i) = \min(lower\_bound(p,i) + l - 1, N).$$

Each processor stores in its local template descriptor the values for l, $num\_seg(p)$, A, and B. With this information it is possible to list all template elements assigned to processor p (e.g. during loop scheduling). To decide if a template element e is stored on processor p, the following formula has to be evaluated and is true if e is stored on p:

$$p = MODULO(\left\lfloor \frac{e-1}{l} \right\rfloor, P) + 1$$

Indirect distribution: Indirect distributions, e.g. :

```
CSVM$ PROCESSORS:: P(N)
CSVM$ TEMPLATE:: T(N)
CSVM$ DISTRIBUTE (I) ONTO P(MAP(I)):: T
```

Table 1: Storage scheme used for inverted template distributions

number of local blocks: n	
$lower\_bound_1$	$upper\_bound_1$
...	...
$lower\_bound_n$	$upper\_bound_n$

are stored in such a way that every processor stores locally only those iterations that are assigned to the processor. That means that the distribution function has to be inverted to compute the assigned elements. The data structure is a list of blocks each containing a lower and upper boundary for a block, i.e. a from- and to-value for every block.

The advantage of this type of storage scheme is the efficiency with which loops can be executed as all information is available at loop execution time. The disadvantage is the (possible) memory overhead as every block needs two integer values.

Every processor involved in the distribution process executes a loop over all possible values of I (e.g. 1,...,N). If MAP(I) is equal to the local processor number, I is attached to the local list of blocks to execute. As the iterations are executed in increasing order the insertion in the list of blocks is easy: If the new index which has to be inserted is one more than the upper value of the last local block, this upper value is increased; otherwise a new block is opened which contains the index as the lower and upper boundary.

Linked distribution: Linked distributions, e.g. :

```
CSVM$ PROCESSORS:: P(3)
CSVM$ TEMPLATE:: T1(15), T2(5)
CSVM$ DISTRIBUTE (BLOCK) ONTO P:: T1
CSVM$ DISTRIBUTE (I) ONTO HOME(T1(3*I)):: T1
```

are handled similar to indirect distributions. An iteration is inserted in the local list of iteration blocks if the template element specified in the HOME-clause is assigned to the processor, i.e. is in the local block list for template T1.

Loop Scheduling

SVM-Fortran has a rich set of loop scheduling strategies from low-overhead static strategies (e.g. block, cyclic) to locality-driven strategies where the runtime overhead might be higher (e.g. predefined scheduling or aligned scheduling).

For direct static strategies as block scheduling efficient inline code is generated if inside the loop body no processor set information is needed (which is true for many loops). Dynamic strategies are still subject of our evaluation. For loops scheduled with predefined scheduling, the compiler usually does not know at compile time how template elements are distributed to processors. Therefore, code is generated which is able to handle any possible distribution.

A detailed description on our scheduling can be found in [1].

Synchronization

The programming model supports three forms of synchronization: barriers to ensure that every processor in a processor set has reached a point, critical sections to protect regions of code, and page locks to protect data. As the NX-Library on the Intel Paragon XP/S supports only barriers on all processors assigned to an application, we distinguish between the case that a processor set is identical with the initial processor set – in that case we use the system call – or if only a subset of the initial processor set is involved in the barrier – in that case we use our own barrier implementation with an hierarchical algorithm based on point-to-point message passing. Currently, we work on an efficient and scalable implementation for critical sections and page locks.

5 Performance Results

In this section we present performance results for two applications parallelized with SVM-Fortran. To relate our results to other programming models, we give the corresponding performance numbers for an HPF version and – in one case – for a message passing version, too.

Shallow Water Equations

The Shallow-Water-Equations [13] is a small application program for the simulation of the atmospheric flow based on a finite difference scheme. We used a grid size of 1024×1024 which fits on 8 processors and more without paging to disk. The work is distributed with templates to processors. We used the highest available optimization for the target compiler (i.e. -O4 -Knoieee -Mvect on the Intel Paragon XP/S). For comparism, we ported the same program to HPF and used the PGI HPF-compiler available on the Intel Paragon XP/S.

Table 2: Performance of Shallow Water Equations (times in seconds)

	SVM-Fortran			HPF
proc	time	barrier time	page fault time	time
8	56.85	2.67	4.18	56.25
16	31.07	2.54	2.92	35.95
32	18.08	2.37	2.49	29.83
64	11.54	2.38	2.10	39.13
128	8.35	2.68	1.68	75.23

To estimate the overhead for the parallel version we run the same program with smaller array dimensions (256×256) such that all data fit into the memory of one processor. Then, the time for the parallel SVM-Fortran version run on one processor (25.86 sec) was comparable to the time for the sequential version (25.22 sec).

Crystal Growth Simulation

The Crystal Growth Simulation [9] 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.

Table 3: Performance of Crystal Growth Simulation (times in seconds)

	SVM-Fortran			HPF	MP
proc	time	barrier time	page f. time	time	time
4	104.8	5.9	2.5	86	67.0
8	56.2	6.1	3.4	45	35.5
16	31.2	6.0	3.4	28	18.1
32	20.4	6.2	4.0	16	11.2
64	16.2	6.4	4.1	14	7.4

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 disk.

The SVM-Fortran execution times can be compared to the execution times for the HPF-version compiled

with the Portland Group HPF compiler and the execution of a hand-written message passing version. With respect to the much longer development time of the message passing implementation, almost half a year, the performance results of HPF and SVM are very good. Porting the code to the last two programming models was done in a few days.

This example shows good speedups up to 16 processors. For more processors the problem size of $42 \times 92 \times 202$ (60 MB shared data) is too small. The main performance losses are the barrier synchronization and also the service time for page faults.

6 Conclusion

SVM-Fortran is a language to program highly parallel systems with a global address space. The language supports nested parallelism and global work distribution through templates. Distinct from other approaches, we do not distribute data but rather work through the use of templates.

The compiler for SVM-Fortran generates intermediate code which is translated by the native compiler of the target system. Currently, code generation for the Intel Paragon XP/S with a shared virtual memory extension called ASVM is done; additionally, compilation for and running on single or multi-processor Unix systems is possible. Efficient SPMD-like code is generated with low runtime overhead. To support the exclusive execution mode, dispatcher operations were introduced. Special emphasis was taken to handle (global) work distribution with templates and work sharing with a large set of distinct scheduling strategies on parallel loops.

Performance numbers for two applications have shown good scalability. We expect to get better performance results when next-generation parallel systems are available which support a global address space efficiently.

References

[1] R. Berrendorf and M. Gerndt. Compiling Data Parallel Languages for Shared Virtual Memory Systems. Technical Report KFA-ZAM-IB-9517, Forschungszentrum Jülich (KFA), ZAM, 1995.

[2] R. Berrendorf and M. Gerndt. SVM-Fortran Reference Manual. Technical Report KFA-ZAM-IB-9510, Forschungszentrum Jülich (KFA), ZAM, April 1995.

[3] F. Bodin, L. Kervella, and T. Priol. Fortran-S: A Fortran Interface for Shared Virtual Memory Architectures. In *Supercomputing*. IEEE, 1993.

[4] G. Cabillic, T. Priol, and I. Puaut. MYOAN: An Implementation of the KOAN Shared Virtual Memory on the Intel Paragon. Technical Report 812, INRIA-IRISA, Rennes, France, April 1994.

[5] F. Darema, D.A. George, V.A. Norton, and G.F. Pfister. A Single-Program-Multiple-Data Computational Model for EPEX/FORTRAN. *Parallel Computing*, 7:11–24, 1988.

[6] Bruce Leasure (ed.). Parallel Processing Model for High Level Programming Languages. ANSI Technical Committee X3H5, March 1993.

[7] High Performance Fortran Forum. High Performance Fortran Language Specification, 1st edition, January 1993.

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

[9] M. Mihelcic, H. Wenzl, and K. Wingerath. Flow in Czochralski Crystal Growth Melts. Technical Report 2697, ISSN 0366-0885, Forschungszentrum Jülich, 1992.

[10] The Parallel Computing Forum. PCF Fortran: Language Definition, August 1988.

[11] D.M. Pase, T. McDonald, and A. Meltzer. MPP Fortran Programming Model. Technical report, Cray Research, Inc., Eagan, Minnesota, 1994.

[12] Kendall Square Research. KSR Fortran Programming. Waltham, Massachusetts, 1 edition, 1992.

[13] R. Sadourny. The Dynamics of Finite-Difference Models of the Shallow-Water Equations. *Journal Atm. Sci.*, 32(4), April 1975.

[14] T. von Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *19th International Symposium on Computer Architecture*, ACM Press, May 1992.

[15] S. Zeisset. Evaluation and Enhancement of the Paragon Multiprocessor's Shared Virtual Memory System. Master's thesis, TU München, 1993.