Compiling Applications with the KarHPFn* Compiler

Matthias M. Müller
Institute for Program Structures and Data Organization,
Universität Karlsruhe, Germany.
e-mail: muellerm@ira.uka.de
Technical Report No. 2000-11

April 26, 2000

Abstract

This paper compares the prefetching technique VSCAP (software controlled access pipelining with vector commands) with Cray T3E's highly optimized shared-memory functions (SHMEM) and with Portland Group HPF [BMN+97] (PGHPF) on three application benchmarks namely PDE1, FIRE, and Veltran. Previous work showed the good performance of VSCAP for single communication kernels. This paper examines VSCAP and practicability of KarHPFn, our prototype HPF compiler, in the context of whole applications. The results show that VSCAP generated by KarHPFn reduces communication overhead of fine-grained data parallel applications to a minimum. This leads to a performance gain compared to PGHPF between a factor of 2.8 for FIRE to a factor of 9.6 for PDE1. VSCAP programs are nearly as fast as SHMEM for regular communication patterns but 3.6 times faster than SHMEM in the case of dynamic communication patterns. All results were measured on 128 processors.

1 Introduction

As microprocessors get faster and the gap between computation and communication speeds widens, network latency becomes the dominant factor of the execution time of fine-grained parallel programs. Instead of a single communication operation a processor could perform hundreds to thousand arithmetic operations. This situation is even worse by an order of magnitude once the software overhead of communication libraries is taken into account.
Software controlled access pipelining with vector commands (VSCAP) overcomes these shortcommings by means of prefetching. The potential of VSCAP and its predecessor SCAP has been demonstrated by analytic models, simulations, and hand coded benchmarks [War97, MWT98]. In [Mül00a] we introduced KarHPFn, an optimizing HPF compiler which transforms data-parallel programs to programs using VSCAP for the communication. The benchmark set consisted of small communication kernels which emphasized the impact of fast communication. The results showed that VSCAP is as fast as T3E's shared-memory functions for regular communication patterns. But VSCAP programs are up to 6 times faster in the case of dynamic communication patterns. KarHPFn compiled programs are at least 30 times faster than programs compiled by the Portland Group HPF compiler.
Now, this paper extends the above comparison to applications which are not characterized by a single communication pattern. This comparison shows also the practicability of KarHPFn. The benchmark set consists of applications from geophysics (Veltran) and fluid dynamics (FIRE). A solver for partial differential equations (PDE1) is also presented. The tests were done on a Cray T3E using up to 128 processors.

*Karpfen (without *h*) is the german word for carp.

Prefetching is not new. Previous research addresses it in the context of prefetching cache lines, non-blocking loads, scheduling techniques, and speculative execution on uniprocessors or small-scale cache-coherent multiprocessors [CB92, RL92, MLG92, CKP91, GGV90]. Prefetching is also used by software distributed shared-memory systems to prefetch whole memory pages [LCD+97, BPA98].
But little is known about the effects of latency hiding applied to communication networks in massively parallel computers with distributed memory. And to our knowledge, KarHPFn is the first work targeting compiler directed prefetching for these architectures.
The next section describes the basic principle and the transformation rules of VSCAP which are incorporated in our KarHPFn compiler. Section 4 discurses the benchmark set. After that, the test environment is described. A presentation of our results terminates this paper.

2 VSCAP

In fine-grained parallel applications, as in most other parallel applications, latency prevents fast access to non-local memory. Software controlled access pipelining with vector commands (VSCAP) targets latency hiding through overlapping of computation and communication by splitting non-local memory access into low overhead prefetch and access. This section explains VSCAP by showing the basic concepts of SCAP, the predecessor of VSCAP without vector commands. The concepts of SCAP can be easily extended to VSCAP with additional vector commands for prefetch and access. A detailed explanation of SCAP and VSCAP can be found in [War97] and [Mül00b], respectively.

2.1 Basic Idea of SCAP and its extension to VSCAP

The aim of SCAP is a run-time improvement achieved by overlapping several communication requests leading to a communication pipeline in fine-grained data parallel applications. For a better understanding of the basic principle of SCAP, we first explain how communication is usually done.
The processor issues a request to the network (downwards arrow in Figure 1) and waits until the network replies (upwards arrow). Only then, the processor continues its execution and issues a new request. This is done as long as the processor requires remote data elements to perform its local part of computation. As the processor blocks after each data request, we call this kind of communication the *blocking mode*, see the upper half of Figure 1.
Now, let us assume the processor could issue all its communication requests and the network would be able to process them in an overlapped fashion. This would lead to a shorter waiting period for the processor accessing the first and all other successive remote data elements. Finally, communication could be performed faster compared to the above mentioned blocking execution. We call this kind of communication *overlapping communication*, see the lower half of Figure 1. To enable overlapping communication, the network interface has to provide a *prefetch buffer* that decouples the processor from the network execution, see Figure 2.
The second task of the prefetch buffer is to synchronize the processor with the network execution. The synchronization becomes necessary if the processor wants to access a data element which has not been delivered by the network yet. In this case, the processor is stalled until the value arrives. The waiting time in the lower part of Figure 1 denotes a processor stall. Figure 2 shows an overlapped execution of the processor and the network without processor stalls.
VSCAP extends SCAP by the means of vector commands for prefetch and access. Instead of issuing a communication requests for each non-local data element, the processor prefetches and accesses $L > 1$ data elements at once. $L$is the vector length of the vector commands. VSCAP's vector commands reduce prefetch and access overhead of SCAP and improve communication time further. But the usage of vector commands supposes regular communication patterns with equidistant offsets between successive data accesses. *Vector strategies* describe the possible combinations of vector and single element access.

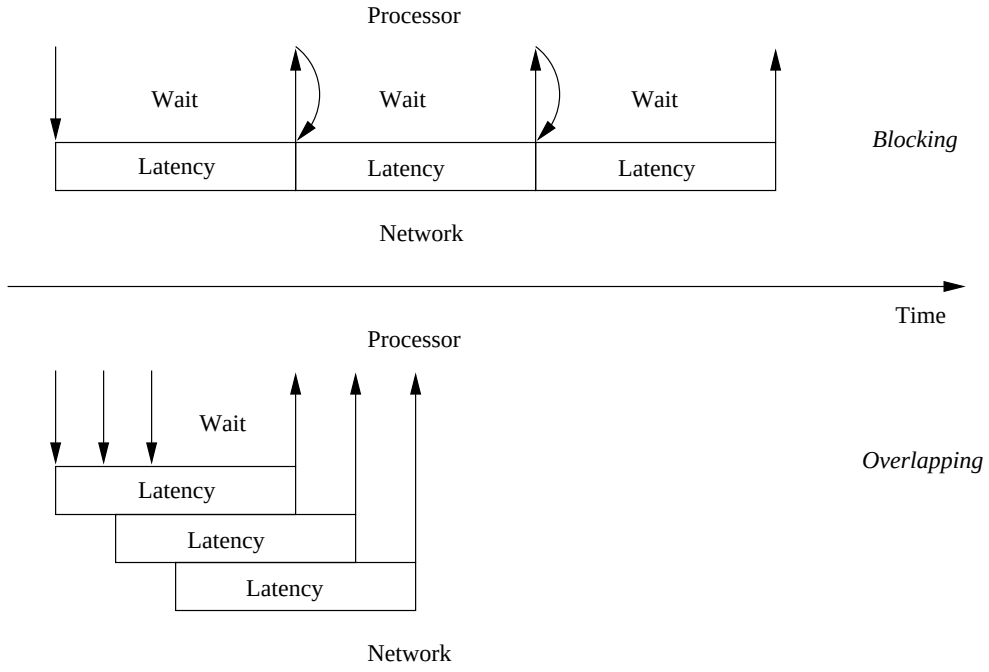

Figure 1: Basic idea of SCAP

2.2 Vector strategies

Vector commands for prefetching are only useful if displacements of the elements are equidistant and known at compile time. Otherwise, if element locality can be computed only at run time as in dynamic communication patterns or if distance of elements varies on a per-element basis, single-element prefetch instructions are used as a full block strategy. For this reason we introduce the notion of a *vector strategy* which declares the usage of vector operations.

(p, a)-vector strategy

A (p, a)-*vector strategy* declares usage of vector operations for prefetch ($p \geq 1$) and access ($a \geq 1$) operations. Assuming vector lengths $p, a \in \{1, L\}$ there are four possible vector strategies:

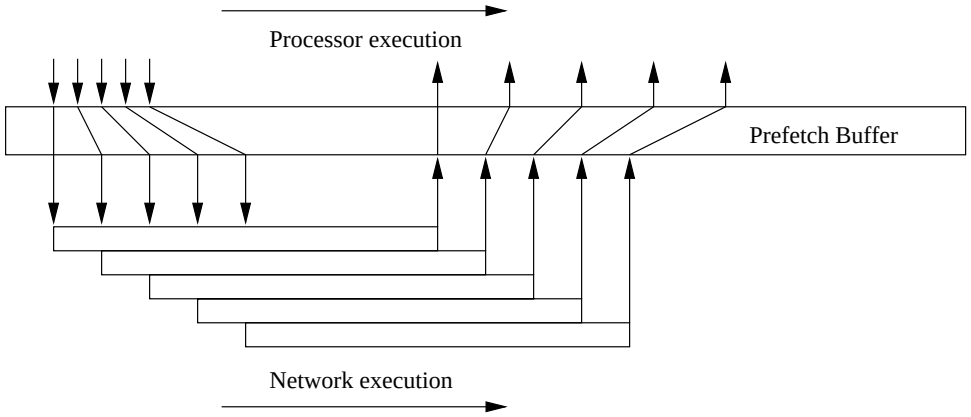

Figure 2: Usage of the Prefetch Buffer

Vector strategy	Explanation
(1,1)	Elementwise prefetch and access operations
(1,L)	Element operations for prefetch but vector access
(L,1)	Vector prefetch but elementwise access operation
(L,L)	Vector operations both for prefetch and access

The four vector strategies are characterized as follows:

$(1,1)$-**vector strategy** This is SCAP. It is used for communication patterns that do not allow any vector commands. They are characterized in varying element distances for prefetch and access, e.g. in masked assignments or in arbitrary block-cyclic distributions.

$(1,L)$-**vector strategy** Elementwise prefetching is done in dynamic communication patterns, e.g. in indirect indexed array access like the one shown in Section 2.3.

$(L,1)$-**vector strategy** This strategy uses vector prefetch and elementwise access operations as in scatter operations. As message passing architectures utilize remote write accesses already, this vector strategy is not investigated further.

(L,L)-**vector strategy** Vector operations can be used both for prefetch and for access if the location of data elements can be determined at compile time, e.g. in affine communication patterns $B(a * I + b)$, where a and b are variables that do not change their values during execution of communication.

This paper shows performance results of the (L,L) and $(1,1)$ vector strategies. Results of the $(1,L)$ vector strategy can be found in [Mül00a].
The $(L,1)$-vector strategy is not discussed further for the above reason. The remaining three strategies are selected automatically during the compilation. Their usage depends on the communication pattern, data-distribution, and the iteration space of the data-parallel forall. Section 3.2 describes the selection of the vector strategy according to the 3-dimensional parameter space.

2.3 Transformation rules

This paragraph describes the techniques used in the transformation from a data-parallel forall-statement to VSCAP. Section 3.2 describes the implementation of these techniques within the KarHPFn compiler.
The transformations are illustrated using the following simple forall-statement:

```
FORALL i = 0 TO N-1
  A(i) := B(q(i));
END
```

The program fragment updates array A in parallel, indexing array B with array q. A parallelizing compiler maps the problem size N onto P real processors ($N > P$). This technique is called *virtualization*. Assuming that P divides N each processor emulates $V = \frac{N}{P}$ virtual processors within a virtualization loop. Both A and B are distributed over the P processors using the the *owner-computes* rule. Since the value of $q(i)$ can not be determined at compile time, the compiler has to insert remote memory accesses. The virtualization of the program fragment is as follows, given the blocking execution mode:

```
FORALL j = 0 TO P-1                  // Forall processors in parallel
  FOR k = j*V TO (j+1)*V-1           // Simulate V virtual processors
    adr := calculate_address(B(q(k)));  // Calculate remote address
    A(k) := remote_read(adr);        // Read remote data element
  END
END
```

In the worst case, every processor issues V non-local memory accesses. These stall the processor if the network can not serve the desired values fast enough. Hence, execution time of this loop is at least V times the network latency.
The following transformation of the loop shows how communication and computation can be overlapped:

```
FORALL j = 0 TO P-1
  FOR k=j*V TO (j+1)*V-1              // Prefetch loop
    adr := calculate_address(B(q(k)));  // Calculate remote address
    prefetch(adr);                    // Start read request
  END
  FOR k=j*V TO (j+1)*V-1              // Access loop
    adr := calculate_address(B(q(k)));
    A(k) := access(adr);              // Access data element
  END
END
```

In this transformation, the main loop is split into two instances: a prefetch and an access (or calculation) loop. Instead of stalling on a remote memory access as in blocking mode, the processor issues remote memory prefetch requests. After the prefetch loop is executed the calculation loop accesses non-local memory without waiting time (if the data is already present) in the prefetch buffer. This is the code for SCAP or the $(1,1)$-vector strategy. In the best case, program speed-up is about $(V-1)$ times the network latency because there is at most one waiting period (arrival of first data item) compared to V waiting times in a blocking network.
VSCAP improves the above code with vector access commands further. While vector prefetch operations cannot be used due to the dynamic prefetch pattern caused by the index array q vector accesses are possible because of the regular array access $A(k)$. This leads to a $(1, L)$-vector strategy:

```
FORALL j = 0 TO P-1
  FOR k=j*V TO (j+1)*V-1              // Prefetch loop
    adr := calculate_address(B(q(k)));  // Calculate remote address
    prefetch(adr);                    // Start read request
  END
  FOR k=j*V TO (j+1)*V-1 STEP L       // Vector access loop
    adr := calculate_address(B(q(k)));
    vector_access(adr,A(k));          // Access L data elements
  END
END
```

For brevity, we assume that L divides V. Otherwise, additional elementwise access operations has to be used to get the remaining V *mod* L data elements which do not fill a vector of length L. The access loop is blocked with blocksize L. Within the loop, `vector_access(adr,A(k))` copies L entries from the prefetch buffer starting at address `adr` to successive memory locations beginning with `A(k)`. If we assume that a vector access lasts as long as an elementwise access operation, the duration of the vector access loop is decreased about the factor L of the vector length.
If the number of non-local memory accesses is too large to fit into the prefetch buffer, VSCAP's transformation rule uses a three loop execution pattern where the middle loop alternates between access and prefetch instructions. This transformation is not shown for the sake of brevity.

3 KarHPFn

This section describes the architecture and the compilation steps of the Karlsruher HPF compiler *KarHPFn*.

3.1 Overview

KarHPFn is a source-to-source compiler to transform a data parallel HPF program into an executable Fortran 90 node program that uses only Cray T3E's E-register operations for communication, see Figure 3. KarHPFn's program transformations concentrate on the forall-statement.

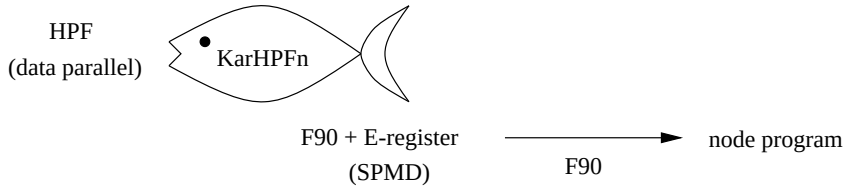

Figure 3: Overview of KarHPFn

KarHPFn is based on the ADAPTOR front-end [Bra94]. All subsequent analysis and transformation phases use the Cocktail-Toolbox [GE90] to operate on the abstract syntax tree built by that front-end. Dependence and partitioning analysis phases use common techniques to perform their tasks.

3.2 Transformation within KarHPFn

KarHPFn transforms a given source-program in four steps. First, data distribution information is evaluated and the computation is spread among processors using the *owner-computes* rule. The second step analyses subscripts of the arrays involved to determine the communication pattern. The appropriate pattern is selected using the pattern matching technique from [Boz95]. In the example of Section 2.3, KarHPFn determines the tuple $(i, q(i))$. KarHPFn assumes that all data accesses are remote because the value of q cannot be computed at compile time. As each reference to $B(q(i))$ is treaten as a non-local array access (although some elements could be local) we call this *speculative prefetch*.
In the third step, KarHPFn determines the vector-strategy. It uses the subscript information $(i, q(i))$ and the data-distribution, e.g. a blockwise distribution, and looks up the corresponding table for a suitable vector-strategy, see Table 1.

Table 1: Selection of vector-strategies

Comm. pattern	**Data-distribution**	**Vector-strategy**
$(i,b)^*$, $(i,i+c)^\dagger$, $(i,i+b)$	BLOCK	(L,L)
	CYCLIC	(L,L)
	CYCLIC(k)	(L,L), $(1,1)$
$(i, a*i+b)$	BLOCK	(L,L)
	CYCLIC	$(1,L)$
	CYCLIC(k)	$(1,L)$, $(1,1)$
$(i,q(i))^\ddagger$	BLOCK	$(1,L)$
	CYCLIC	$(1,L)$
	CYCLIC(k)	$(1,L)$, $(1,1)$

$^*a$ and b are arbitrary variables
$^\dagger c$ is a constant
$^\ddagger q$ is a function or an array

The lines for the CYCLIC(k) distribution have an additional $(1,1)$ entry for possible vector strategies. This is caused by the fact that some combinations of iteration space and blocksize k force a calculation of the local index set at run-time. As these index sets do not have equidistants

offsets between consecutive data elements each data element has to be prefetched and accessed separately. The switch to a $(1,1)$-vector strategy is done automatically.
Returning to our example, KarHPFn selects with the tuple $((i,q(i))$,BLOCK) the $(1,L)$-vector strategy.
The fourth and final step generates the appropriate pipeline for communication and inserts it into the final program.
As KarHPFn supports four different communication modes (BLOCK, SCAP, VSCAP, and SHMEM[1]) and three different data distributions (BLOCK, CYCLIC, and CYCLIC(k) distributions) there are 12 possible pipelines for the $(1,L)$-vector strategy. Furthermore, KarHPFn can apply two different optimizations to this kind of vector strategy. The first one reduces address calculation overhead as it uses capabilities of the *hardware centrifuge* of the Cray T3E. The second one introduces a *software test* to determine locality of data accesses. This is useful if only non-local data-elements have to be prefetched. These two optimizations can be applied to three of the above four communication modes for all possible data-distributions, this leads to additional $9+9=18$ pipelines.
Therefore, KarHPFn can generate $12+18=30$ different communication pipelines for the $(1,L)$-vector strategy which shows the need for a sophisticated software architecture for the pipeline generation module.

4 Benchmarks

The purpose of our benchmark set is to show the effects of fast communication in the context of an entire application instead of the focus on a single and small part of the program. Execution on more than hundred processors indicates performance in a massively parallel environment.
We chose three applications from three different domains: *PDE1*, a solver for partial differential equations, *FIRE*, a fluid dynamics application, and *Veltran* from geophysics. Characteristics of these benchmarks are explained below.

4.1 PDE1

PDE1 is a 3-dimensional Poisson solver using red-black relaxation. The 3-dimensional grid is divided into red and black points. In an iteration first, red points are calculated using values of the six adjacent black points. In the second step of the iteration, black points are determined using the new red values.
PDE1 splits the entire 3d-grid into smaller cubes that are distributed among processors. Before computation takes place, each processor has to read the border of its local cube from virtually neighbouring processors. This so called *nearest-neighborhood* communication pattern results in a linear communication compared to a cubic computation time. Therefore, we expect a small advantage from fast communication for problem sizes with large local cubes because calculation time dominates communication. But as the number of processors increase and as the virtualization and computation time decrease, the usage of fast communication results in a further speed up.
Due to the regular communication pattern, KarHPFn uses a (L,L)-vector strategy for VSCAP. Remote data is read into overlap areas of local cubes.

4.2 FIRE

FIRE is a fluid dynamics package from AVL List using the method of conjugate gradients on unstructured meshes [BSCG96]. The main communication loop in *FIRE* gathers cells from `DIREC1` indexed through `LCC`:

[1] shared-memory library

```
FORALL J=1 TO 6
  FORALL NC=1 TO NNINTC
    IF LCCMASK(J,NC) THEN
      DIREC1V(J,NC) := DIREC1(LCC(J,NC));
    END
  END
END
```

FIRE distributes the problem domain in blocks. It has an high proportion of communication to compuation leading to a large communication overhead regardless of local problem sizes.
Without the if-statement in the above program fragment KarHPFn would use a $(1, L)$-vector strategy but predication with `LCCMASK` forces a $(1, 1)$-vector strategy for *FIRE* because prefetch and access operations depend on run-time information.

4.3 Veltran

Veltran is an application from geophysics that uses velocity analysis to calculate consistency of earth layers [JPK98]. It uses the method of conjugate gradients.
Communication in *Veltran* is a scatter operation of local parts of two distributed dimensions of the problem domain. If we use a concurrent-read memory model, the communication reduces to a read of the remote data elements. The read access reduces the communication overhead for every data element from $O(log(P))$ as in message passing architectures to $O(1)$, where P denotes the number of processors. Nevertheless, an equal amount of communication and computaion leads to a high communication overhead for all virtualizations.
Two dimensions of the 3-dimensional global problem are distributed in blocks over processors. KarHPFn uses a (L, L)-vector strategy for the scatter operation.

5 Test environment

This section presents a short overview of the Cray T3E and the investigated different benchmark versions.

5.1 Architecture of Cray T3E

The T3E 900/512 used for our measurements consists of up to 2048 DEC Alpha 21164 processors running at 450 MHz. They are connected with a 3-dimensional torus network. The net is decoupled from the processors at a speed of 75 MHz [ST96] with overlapped communication. Each link has a bandwidth of approximately 500 MB/s resulting in a 3 GB/s transfer rate for a single node.
The network interface consists of 512 user and 128 system E-registers, memory mapped into the I/O-space of each processor. E-registers provide the only means to transfer data between processors. Read and write operations between E-registers and the global memory are called *gets* and *puts*. To load a global memory content into the processor, a *get* and a subsequent read of the E-register has to be executed. The latter operation stalls the processor until the value arrives. This synchronization is implemented in hardware using the readiness state of the E-register. On a *put*, the memory of a remote node is modified and the cache is updated [Oed96]. Hence, the T3E implements a *global address space* with locally consistent memory.
Eight E-registers can be combined to a vector. Distance between successive vector elements have to be equidistant to ensure correct address translation. Thus, the T3E enables VSCAP with a vector length of $L = 8$.

5.2 Benchmark versions

Benchmarks were compiled to four different versions:

BLOCK simulates blocking communication of the T3E as described in Section 2.1.

VSCAP does prefetch and access with vector operations. 16 E-register vectors (of 8 E-registers each) are used to allow a total number of 128 outstanding communication requests. 128 E-registers suffice to hide network latency and to get maximum throughput [Sco96].

SHMEM uses Cray's shared-memory system functions for communication. SHMEM delivers maximum communication performance on the T3E for regular communication patterns. SHMEM behaves like BLOCK in dynamic patterns because of lack of support by the system library. Hence, each element has to be read by a separate function call to *shmem_get* resulting in blocking performace.

PGHPF represents the executables of the Portland Group HPF compiler. PGHPF is the commercial HPF compiler available for the Cray T3E.

The first three version were compiled by KarHPFn using different command line options while PGHPF compiled the fourth one. Both compilers got the same HPF-source. Standard optimizations were turned on for both PGHPF and for the Fortran 90 compilation step of KarHPFn. Time was measured with the real-time clock (RTC). Problem size was kept constant for each benchmark but the number of processors was increased from 2 to 128 in powers of 2.

6 Results

The runtimes of the four different versions for each benchmark are given. The discussion of each benchmark includes two plots. The first one shows the runtimes, and the second one presents the relative performance compared to BLOCK. Since we focus on the reduction in communication time, the plots do not account for speed up compared to a one processor execution.

6.1 PDE1

A global cube with $N = 128^3$ data points formed the base of the measurements of *PDE1*. Figure 4 presents runtimes and relative performance of the four different versions of *PDE1*.

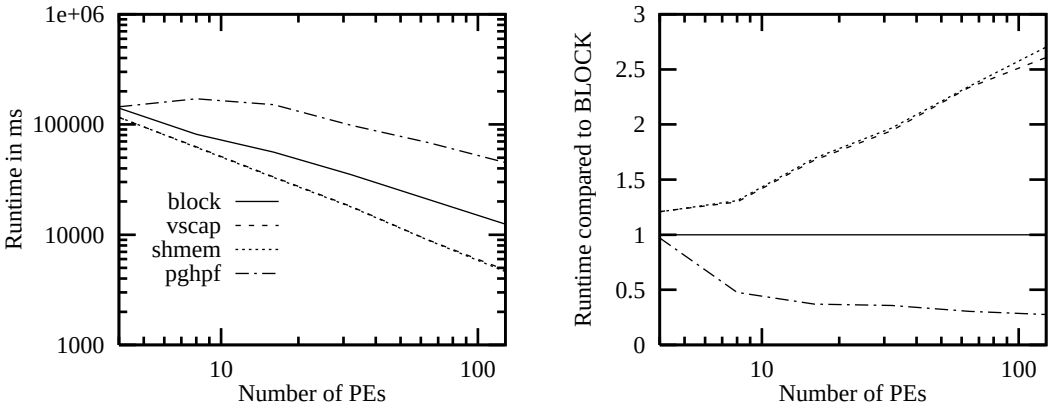

Figure 4: Runtimes and speed up relative to BLOCK of *PDE1*

The runtime plot on the left displays three different lines. The upper one denotes runtime of PGHPF. The solid line in the middle is execution time of BLOCK, our base model, and the lower line presents runtime of VSCAP and SHMEM. The plot leads to three different observations. First, fast communication improves performance only for smaller local problemsizes, e.g. at execution on more than 10 processors. This behaviour is caused by the fact that with an increasing number of processors less computation is needed and therefore, the relative advantage of better communication grows. Second, VSCAP is as fast as highly optimized shared-memory functions, and third,

PGHPF is slower than reading one remote data element after the other. The plot on the right hand side quantifies the last two observations. VSCAP and SHMEM are more than 2.5 times faster than BLOCK on 128 processors while PGHPF is about 3.7 times slower. The comparison of VSCAP with SHMEM and PGHPF shows that VSCAP is only 4% slower than SHMEM but 9.4 times faster than PGHPF on 128 processors.

6.2 FIRE

FIRE was calculated with 318045 cells. Thus, virtualization variied during the measurements from 159023 for 2 processors to 2485 cells for 128 processors. Figure 5 shows the execution time and the relative performance of the *FIRE* versions.

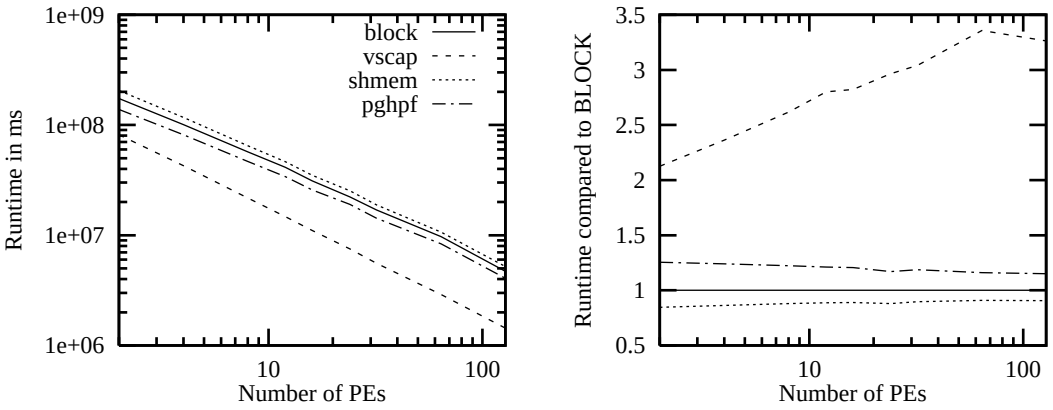

Figure 5: Runtimes and speed up relative to BLOCK of *FIRE*

The runtime plot shows two different groups. SHMEM, BLOCK and PGHPF belong to the upper (slower) group and VSCAP to the fast one. The plot shows a constant difference in runtime caused by the constant ratio of communication to computation time throughout all virtualizations. The plot on the right displays relative performance compared to BLOCK. PGHPF is 30% faster, SHMEM is 15% slower than BLOCK. Only VSCAP shows with a substantial speed up compared to BLOCK (factor 3.4 on 64 processors). VSCAP is also 3.6 times faster than SHMEM and 2.8 times faster than PGHPF on 128 processors.
The poor performance of SHMEM is due to the fact that the shared-memory library does not support any kind of dynamic communication pattern. Though, each non-local element has to be read with a seperate call to *shmem_get* resulting in a blocking execution. The reason for the PGHPF behaviour is the usage of *inspector-executor* model that increases communication overhead by additional execution time for the communication schedule. VSCAP is not affected by this overhead because time for prefetch is limited to predicate evaluation and to issue the prefetch.

6.3 Veltran

Measurements of *Veltran* were done using a global problem size of 3240000 points. Figure 6 show the results.
The runtime plot on the left shows two different lines. The upper line contains BLOCK and PGHPF while VSCAP and SHMEM belong to the lower one. The constant advantage in runtime accounts for the constant ratio of communication and computation. Relative performances are shown in the plot of the right hand side of Figure 6. PGHPF is almost as fast as BLOCK ($\pm 5\%$) while VSCAP and SHMEM are at least 2.2 times faster. Relative runtime gain of the latter two grows with the increasing number of processors used in parallel. It reaches its climax at 128 processors where VSCAP is 4.6 and SHMEM 4.7 times faster than BLOCK and PGHPF. VSCAP is only 4% slower than SHMEM.

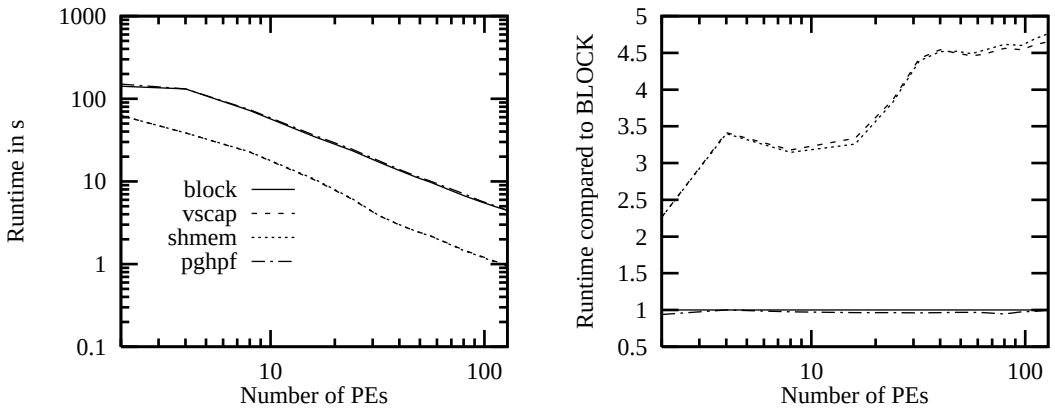

Figure 6: Runtimes and speed up relative to BLOCK of *Veltran*

The plot on the right hand side shows waves in the speed up of VSCAP and SHMEM. These waves are caused by the 2-dimensional virtual processor grid that was changed between successive measurements to take the increasing number of processors into account. Different from expectation, however, this change did not lead to a constant improvement.

7 Conclusions

This paper examined the performance of VSCAP on applications (PDE1, FIRE, and Veltran) and showed the practicability of our HPF compiler KarHPFn. We compared VSCAP both with the highly optimized shared-memory library and with the Portland Group HPF compiler. Test programs were generated automatically by KarHPFn operating on the same HPF sources as PGHPF.

On regular communication patterns KarHPFn's VSCAP is nearly as fast as the system functions ($\pm 4\%$). On dynamic communication patterns (FIRE), however, VSCAP is 3.6 times faster due to lack of support of these patterns by the system library.

A comparison of VSCAP to PGHPF shows KarHPFn's strength: a programmer gets a 9.6 times faster program (PDE1) just by exchanging PGHPF with KarHPFn without the need for additional knowledge on communication techniques. These results were not limited to small problems as the large problem sizes and the measurements on 128 processors show.

Further work concentrates on VSCAP for workstation clusters broadening the range of suitable architectures. The SCI standard seems to be a promising candidate to achieve this goal. Further questions in this context include the behavior of VSCAP on dynamic communication patterns and the mixture of VSCAP's remote read with remote write access.

References

[BMN+97] Z. Bozkus, L. Meadows, S. Nakamoto, V. Schuster, and M. Young. PGHPF – an optimizing High Performance Fortran compiler for distributed memory machines. *Scientific Programming*, 6(1):29–40, Spring 1997.

[Boz95] Zeki Bozkus. *Compiling Fortran 90D/HPF for Distributed Memory MIMD Computers.* PhD thesis, Syracuse University, June 1995.

[BPA98] Ricardo Bianchini, Raquel Pinto, and Claudio L. Amorium. Data Prefetching for Software DSMs. In *12th International Conference on Supercomputing, Melbourne*, pages 385–392, July 13–17, 1998.

[Bra94] Thomas Brandes. Adaptor: A compilation system for data parallel fortran programs. Technical report, German National Center for Computer Science (GMD), St. Augustin, Germany, 1994. ftp://ftp.gmd.de/GMD/adaptor/docs/adaptor.ps.

[BSCG96] P. Brezany, V. Sipkova, B. Chapman, and R. Greimel. Automatic parallelization of the AVL FIRE benchmark for a distributed-memory system. *Lecture Notes in Computer Science*, 1041:50–60, 1996.

[CB92] Tien-Fu Chen and Jean-Loup Baer. Reducing memory latency via non-blocking and prefetching caches. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 51–61, Boston, Massachusetts, October 1992. Also available as U. Washington CS TR 92-06-03.

[CKP91] David Callahan, Ken Kennedy, and Allan Porterfield. Software prefetching. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, Santa Clara, California, April 1991.

[GE90] Josef Grosch and Helmut Emmelmann. A tool box for compiler construction. In Dieter Hammer, editor, *Compiler Compilers, Third International Workshop on Compiler Construction*, volume 477 of *Lecture Notes in Computer Science*, pages 106–116, Schwerin, Germany, 22–26 October 1990. Springer, 1991.

[GGV90] Edward H. Gornish, Elana D. Granston, and Alexander V. Veidenbaum. Compiler-directed data prefetching in multiprocessor with memory hierarchies. In *Proceedings 1990 International Conference on Supercomputing*, pages 354–368, Amsterdam, June11–15 1990.

[JPK98] Matthias Jacob, Michael Philippsen, and Martin Karrenbach. Large-scale parallel geophysical algorithms in Java: a feasibility study. *Concurrency: Practice and Experience*, 10(11–13):1143–1153, September 1998. Special Issue: Java for High-performance Network Computing.

[LCD+97] Honghui Lu, Alan L. Cox, Sandhya Dwarkadas, Ramakrishnan Rajamony, and Willy Zwaenepoel. Compiler and Software Distributed Shared Memory Support for Irregular Applications. *ACM SIGPLAN Notices*, 32(7):48–56, July 1997.

[MLG92] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, Boston, Massachusetts, October 1992.

[Mül00a] Matthias M. Müller. KaHPF: Compiler generated Data Prefetching for HPF. In *High Performance Computing in Science and Engineering 1999*, pages 474–482. Springer, 2000.

[Mül00b] Matthias M. Müller. *Latenzzeitverbergung in datenparallelen Sprachen.* PhD thesis, School of Computer Science, Universität Karlsruhe, February 2000.

[MWT98] Matthias M. Müller, Thomas M. Warschko, and Walter F. Tichy. Prefetching on the Cray-T3E. In *12th International Conference on Supercomputing*, pages 368–375, Melbourne, July 13–17, 1998.

[Oed96] Wilfried Oed. Massiv-paralleles Prozessorsystem CRAY T3E. Technical report, Cray Research GmbH, München, November 1996.

[RL92] Anne Rogers and Kai Li. Software support for speculative loads. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 38–50, Boston, Massachusetts, October 1992.

[Sco96] Steven L. Scott. Synchronization and communication in the T3E multiprocessor. *ACM SIGPLAN Notices*, 31(9):26–36, September 1996.

[ST96] Steven L. Scott and Gregory M. Thorson. The Cray T3E network: Adaptive routing in a high performance 3D torus. *HOT Interconnects IV*, August 15–16 1996.

[War97] Thomas M. Warschko. *Effiziente Kommunikation in Parallelrechnerarchitekturen*. PhD thesis, School of Computer Science, Universität Karlsruhe, 1997.