About the Performance of HPF: Improving Runtime on the Cray T3E with Hardware Specific Properties

Matthias M. Müller
Fakultät für Informatik
Universität Karlsruhe
e-mail: muellerm@ipd.uka.de

Abstract

High Performance Fortran permits to write parallel programs with much less programming effort than by using standard communication libraries such as MPI or PVM. The performance of compiled HPF programs is considered low, though. We show that a compiled HPF application will gain a substantial runtime improvement if compilation incorporates properties of the hardware architecture into the final program.

Our prototype HPF compiler "KarHPFn"[1] inserts communication primitives of the Cray T3E into the target programs. Programs compiled with KarHPFn run up to 30 times faster than their counterparts compiled with Portland Group HPF.

1 Introduction

From the early nineties on, a lot of research effort has been spent to develop a parallel programming environment which eases the implementation of parallel Fortran applications. High Performance Fortran (HPF) aims to support natural scientists in their effort to write parallel solutions for their ever growing computational problems. The resulting HPF standard supports the data-parallel programming paradigm, provides lots of features for data-distribution, and permits to write parallel programs with much less programming effort than with standard communication libraries, such as MPI or PVM. But HPF is not in widespread use because of its low runtime performance. Natural scientists develop their proof of concept in HPF as a reference implementation to which they can compare their production implementation written with MPI or PVM.

The HPF implementation of Portland Group PGI on the Cray T3E is no exception to this rule. It is mainly used for teaching purposes whereas the real number-cruncher applications are still written using standard communication libraries. So far, the PGI HPF implementation has missed its target of freeing the programmer from the burden of explicitly parallelizing applications when writing the code. The PGI implementation of HPF relies on a large and architecture independent communication library. From an economics points of view, an architecture independent library is reasonable because it eases the portability of HPF across different platforms. On the other side, this approach results in a compiler that nobody uses because of the communication overhead this library incurs on the parallel runtime. This situation is dissatisfying for both interest-groups of the parallel architecture: the suppliers and the developers. A supplier of a parallel architecture with a sophisticated network interface is faced with the situation that his parallel architecture achieves best hardware performance figures but this performance is devoured by the communication overhead of the programming environment. And the programmer of a parallel platform has only two choices. He can parallelize an application by hand which is complicated and error prone but which results in a fast parallel program, or, he can use HPF which is easy to program but which results in a bad runtime performance. Thus, the main drawback of common HPF compilers is that they don't support available hardware properties which can decrease communication overhead and thus, increase the runtime performance of the resulting executables compared to hand written code.

This paper suggests the following approach: instead of relying on large platform-independent communication libraries, specific hardware properties should be incorporated into the compiled program to deliver to the application all the network performance the parallel architecture supplies. Our Karlsruhe HPF compiler "KarHPFn" is a prototype compiler to demonstrate the feasibility of this approach. KarHPFn inserts communication primitives of the Cray T3E into the target programs.

[1] *Karpfen* [ˈkarpfən] without 'h' is the german word for carp.

In this paper, we present a comparison of the performance of KarHPFn compiled programs with PGI HPF on the Cray T3E. We studied 25 benchmarks from a wide range of common algorithm classes. The result is quite promising. All KarHPFn compiled programs outperformed their PGI HPF counterparts. The KarHPFn programs achieve a performance improvement of up to a factor of 30 for reasonable local problem sizes where each processor acts on at least 8 local data-elements. For smaller local problem sizes the performance improvement is up to three orders of magnitude higher, overall. Measurements on 128 processors on the Cray T3E also showed that KarHPFn compiled programs achieve a speed-up compared to a single-processor execution between 64 and 76. The PGI HPF compiled versions reach only speed-ups of 7 to 33.

KarHPFn uses *software controlled access pipelining with vector commands* VSCAP for communication. VSCAP hides network latency not only by computation but also by issuing prefetch requests to the network. If all the prefetch and local computation is done, VSCAP tries to access the first non-local data-element. If the network latency is shorter than the time spent for prefetch and local computation, VSCAP accesses the first non-local data-element without any delay. Thus, the total communication overhead reduces to the time spent for the prefetch instructions. If prefetch and access is done on a per-element basis, we call this kind of communication SCAP [14]. VSCAP reduces the prefetch overhead further by using vector commands which issue L prefetch requests at once. Earlier work about VSCAP reported hand coded results [11], first experiences with KarHPFn [9], and an extensive evaluation of KarHPFn [10].

The paper is organized as follows. The next sections describes the communication technique used by KarHPFn. Section 3 describes the different communication pipelines which KarHPFn inserts into the target programs. Section 4 presents an overview about KarHPFn and its architecture. The benchmark set is shown in section 5 and the experimental results are discussed in section 6. Conclusions are drawn in section 7.

2 KarHPFn's communication model

This section explains the communication technique VSCAP used by KarHPFn.

2.1 Basic Requirement: Overlapping Communication

The aim of VSCAP is to improve runtime of a program by overlapping several communication requests leading to a communication pipeline between prefetch and access instructions.

Blocking Communication

For a better understanding of the basic principle of VSCAP, it is first explained how communication is done usually. The processor issues a request to the network (downwards-arrows in figure 1) and waits until the network replies (upwards-arrows). 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.

Overlapping Communication

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 mode. 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 processor on the upper half of figure 2 issues all its communication requests as prefetch instructions (downwards-arrows). Each prefetch reserves an entry in the prefetch buffer (solid box in the middle). An entry invokes (dashed downwards-arrows) network execution. The network reads the desired remote data-element and writes it back in the reserved entry of the prefetch buffer (dashed upwards-arrows). The network execution-time for each remote read operation (thin solid boxes) is the network latency. After the processor has issued all its communication requests, it accesses the remote data elements from the prefetch buffer (upwards-arrows). The prefetch buffer decouples processor from network execution as it temporarily buffers the data-elements written by the network until the proces-

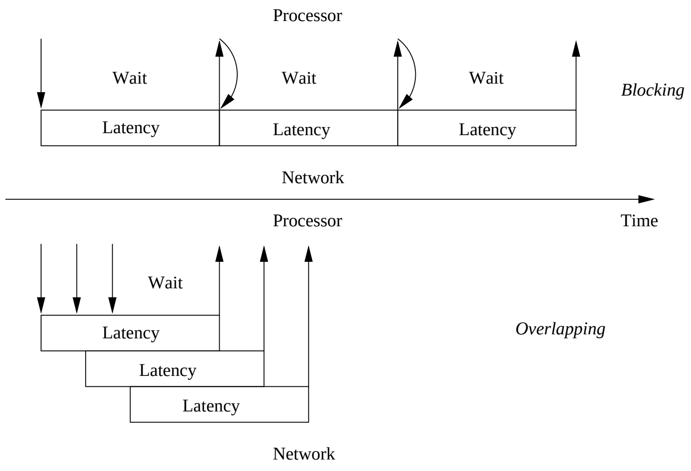


Figure 1: Blocking vs. Overlapping Communication.

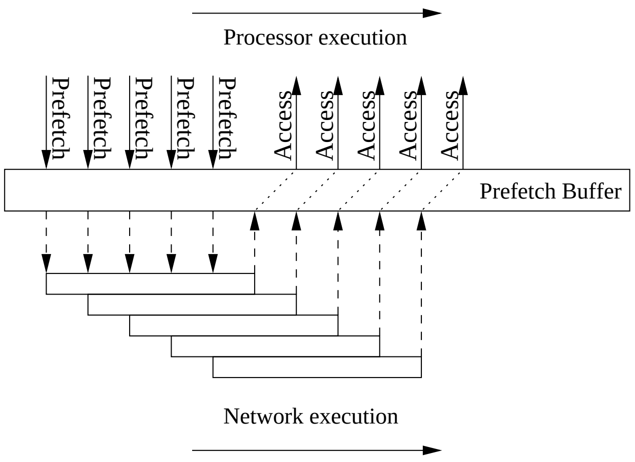


Figure 2: Usage of the Prefetch Buffer.

sor accesses them (buffer-time is shown with dotted lines). The other 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 that 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.

In summary, the processor forms a pipeline of prefetch and access operations to and from the prefetch buffer. As the prefetch and access operations are software controlled, we call this technique *software controlled access pipelining* or *SCAP*, for brevity. This sophisticated interaction between software, processor, prefetch buffer, and network was developed by Warschko [14].
VSCAP augments the above overlapping communication with vector commands for prefetch and access. Instead of issuing a communication requests for each single non-local data-element, the processor can prefetch and access $L > 1$ 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.

2.2 Transformation Rules

This paragraph describes the techniques used in the transformation from a data-parallel forall-statement to VSCAP. 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 permutation 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 *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 processors in parallel
FORALL j = 0 TO P-1
  // Simulate V virtual processors
  FOR k = j*V TO (j+1)*V-1
    // Calculate remote address
    addr := calculate_address(B[q(k)]);
    // Read remote data-element
    A[k] := remote_read(addr);
  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
  // Prefetch loop
  FOR k=j*V TO (j+1)*V-1
    // Calculate remote address
    addr := calculate_address(B[q(k)]);
    // Start read request
    prefetch(addr);
  END
  // Access loop
  FOR k=j*V TO (j+1)*V-1
    addr := calculate_address(B[q(k)]);
    // Access data-element
    A[k] := access(addr);
  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 the blocking mode, the processor issues remote memory prefetch requests. After the prefetch loop is executed, the calculation loop accesses non-local data-elements without waiting time (if the data is already present) from the prefetch buffer. This is the code for SCAP. Ideally, 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. Vector prefetch operations cannot be used due to the dynamic prefetch pattern caused by the permutation q. Therefore, only vector accesses are possible because of the regular array access $A[k]$.

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

For brevity, we assume that L divides V. Otherwise, additional element wise access operations would have to be used to get the remaining $V \textit{ mod } L$ data-elements that do not fill a vector of length L. The access loop is blocked with block size L. Within the loop, `vector_access(A[k],inc,addr)` copies L entries from the prefetch buffer starting at address `addr` to the memory locations denoted by $A[k] + i * inc$, for $0 \leq i < L$.

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 shown in [10].

3 Types of communication pipelines

3.1 Vector Strategies

In principal, vector commands for prefetching are only useful if displacements of the elements are equidistant and known at compile-time. Otherwise, if element addresses can be computed only at runtime as in dynamic communication patterns or if the distances of elements vary on a per-element basis, single-element prefetch instructions are used throughout the whole prefetch loop. For this reason we introduce the notion of a *vector strategy.*

(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 fixed vector lengths $p, a \in \{1, L\}$ there are four possible vector strategies:

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

The following three sections describe the first three vector strategies. The (L,1)-vector strategy is an exception compared to the other strategies. It would be applied in assignments such as `A[q(i)] := B[i]` where the left hand side index $q(i)$ could denote non-local memory addresses. But this leads, however, to remote write operations which spread the data-elements across all processing elements. This vector strategy is not considered further, because the overlapping of several remote write operations are already done and well understood in message passing architectures. As the above proposed overlapping network model can be augmented naturally by remote write operations, message passing techniques can be applied.

3.2 (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. Overall, there are three different applications of the (L, L)-vector strategy in KarHPFn: the one-block and multi-block pipelines and reductions.

3.2.1 One-block pipeline

A one-block pipeline reads remote data-elements from only one remote network node. It is used if static analysis shows that processors need data-elements from only one remote network node to perform their part of the global computation. For example, the one-block pipeline is mainly used in n-dimensional nearest neighborhood applications (AC7, see 5). Figure 3 shows the resulting communication pattern of the one-block pipeline for one of the processors.

3.2.2 Multi-block pipeline

Multi-block pipelines are mainly used to cope with remote data-elements for whom a processor has to access several remote network nodes. The multi-block pipeline consists of several one-block pipelines to collect the data from the respective nodes. Affine communication patterns $B[a * I + b]$, where a and b are variables that do not change their values during the execution of the communication are the main application of the multi-block pipeline. Figure 4 shows the resulting communication pattern of the multi-block pipeline.

3.2.3 Reductions

During reductions, each processor initially calculates its local part of the global result. The local P results are then combined to the global result of the reduction. The collection of local results is performed in $log_2(P)$ steps. And finally, each processor accesses the global result and loads it into its local memory. Figure 5 depicts the associated communication schedule.

The initial proposal of SCAP [14] suggests an increased fan-in for the reduction to obtain more remote data-accesses to hide network latency with. Measurements on the T3E showed a fan-in of $f = 16$ to be optimal. However, measurements with larger values for f lead to impractical results such that the value of $f = 16$ was used throughout the measurements.

The first $log_f(P)$ communication steps are performed with one-block pipelines. But now, not the array index is varied but the processor number.

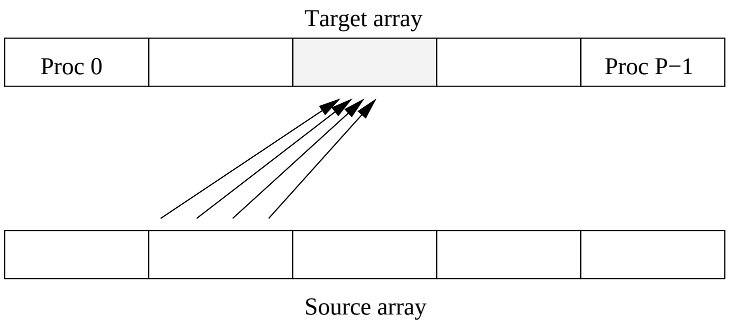


Figure 3: Communication pattern for one-block pipeline.

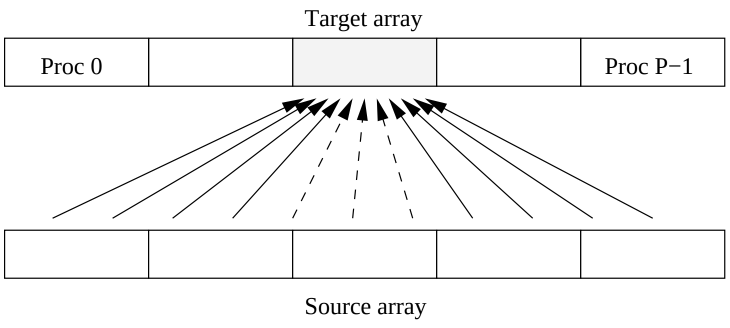


Figure 4: Communication pattern for multi-block pipeline.

3.3 $(1, L)$-vector strategy

Element wise prefetching is done in dynamic communication patterns, e.g. in indirect indexed array access like the one shown in section 2.2. Figure 6 shows the resulting communication pattern.

3.4 $(1, 1)$-vector strategy

This strategy implements SCAP. The $(1, 1)$-vector strategy is used for communication patterns that do not allow any vector commands. They are characterized in varying element distances for prefetch and access which can not be computed at compile-time, e.g. in arbitrary block-cyclic distributions. Figure 6 depicts the communication pattern again, but now, the displacements of the elements in the target array are not equidistant anymore.

4 KarHPFn

This section presents an overview of the prototype HPF compiler of Karlsruhe KarHPFn, explains the structure of the resulting executables, and summarizes supported HPF language features.

4.1 Overview

KarHPFn is a source-to-source compiler transforming a data parallel HPF program into an executable Fortran 90 node program that uses e-register operations for communication [12]. Its program transformations focus on the forall-statement.

KarHPFn is based on the ADAPTOR [2] front-end developed by Brandes at the German National Center for Computer Science (GMD). All subsequent analysis and transformation phases operate on the abstract syntax tree built by the front-end. The dependence and partitioning analysis phases use common techniques to perform their tasks. A detailed description of KarHPFn's transformation steps can be found in [10].

4.2 Structure of node program

In contrast to the Portland Group HPF compiler KarHPFn does not rely on a large communication library. Though, the final Fortran 90 node program is linked together with two libraries, both libraries are rather small and contain only initialization code and optimized versions of the one-block and multi-block

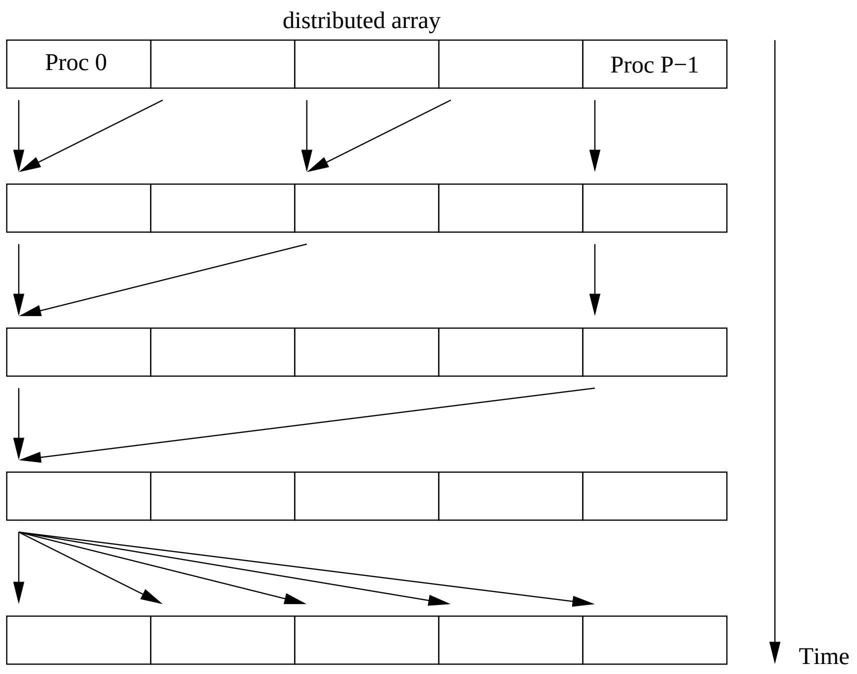
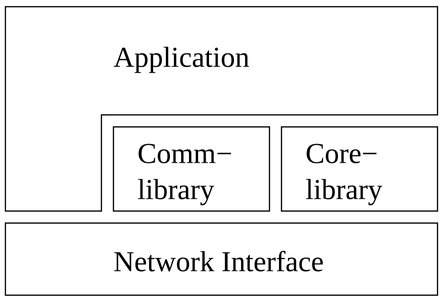


Figure 5: Communication schedule for reduction with a fan-in of 2.

pipelines. The following figure depicts the structure of a node program.



Application This part of the program contains all application specific code such as calculation loops and specific communication pipelines, e.g. pipelines for the $(1, L)$-vector strategy.

Comm-library The communication library consists of application independent implementations of the one-block and multi-block pipelines.

Core-library The core library contains one method to initialize the e-registers. It is called once at startup.

Network interface The network interface is built up of e-registers which lie in the I/O space of a local processor.

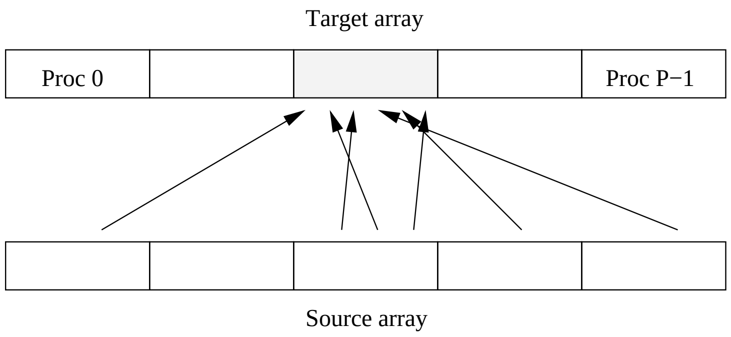


Figure 6: Communication pattern for $(1, L)$- and $(1, 1)$-vector strategy.

4.3 Supported language features

KarHPFn implements most features of *Subset*-HPF [5]. The only restriction of the standard affects the *align*-directive as affine alignments are not supported. Additionally, the Fortran 90 reduction functions XXX_PREFIX, XXX_SCATTER, XXX_SUFFIX are also not implemented. Where necessary the author inserted these functions manually but tried to reproduce the effect of an automatic compilation.

5 Benchmarks and their Implementation

The set of benchmarks is categorized into algorithm classes (AC). These algorithm classes are presented first. The benchmarks are described subsequently.

5.1 Algorithm Classes

The distinctive properties of each AC are the relationship of communication (T_{Comm}) to computation time (T_{Calc}) and the data access pattern. Table 1 gives an overview. The columns present the different amount of communication while the rows distinguish communication patterns. Each entry presents the AC, the benchmarks within this class, and the vector strategy used by the compilation. The ACs are characterized as follows.

AC1: Reductions are a common operation in parallel applications. The reduction in AC1 is structured in such a way that each processor performs a local reduction and then collects the results from other processors. Thus, the amount of communication is merely depending on the number of processors and small compared to computation.

AC2: This reduction is not implemented as efficiently as the one of AC1. Rather than a local reduction each array element collects results from $log(N)$ neighbors, where N is the size of the array. Thus, communication grows with the size of the array.

AC3: This AC deals with indexed arrays. The index offsets are constants.

AC4: The index offsets in these indexed arrays are arbitrary integer variables or the data-distribution involves a lot of communication.

AC6: This AC represents dynamic communication caused by indexing. The indexes are computed dynamically from other data.

AC7: This AC deals with blocked nD-grids in which communication is limited to the border of the blocks.

AC9: In this kind of scatter operations, local data has to be spread to a subset of the processing elements.

AC10: These scatter operations spread data to *all* processing elements.

AC5 and AC8 are empty. For AC5, the author did not find an appropriate benchmark and for AC8, there exists no benchmark because the proportion of computation to communication is always high in nD-grid applications.

5.2 Benchmarks

Despite the empty ACs, the benchmark set represents a wide range of common parallel applications. Most of the benchmarks are data-parallel versions of the Livermore loop kernels (*LL*). For their description and parallelization see [4, 14]. *Rotate* implements a cyclic shift of an array where a cyclic distribution leads to the large amount of communication. *Indirect* is the example kernel from section 2.2. *Fire* is a fluid dynamics package from AVL List using the method of conjugate gradients on unstructured meshes [3]. *Jacobi* and *Laplace* perform successive over-relaxation on a 2D-grid. *PDE1* is a 3D-grid Poisson solver using red-black relaxation. *Veltran* is an application from geophysics that uses velocity analysis to calculate density of earth layers [7]. *Veltran* uses the method of conjugate gradients.
Most of the benchmarks contain only one of the access patterns of Table 1. The benchmarks containing more than one access pattern are classified into the AC that dominates its runtime behavior. Thus, *LL13*, *LL14*, and *LL23* are classified into AC2; *LL6* into AC4; *Laplace* into AC7; and *Fire* into AC6.
The benchmarks were compiled to the following versions.

KarHPFn 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 [12].

Table 1: Data access patterns and the associated ACs.

Access Pattern	$T_{\text{Comm}} < T_{\text{Calc}}$	$T_{\text{Comm}} \sim T_{\text{Calc}}$
Reduction	AC1: LL3, LL4, LL24 (L,L)	AC2: LL2, LL5, LL11, LL13, LL14, LL19, LL23 (L,L)
Indexed Arrays	AC3: LL1, LL7, LL8 LL12, LL15, LL18 (L,L)	AC4: LL6, Rotate (L,L)
Indirect Indexed Arrays	AC5: empty	AC6: Indirect, Fire (1,L)
nD-grid	AC7: Jacobi, Laplace, PDE1 (L,L)	AC8: empty
Scatter	AC9: LL21 (L,L)	AC10: Veltran (L,L)

PGI HPF represents the executables of the Portland Group HPF compiler [1]. PGI HPF is the commercial HPF compiler available for the Cray T3E.

Both compilers got the same HPF-source. Standard optimizations were turned on for both PGI HPF and for the Fortran 90 compilation step of KarHPFn. Time was measured with the real-time clock (*RTC*). Except for *PDE1*, *Fire*, and *Veltran*, runtime was measured for different problem sizes while the number of processors was kept constant. The remaining three benchmarks were measured with a fixed problem size and the number of processors was varied from 2 to 128. All other benchmarks were measured on 32 processors except for *LL1*, *LL7*, *LL13*, *LL21*, *Jacobi* and *Laplace* which were ran on 64 processors.

6 Results

The runtimes of the KarHPFn and PGI HPF executables are compared. The results of each algorithm class is presented in a different section. Each section contains one or two plots, depending on the number of the benchmarks within the algorithm class. The plots show for every benchmark the relative runtime improvement factors of the KarHPFn executables compared to the PGI HPF executables. The x-axis of a plot shows the virtualization while the y-axis presents the comparison to PGI HPF. Numbers larger than one on the y-axis mean a runtime improvement and numbers smaller than one a runtime loss compared to PGI HPF. Therefore, the PGI HPF version of each benchmark would have a straight solid line at one (this line is not shown in the plots).
The following sections discuss all but one of the algorithm classes of section 5. The discussion of AC10 is omitted as the only member is shown in 6.8 where the number of processors is varied and not the problem size.

6.1 Algorithm Class AC1

Algorithm class AC1 contains benchmarks which mainly use reductions. For example, benchmark *LL3* calculates the inner product of two vectors. The number of non-local data-accesses is $log_f(P) + 1$ for each processor. Figure 7 shows the relative runtime improvement factors.
The KarHPFn versions of the benchmarks achieve only for small to medium sized virtualizations ($V \leq 1000$) a noteworthy runtime improvement compared to PGI HPF. This is caused by the constant com-

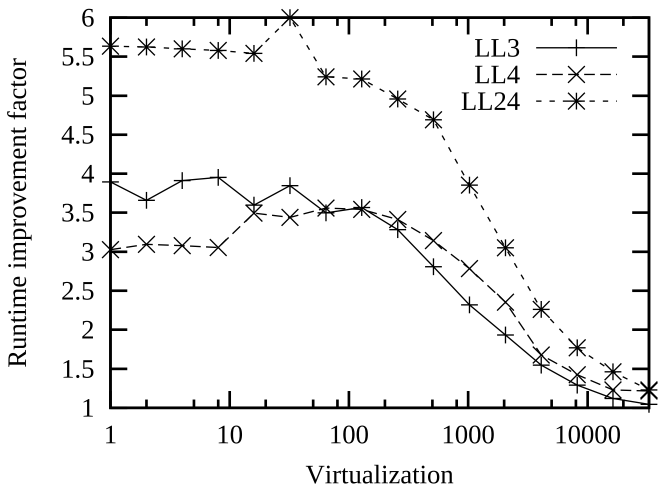


Figure 7: Runtime improvement for AC1.

munication overhead which is large compared to the small amount of calculation for small virtualizations and which almost diminishes for large virtualizations. *LL24* achieves the highest runtime improvement factor of 6 at a virtualization of $V = 16$.

6.2 Algorithm Class AC2

Benchmarks in AC2 are dominated by reductions whose communication overhead increases with the problem size. Due to the large amount of non-local data-accesses, KarHPFn programs can hide network latency more efficient than with benchmarks of AC1. Figure 8 depicts the runtime improvement factors of the different benchmarks.

The left plot shows the runtimes of the programs generated completely by KarHPFn, whilst the compilation of the benchmarks in the right plot involved some manual transformation, because *LL13* and *LL14* use the HPF SUM_SCATTER function which is yet not supported by KarHPFn, see 4.3.

The KarHPFn programs in the left plot achieve best relative performance at medium sized virtualizations ($100 \leq V \leq 10000$). These benchmark version are up to 8 to 24 times faster than their counterparts compiled with PGI HPF.

6.3 Algorithm Class AC3

Communication of benchmarks in AC3 are characterized by indexed array accesses. For example, *LL1* is a kernel which has at most 11 non-local data-accesses if arrays are distributed in a block-wise fashion. As the amount of communication is fixed for these benchmarks, the relative performance improvement of the KarHPFn compiled versions compared to the PGI HPF versions diminishes as virtualization is increased. Figure 9 shows the results.

As for AC1, KarHPFn programs in the left plot achieve only for small virtualizations ($V \leq 100$) a substantial runtime improvement compared to their PGI HPF counterparts. This is caused by the small proportion of communication to computation which almost diminishes for large virtualizations. The two benchmarks on the right side of figure 9 show a slightly different result. Whilst *LL7* shows almost the same behavior as the benchmarks on the left plot, *LL1* starts with a relatively small runtime improvement factor and increases to a factor of 10. *LL7* starts with an improvement factor of 11 and diminished to a factor of about 9.

LL7 achieves the best relative runtime improvement, overall. At a local-problem size of 1 local data-element, *LL7* is 1750 times faster than the PGI HPF version. *LL18* shows a quite similar behavior at a virtualization of $V = 1$, as it is 377 times faster than its PGI HPF counterpart.

6.4 Algorithm Class AC4

Benchmarks of AC4 also possess an indexed array access. But now, the amount of communication is much higher as compared to AC3 due to a different access pattern or a different data-distribution. For example, *Rotate* implements a cyclic shift of an array which is distributed in a cyclic fashion. Figure 10 depicts the result of both benchmarks of AC4.

The ascending runtime improvement factor of both benchmarks is a result of the linear increasing

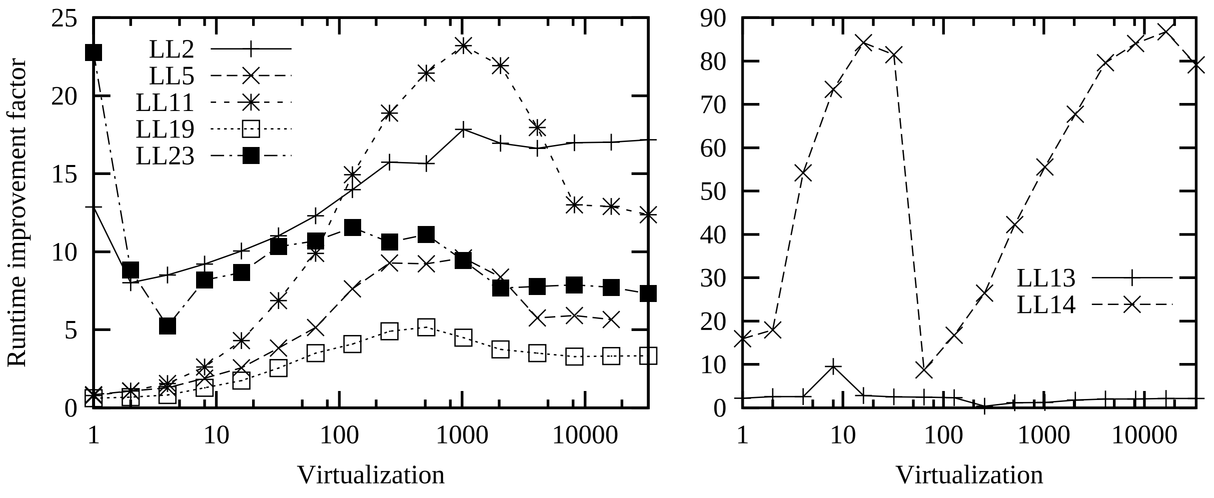


Figure 8: Runtime improvement for AC2.

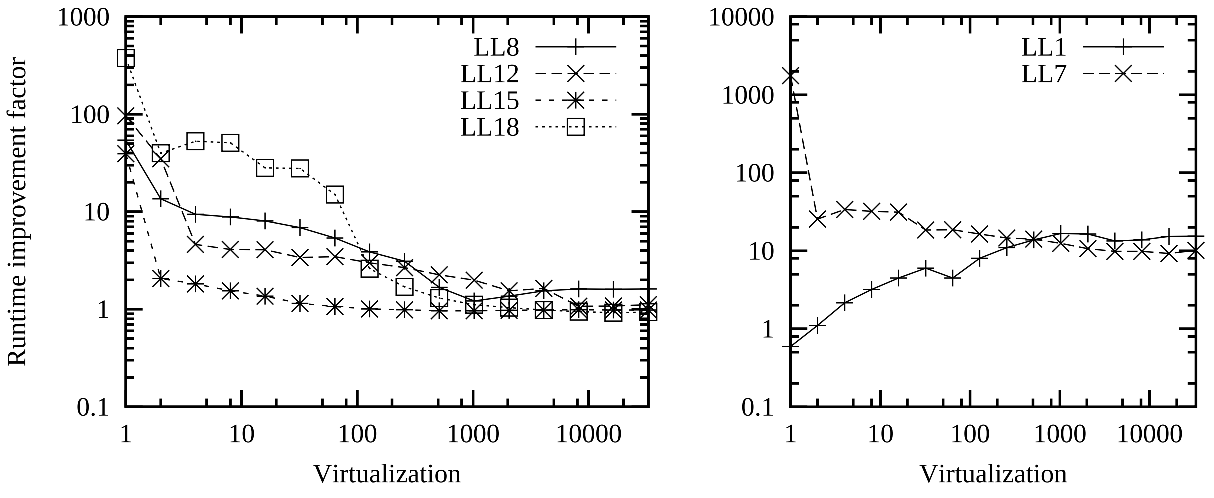


Figure 9: Runtime improvement for AC3.

amount of communication for growing virtualizations. The KarHPFn compiled version of Rotate achieves a maximum runtime improvement factor of 30 compared to the PGI HPF version.

6.5 Algorithm Class AC6

Algorithm class AC6 specifies algorithms which contain indirect indexed array accesses. The main characteristics of these array accesses are that they generally can not be computed at compile time. PGI HPF uses the *inspector-executor* technique [8, 13] to deal with this kind of irregular communication pattern, while KarHPFn inserts its normal prefetch instructions which can also access data-elements in local memory. This kind of KarHPFn's generated data-access is called *speculative* prefetch as the target of a prefetch operation does not necessarily have to be remote. The left plot in figure 11 shows only the results of *Indirect*. The results for the other benchmark *Fire* is presented in 6.8. *Indirect* is the running example of 2.2.

The runtime improvement factor of *Indirect* increases for larger virtualization. It reaches its maximum with 4.8 at $V = 2048$. Relative performance is not as high as for AC4 because now, KarHPFn inserts pipelines with a $(1, L)$-vector strategy. Hence, *Indirect* uses fast e-register vector operations only for the access part of the pipeline. Prefetching is done with element-wise operations.

6.6 Algorithm Class AC7

AC7 contains benchmarks using nearest neighborhood communication in a block-wise distributed nD-grid. For example, *Jacobi* calculates for each entry

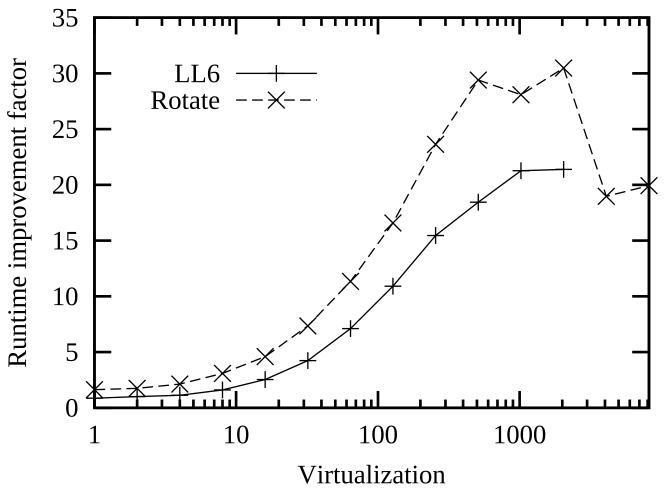
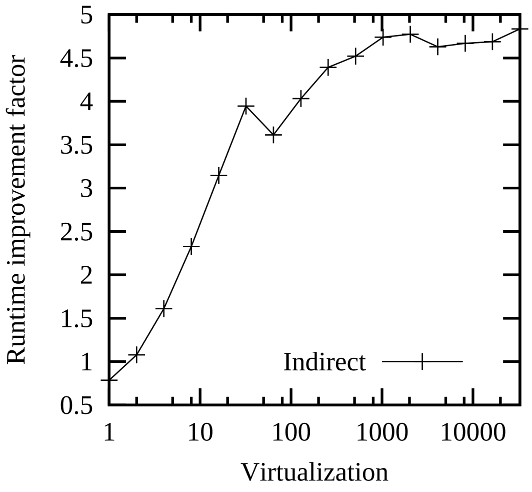


Figure 10: Runtime improvement for AC4.



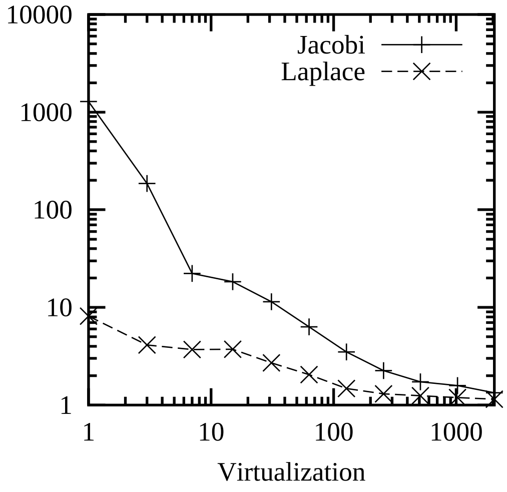


Figure 11: Runtime improvement for AC6 (left) and AC7 (right).

in the grid the arithmetic mean of its four neighbors. The right plot in figure 11 depicts the results of *Jacobi* and *Laplace*. The results of *PDE1* are shown in 6.8. The virtualization on the x-axis denotes the size of one dimension of the local quadratic matrices.

The execution of both benchmarks is characterized by two steps. During the first step, the communicational step, adjacent remote field regions are copied into so called *shadow regions* [6]. The second step involves computation on the augmented local fields. As the amount of computation is quadratic compared to the linear amount of communication, the communicational part of the benchmarks decreases as the problem sizes increases. Both benchmarks show a decrease in the runtime improvement factor for increasing virtualizations. For example, *Jacobi* reaches a runtime improvement of 11 with local matrices of size 7×7. This improvement almost diminishes for virtualizations of 2047×2047.
For small virtualizations $V \leq 4$ *Jacobi* shows a relative runtime improvement factor of up to 1282 which is comparable to the behavior of *LL7* and *LL18* of AC3.

6.7 Algorithm Class AC9

Execution of AC9 involves scatter operations in which data-elements have to be distributed from one node to a subset S of all nodes. In message-passing architectures, scatter operations are normally performed using $log(|S|)$ communication steps to reduce the amount of communication overhead. In

contrast, VSCAP makes this reduction obsolete because each processor prefetches its part from remote nodes. Though, the VSCAP way of doing scatter operations increases the pressure on the network interface from whom data is scattered, we measured no decrease in network bandwidth while scattering data this way. The only benchmark in AC9 *LL21* implements a matrix multiplication using the *matmul*-function. Figure 12 shows the result for *LL21*. The virtualization on the x-axis denotes one dimension of the local quadratic matrices.
As the proportion of communication to computation decreases *LL21* shows quite the same runtime improvement factor behavior as the benchmarks in AC7.

6.8 Variation of number of processors

For the remaining three benchmarks (*PDE1*, *Fire*, and *Veltran*) the number of processors was varied while the problem size was kept constant. These tests aim to show the scalability of VSCAP on up to 128 processors. The x-axis of figure 13 accounts for the varying number of processors.
Two facts can be seen from figure 13. First, all three KarHPFn versions run faster than their PGI HPF counterparts, and second, the runtime improvement factor increases as the number of processor increases. The last statement can also be put another way: the less the virtualization the faster the KarHPFn versions compared to the PGI HPF versions. The *Fire* (*PDE1*, *Veltran*) version of KarHPFn is at its best 2.9 (9.5, 4.7) times faster than the PGI HPF version.
Compared to a single-processor execution of the benchmarks, the six programs achieve the following speed-ups on 128 processors (these figures can not be deduced from figure 13).

Program	Version	Speed-up on 128 procs.
Fire	KarHPFn	79
	PGI HPF	33
PDE1	KarHPFn	64
	PGI HPF	7
Veltran	KarHPFn	76
	PGI HPF	15

7 Conclusions

This study presented a comparison of Portland Group's HPF compiler and the Karlsruhe HPF compiler KarHPFn. KarHPFn uses VSCAP for communication whose communication pipelines are inserted directly into the program. The executable program uses communication features of the Cray T3E with only minimal overhead and hence, gains substantial performance improvements compared to executables generated by the PGI HPF compiler.
The comparison of KarHPFn and PGI HPF was done on a set of 25 benchmarks from a wide range of common algorithm classes. The KarHPFn compiled programs run up to 30 times faster than their PGI HPF counterparts for reasonable virtualizations of $V \geq 8$. For small virtualizations KarHPFn programs are up to 1700 times faster than their PGI HPF counterparts.
Compared to a single-processor execution, measurements on 128 processors showed a speed-up of the KarHPFn generated programs ranging between 64 and 79 while the PGI HPF executables reached numbers from 7 to 33.
KarHPFn is a research prototype and far away from being a commercial product. Nevertheless, the performance of its generated programs should be a motivation for all interested in HPF to decrease execution times of HPF such that is usable for up to day problems in scientific computing.
The web page `http://www.ipd.uka.de/KarHPFn` provides further information about KarHPFn and some additional optimization techniques for the T3E.

References

[1] 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.

[2] T. 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.

[3] P. Brezany, V. Sipkova, B. Chapman, and R. Greimel. Automatic parallelization of the AVL FIRE benchmark for a distributed-memory system. In J. Dongarra, K. Madsen, and J. Wasniewsky, editors, *Applied parallel computing: computations in physics, chemistry, and engineering science: second international workshop, PARA '95*, volume 1041 of *Lecture Notes in Computer Science*, pages 50–60, Lyngby, August 1996. Springer.

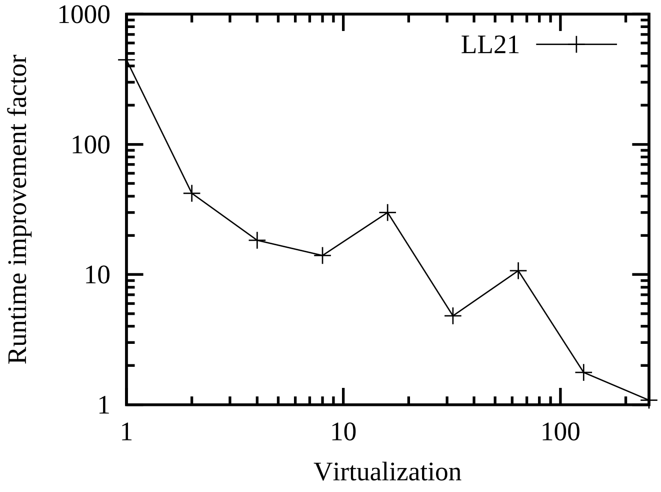


Figure 12: Runtime improvement for AC9.

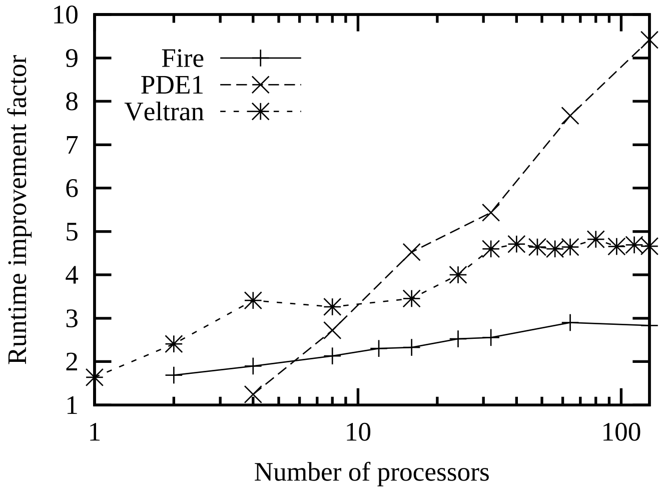


Figure 13: Speed-up of PDE1, Fire, and Veltran.

[4] J. Feo. An analysis of the computational and parallel complexity of the Livermore loops. *Parallel Computing*, 7(2):163–185, June 1988.

[5] High Performance Fortran Forum. *High Performance Fortran Language Specification 1.1*, November 1994.

[6] High Performance Fortran Forum. *High Performance Fortran Language Specification 2.0*, January 1997.

[7] M. Jacob, M. Philippsen, and M. 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.

[8] Charles Koelbel and Piyush Mehrotra. Supporting shared data structures on distributed memory architectures. In *2nd ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming, PPOPP*, pages 177–186, March 1990.

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

[10] M. Müller. Compiler-generated vector-based prefetching on architectures with distributed memory. In *High Performance Computing in Science and Engineering 2001*. Springer, 2001.

[11] M. Müller, T. Warschko, and W. Tichy. Prefetching on the Cray-T3E. In *Twelfth International Conference on Supercomputing*, pages 368–375, Melbourne, July 1998.

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

[13] R. v. Hanxleden, K. Kennedy, C. Koelbel, R. Das, and J. Saltz. Compiler analysis for irregular problems in Fortran-D. In *5th Workshop on Languages and Compilers for Parallel Computing*, 1992.

[14] T. Warschko. *Effiziente Kommunikation in Parallelrechnerarchitekturen.* PhD thesis, Fakultät für Informatik, Universität Karlsruhe, December 1997.