Scientific Programming, v6, n1, Spring, 1997, pp. 59–72

Performance Issues in HPF Implementations of Sensor–Based Applications

David R. O'Hallaron, Jon Webb, Jaspal Subhlok

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Applications that get their inputs from sensors are an important and often overlooked application domain for High Performance Fortran (HPF). Such *sensor–based applications* typically perform regular operations on dense arrays, and often have latency and throughput requirements that can only be achieved with parallel machines. This paper describes a study of sensor–based applications, including the fast Fourier transform, synthetic aperture radar imaging, narrowband tracking radar processing, multibaseline stereo imaging, and medical magnetic resonance imaging. The applications are written in a dialect of HPF developed at Carnegie Mellon, and are compiled by the Fx compiler for the Intel Paragon. The main results of the study are (1) that it is possible to realize good performance for realistic sensor–based applications written in HPF, and (2) the performance of the applications is determined by the performance of three core operations: independent loops (i.e., loops with no dependences between iterations), reductions, and index permutations. The paper discusses the implications for HPF implementations and introduces some simple tests that implementors and users can use to measure the efficiency of the loops, reductions, and index permutations generated by an HPF compiler.

1. Introduction

There is an important class of computer applications that manipulate inputs from the physical environment. The inputs are continuously collected by one or more sensors and then passed on to the computer, where they are manipulated and interpreted. The sensors are devices like cameras, antennas, and microphones. The manipulation of the sensor inputs is variously referred to as signal processing or image processing, depending on the dimensionality of the inputs. We refer to the entire class of applications as *sensor–based applications* to emphasize this common quality of processing inputs from the natural world.

Sensor–based applications have traditionally been found in military domains like radar and sonar, and there is an increasing interest in commercial domains, such as medical imaging, surveillance, and real–world modeling. For example, a real–world modeling application could use a stereo algorithm to acquire depth information from multiple cameras and then use the information to build realistic 3D models of the environment. The models could then be used for applications like virtual 3D conferencing, building walk–throughs, or experiencing a sporting event from the point of view of one of the players.

This research was sponsored in part by the Advanced Research Projects Agency/CSTO under two contracts: one monitored by SPAWAR (contract number N00039-93-C-0152), the other monitored by Hanscom Air Force Base (contract number F19628-93-C-0171), in part by the Air Force Office of Scientific Research under contract F49620–92–J–0131, in part by the National Science Foundation under Grant ASC-9318163, and in part by grants from the Intel Corporation. Authors' email addresses: droh@cs.cmu.edu, webb+@cmu.edu, jass@cs.cmu.edu.

Sensor–based applications are an interesting and often overlooked application domain for a parallel language like HPF [7]. The computations, which typically consist of regular operations on dense arrays, are naturally expressed in HPF. Furthermore, there are often stringent latency and bandwidth requirements that demand parallel processing. For example, a stereo program that extracts depth information from multiple cameras can process only a few frames per second on a powerful RISC workstation, which is well below the standard video rate of 30 frames per second. If the results of a sensor–based computation are used to control some process, then there is also be a minimal latency that can be tolerated. For example, an online medical imaging application that gathers and processes multiple images might automatically adjust the scanner to compensate for movement by the patient. The importance of minimizing latency, rather than just maximizing throughput, is one of the key properties that distinguishes sensor–based applications from batch–oriented scientific computations [21].

This paper describes the results of an empirical study of the performance of a set of HPF sensor–based applications on a commercial parallel computer. The results were obtained using a prototype compiler, developed at Carnegie Mellon, for a dialect of HPF running on an Intel Paragon. The paper makes several main points: First, contrary to the fears of many in the HPF community, performance for the HPF applications we studied is good. Second, a few core computational patterns (parallel DO loops, reductions, and index permutations) dominate sensor–based applications. HPF implementors can realize great benefits by focusing on these patterns. Third, there are some simple tests that HPF programmers and developers can use to evaluate the efficiency of the parallel DO loops, reductions, and index permutations that are crucial to the effective execution of sensor–based applications. Fourth, since the data sets in sensor–based applications are often fixed by properties of the sensors, scalability is an important issue. Finally, the same patterns that appear in sensor–based computations also appear in scientific applications. In particular, we will briefly examine an Fx regional air quality modeling code and an Fx earthquake ground motion modeling based on the method of boundary elements.

In Section 2 we give a brief overview of the prototype HPF compiler (the Fx compiler) that was used in the study. Section 3 describes the applications that we implemented in Fx and their performance on the Intel Paragon. Sections 4, 5, and 6 describe some key issues in generating efficient code for HPF DO loops, reductions, and permutations, and introduce some simple tests for measuring the efficiency of these operations. Section 7 discusses the issue of scalability in sensor–based applications. Finally, Section 8 shows how the same DO loops, reductions, and permutations that are crucial to sensor–based also appear in scientific computations.

2. Fx overview

The Fx project was started in Fall 1991 with the goal of learning how to generate efficient code for programs written in the emerging HPF standard.[1] The input language is a dialect of subset HPF and consists of F77 with HPF data layout statements, array assignment statements with support for general CYCLIC(k) distributions in an arbitrary number of array dimensions [16, 17], an index permutation intrinsic, and a parallel DO loop that is integrated with arbitrary user–defined associative reduction operators [22]. Fx also provides a mechanism for mixing task and data parallelism in the same program [6, 20, 19]. The initial target was the Intel iWarp. Fx was later ported to the IBM SP/2, the Intel Paragon, and workstation clusters.

Much of the early work on Fx was driven by the 2D fast Fourier transform (FFT) and algorithms for

[1] Although the first meeting of the HPF Forum was not until January 1992, preproposals from Rice, Vienna, and ICASE were already circulating during Summer 1991, so the general form of the HPF programming model was already clear by Fall 1991.

global image processing. The system was generating inefficient versions of the FFT by Spring 1992 and efficient versions of the FFT by Fall 1992. By Spring 1993 task and data parallelism were integrated into the compiler, and by Fall 1993 arbitrary user–defined reductions were integrated into the parallel DO loop. The first significant Fx applications were written in Summer 1994: a multibaseline stereo program (from scratch) and a spotlight synthetic aperture radar program (from F77 code supplied by Sandia Labs). By Spring 1995 there were a number of additional Fx applications, including a stereo front-end for a real–world modeling application developed by the second author (from scratch), a magnetic resonance image reconstruction program (from scratch), an earthquake ground motion modeling program (from F77 code developed at Southern California Earthquake Center), and a regional air quality modeling program (from F77 code derived from the CIT airshed model).

3. Sensor–based applications

Sensor–based applications typically have the high–level two–stage pipelined structure shown in Figure 1. The front–end accepts a stream of inputs from one or more sensors and manipulates these inputs into some

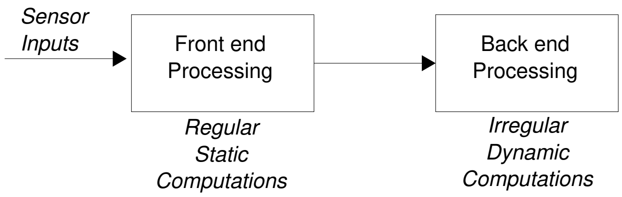


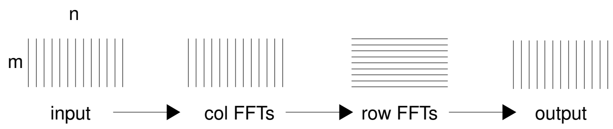
Figure 1: The structure of sensor–based applications.

desired form. The back–end interprets the results of the front–end and either displays them or initiates some action. For example, in a radar tracking application, the front–end might transform input phase histories from an antenna array into an image in the spatial domain, and the back–end would manipulate this image to name, identify and track objects of interest.

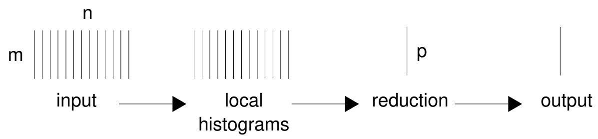
The front–end processing typically consists of numerous, regular, data parallel operations on dense arrays, requires high MFLOPS rates, and the operations performed are usually data–independent. Computations such as the fast Fourier transform (FFT), convolution, scaling, thresholding, data reduction, and histogramming are common operations. The back–end processing is typically more dynamic, irregular, and data–dependent, with real–time scheduling of processes. In this paper, we are concerned with the front–end processing, where HPF on a parallel system is most appropriate. For the remainder of the paper, when we refer to sensor–based applications we are referring to the front–end.

One of the nice qualities of sensor–based computing is that many applications have similar computational patterns. The similarities allow us to focus on a few small application kernels, with the assurance that anything that we learn about compiling these small programs will accrue benefits in larger, more realistic programs. Two examples that capture many of the key computational patterns, and were of tremendous help in the development of the Fx compiler, are the 2D FFT (FFT2) and the image histogram (HIST). The high–level parallel structure of these computations is shown in Figure 2.

Figure 2 depicts the course–grained parallelism that is available in FFT2 and HIST. The vertical lines depict independent operations on array columns and the horizontal lines depict independent operations on rows. The FFT2 program is collection of independent local 1D FFTs on the columns of an array, followed



(a) FFT2 – 2D fast Fourier transform



(b) HIST – Image histogram

Figure 2: Sensor–based computation exemplars

by a collection of independent local 1D FFTs on the rows. If these columns or rows are distributed across the processors, then each column or row operation can run independently. In HPF, the FFT2 example can be written as:

```
COMPLEX a(N,N),b(N,N)
!HPF$ DISTRIBUTE (*,BLOCK):: a,b

!HPF$ INDEPENDENT
DO k=1,N
 call fft(a(:,k))
ENDDO
b = TRANSPOSE(a)
!HPF$ INDEPENDENT
DO k=1,N
 call fft(b(:,k))
ENDDO
a = TRANSPOSE(b)
```

The Fortran 90 TRANSPOSE intrinsic is important because it allows each 1D FFT to operate locally and in place on a contiguous vector. The same effect could also be achieved with the Fortran 90 RESHAPE intrinsic, using the ORDER argument to swap the array indices [1]:

```
b = RESHAPE(a, (/N,N/), (/0/), (/2,1/))
```

An alternative is to distribute one array by columns and the other array by rows, and then redistribute from columns to rows using an array assignment statement. This approach results in rows that are stored locally but not contiguously. Since it would require extra copies to get the rows in contiguous form, this approach will not be considered here.

The HIST example consists of a collection of independent local histograms on the columns of an integer array, followed by a plus–reduction operation that adds the local histogram vectors to form the final result. This might be written in HPF as:

```
INTEGER a(N,N),h(M,N),r(M)
!HPF$ DISTRIBUTE (*,BLOCK):: a

h = 0
!HPF$ INDEPENDENT
DO j=1,N
 DO i=1,N
  h(i,j) = h(a(i,j),j) + 1
 ENDDO
ENDDO
r = SUM(h,2)
```

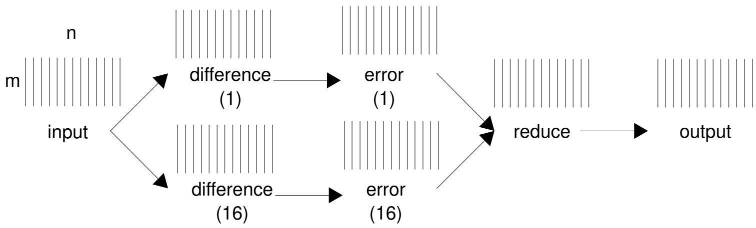
The FFT2 and HIST examples capture the core computational patterns in sensor–based computing: parallel DO loops, reductions, and index permutations. FFT2 is a pair of parallel DO loops, each of which is followed by an index permutation (the TRANSPOSE intrinsic). HIST is a parallel DO loop followed by a reduction. These two patterns occur repeatedly in the sensor–based applications we have studied.

Figure 3 shows a collection of sensor–based applications. All but ABI (Figure 3(b)) have been implemented in Fx, and could be ported to HPF with small changes. The STEREO program, developed at Carnegie Mellon, extracts depth information using the images from multiple video cameras [10]. The RADAR program was adapted from a C program developed by MIT Lincoln Labs to measure the effectiveness of various multicomputers for their radar applications [15]. The SAR program was adapted from a Fortran 77 program developed by Sandia National Laboratories[14]. The MR program was developed from an algorithm by Doug Noll at Pitt Medical Center [13].

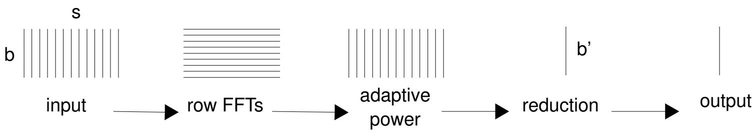
A striking aspect of Figure 3 is the number of parallel DO loops that operate independently along one dimension or another of the array. Each application contains at least one of these loops. The pointwise scaling operation in RADAR is also another form of parallel DO loop, which is usually expressed as an array assignment. Another common pattern from the FFT2 example is: (1) operate along one dimension, then (2) operate along another dimension. This pattern, which occurs in FFT2, ABI, RADAR, SAR, and MR, is typically implemented with a TRANSPOSE intrinsic between (1) and (2). Reductions are found in HIST, STEREO, ABI, and RADAR. The point is that FFT2 and HIST capture the basic computational structure of a wide range of sensor–based computations.

There has been some concern about the performance that can be expected from HPF programs [11]. However, in our experience, the performance of HPF–like programs compiled by the Fx compiler for the Paragon is good, even for moderately sized problems. Figure 4 shows the absolute performance of representatively sized FFT, HIST, and SAR programs. FFT1 is a parallel 1D FFT program, FFT2 is the FFT exemplar, and FFT3 is a parallel 3D FFT program; each is computed in a way similar to FFT. The programs in Figure 4 scale reasonably well (although not linearly) and running time is not dominated by communication overhead. In the case of the SAR program, communication accounts for less 10% of the running time.

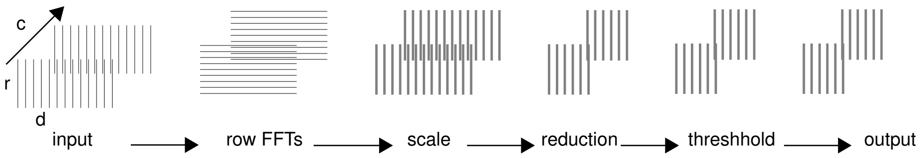
While certainly not exhaustive, Figure 4 offers some hope that good performance can be expected from sensor–based applications written in HPF. In the remaining sections, we will discuss the issues involved in achieving good performance.



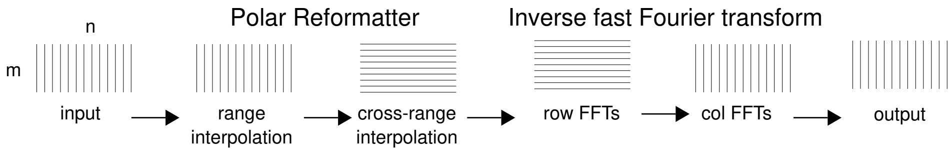
(a) STEREO – Multibaseline stereo.



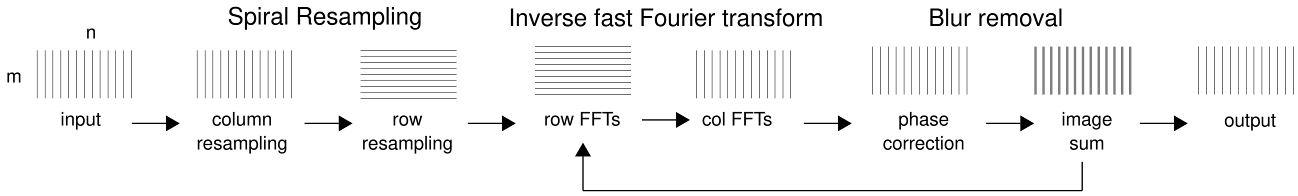
(b) ABI – Sonar adaptive beam interpolation.



(c) RADAR – Narrowband tracking radar.

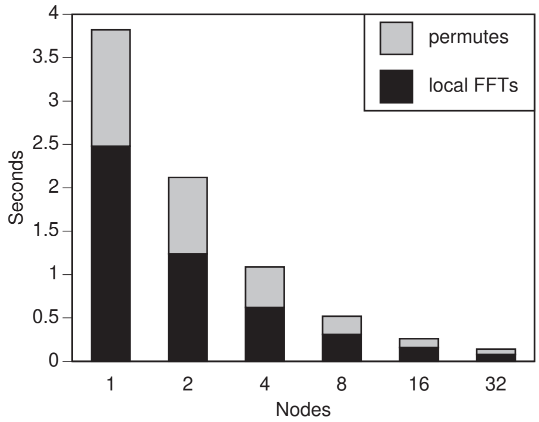


(d) SAR – Synthetic aperture radar image reconstruction.

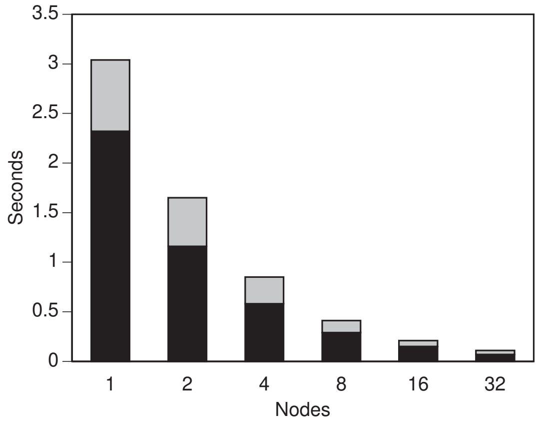


(e) MR – Magnetic resonance image reconstruction.

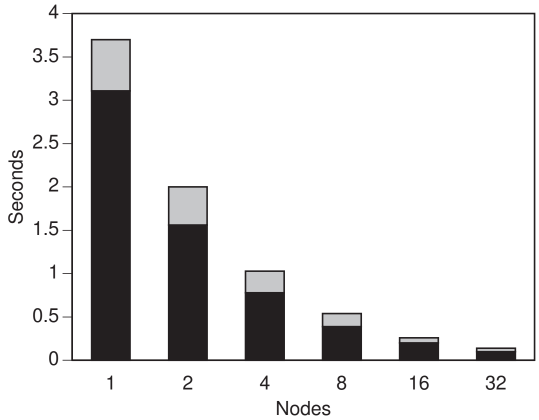
Figure 3: Other Fx/Paragon sensor–based applications



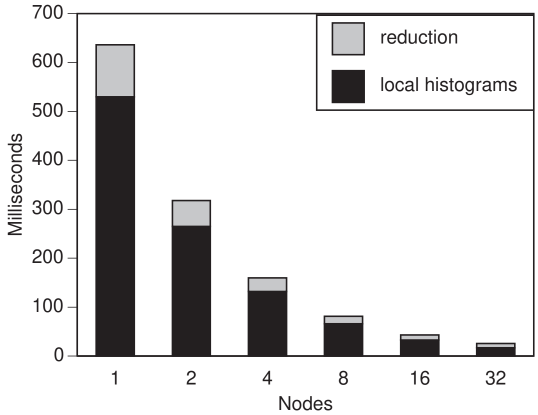
(a) $256K \times 1$ 1D FFT.



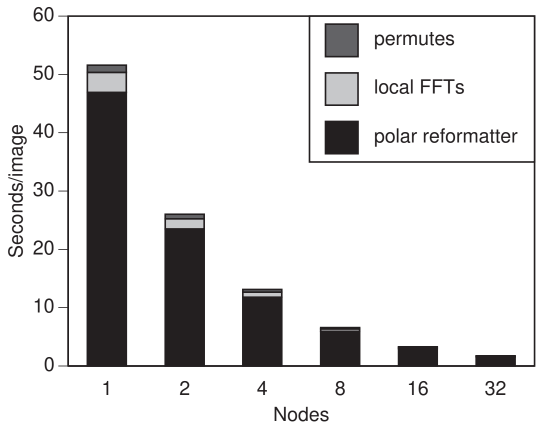
(b) 512×512 2D FFT.



(c) $64 \times 64 \times 64$ 3D FFT.



(d) $1K \times 1K$ image histogram.



(e) 512×512 synthetic aperture radar

Figure 4: Fx/Paragon performance.

4. Parallel loops

DO loops in typical sensor–based applications can be efficiently parallelized using a variation on the simple FORTRAN D copy–in copy–out model [9]. The computation in the main body of the program is modeled as a single thread operating on a global data space. Each iteration in a loop is modeled as a separate thread operating on its own local data space. When control reaches the loop, the contents of the global data space are (conceptually) copied in to each of the local data spaces. Each loop iteration then works independently on its local copy of the global data space. When all of the loop iterations have terminated, the contents of the local data spaces are (conceptually) copied out of the local data spaces back into the global data space. If multiple iterations write to the same address in the local address space, then the values are merged with a user–defined binary associative reduction operator before copying back to the global address space. This loop model, called the *PDO model*, is described in detail in [22].

A parallel DO loop based on the PDO model can be characterized in terms of the addresses that it references. Let R_k denote the set of addresses read by iteration k and W_k the set of addresses written by iteration k. If each R_k is disjoint, then the loop has *disjoint reads*, otherwise the loop has *overlapped reads*. Similarly, if each W_k is disjoint, then the loop has *disjoint writes*, otherwise the loop has *overlapped writes*.

The most common form of loop in sensor–based applications has disjoint reads and writes. Every application in Figures 2 and 3 has at least one loop with disjoint reads and writes, and FFT2 (Figure 2(a)), SAR (Figure 3(d)), and MR (Figure 3(e)) consist exclusively of these kinds of loops. For example, the FFT2 exemplar consists of HPF DO loops of the form:

```
!HPF$ INDEPENDENT
DO k=1,N
 CALL fft(a(:,k))
ENDDO
```

Each invocation of the `fft()` subroutine performs an inplace 1D FFT on the kth column of an array, reading and writing only elements in the kth column. Although `fft()` is a complicated subroutine with a complex pattern of array references (and might even be an assembly language library routine), the pattern of array references between loop iteration is extremely simple: the kth iteration references the kth column. This dichotomy of complicated intra–iteration reference patterns and simple inter–iteration reference patterns is a recurring theme in sensor–based computing, with important implications for HPF implementations.

Another important form of DO loop has overlapped reads and disjoint writes. Loops of this form are typically used to perform convolution operations such as the error computation in STEREO (Figure 3(a)). A similar pattern occurs in relaxation algorithms from scientific computing. For example, a simple 1D convolution is of the form:

```
REAL a(N),b(N),h(3)
!HPF$ INDEPENDENT
DO k=2,N-1
  b(k) = a(k-1)*h(1) + a(k)*h(2) + a(k+1)*h(3)
ENDDO
```

Finally, loops with overlapped writes are typically used by sensor–based applications to implement reductions. We discuss this important class of loops in Section 5. The remainder of this section discusses only loops with disjoint writes.

4.1. Implications for HPF implementations

Generating efficient code inside parallel loops is the key to achieving good performance with sensor–based applications written in HPF. And since parallel loops with disjoint writes are so common, occurring in every application we have studied, generating efficient code for these loops is especially important.

Although loops with disjoint writes are often dismissed as "embarrassingly parallel", it is nontrivial in general to generate efficient parallel code for them. There are a number of reasons, all complicated by the fact that loop bodies of real applications typically contain a lot of code, with complex intra–iteration reference patterns, calls to external library routines, and even inlined assembly language inserts.

First, the HPF compiler must somehow determine that the write sets are disjoint and that addresses that are written by one iteration are not read by another iteration. The HPF INDEPENDENT directive is very helpful here. The INDEPENDENT directive informs the compiler that no address is written by one iteration of a DO loop and then read or written by another iteration.

Second, the HPF compiler must ensure that the read and write sets are aligned with the loop iterations before the iterations are executed. By *aligned*, we mean that the read and write sets of each loop are available in local memory. Aligning the data sets before executing a loop is key to achieving good performance because this allows the programmer to use arbitrary sequential code in the loop body, including calls to efficient sequential math libraries written in assembly language. For example, in a Paragon HPF implementation, the `fft()` routine called by the FFT2 loop might be an assembly language routine hand–crafted for the i860 microprocessor.

Finally, the HPF compiler must compute local loop bounds and translate global array indices in the loop body to local indices. If not handled properly, these computations can be a significant source of runtime overhead.

The Fx compiler relies on a new PDO keyword to assert that write sets are disjoint and that addresses that are written by one iteration are not read by another. (The HPF INDEPENDENT directive conveys the same information and is more compatible with standard F90 compilers.) Also, since the bodies of loops with disjoint writes can be arbitrarily complex, consisting of calls to externally compiled library routines or hand–coded assembly routines, it is not always possible to rely on compile–time analysis to align read and write sets with loop iterations. Thus, the Fx compiler relies on hints from the user that describe the read and write sets for each loop iteration. This is an important point because it allows Fx to generate efficient code for loops with disjoint writes, regardless of the complexity of the loop body.

4.2. Loop efficiency test

There is a simple test, called the *loop efficiency test*, that implementers and users can use to measure the overhead introduced by HPF compilers in the loop bodies of parallel loops. Consider the following canonical parallel DO loop:

```
REAL a(N,N)
```

```
!HPF$ DISTRIBUTE (*,BLOCK):: a

!HPF$ INDEPENDENT
DO k=1,N
  a(:,k) = k
ENDDO
```

This loop requires no communication at runtime, but is somewhat subtle to translate because the *lhs* instance of k must be converted from a global index to a local index, but the *rhs* instance must remain a global index. If we compile and run the loop on P nodes, where P divides N evenly, then the total running time is bounded from below by the running time of the following sequential DO loop:

```
REAL sa(N,N/P)

DO k=1,N/P
  sa(:,k) = k
ENDDO
```

If $L_p(N,P)$ is the running time of the canonical parallel DO loop with N iterations and P nodes, and $L_s(N,P)$ the running time of the corresponding sequential DO loop with N/P iterations, then $E_{loop}(N,P) = L_s(N,P)/L_p(N,P)$ is the *loop efficiency* of the parallel DO loop generated by the HPF compiler. Loop efficiency is a useful measure because it provides a way to isolate the runtime loop overheads that are introduced by the compiler, without having to instrument the generated code. It is important to realize that loop efficiency is not the same as the speedup over the single–node version of the parallel code. Instead, we are comparing to the performance of the tightest *sequential* version of the loop.

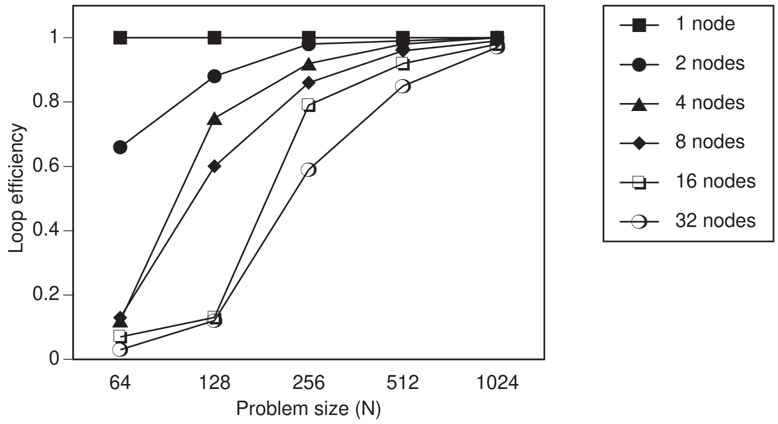


Figure 5: Fx/Paragon loop efficiency ($N \times N$ canonical loop).

If we record loop efficiency for different values of N and P, we get an interesting family of curves. Figure 5 shows the results for the Fx version of the canonical loop on the Intel Paragon.

The family of curves in Figure 5 provides some intestesting insight into the quality of the parallel loops generated by the compiler. Loop efficiency is bounded from above by the curve for 1 node and bounded from below by the curve for 32 nodes, so in general it is only necessary to plot two curves. In Figure 5 the loop efficiency for 1 node is almost unity. For a single node the compiler introduces almost no overhead, which tells us that the loop is nearly as tight as the corresponding sequential loop. The curve for 32 nodes converges to near unity, which tells us that the overheads are being amortized across loop iterations. Further, the 32–node curve converges quite rapidly, with $E_{loop} > 0.5$ at $N = 256$ and $E_{loop} > 0.8$ at $N = 512$. Thus the Fx compiler is introducing minimal overheads that are quickly amortized. This conclusion is confirmed by inspecting the F77 code generated by Fx for $N = 1024$ and $P = 8$:

```
IF (fxcellid.LT.8) THEN
 fxloopstart0 = (MAX(((fxcellid * 128) + 1),1))
 fxlmidx0 = IFXLM(fxadesc,1,fxloopstart0)
 DO k = fxloopstart0, MIN(((fxcellid * 128) + 128),1024), 1
  DO fxindex1 = 1, 1024, 1
   a(fxindex1,fxlmidx0) = (k)
  ENDDO
  fxlmidx0 = fxlmidx0 + 1
 ENDDO
ENDIF
```

The parallel loop overhead consists of a few statements before the loop that compute the local loop bounds, and a function call that computes the initial local index value. The only overhead in the loop body is a statement that increments the local index value. A similar approach to index conversion is first described in [4]. It is hard to imagine a tighter loop.

In summary, the loops in sensor–based applications can often be implemented as independent parallel loops, without any communication between loop iterations. A primary goal of an HPF implementation should be to generate parallel loop bodies that are as efficient as their sequential counterparts. In particular, implementors should focus on minimizing the overhead of parallel loops with disjoint reads and writes. The loop efficiency test we have introduced in this sections provides a simple way to characterize these overheads.

5. Reductions

In sensor–based applications, loops with overlapped writes are used primarily to implement reductions. For example,

```
DO k=1,N
 v = v + a(:,k)
ENDDO
```

A common pattern in sensor–based applications is to operate independently on the columns of an array, and then reduce the columns into a single column by adding them together. The HIST, STEREO, ABI, and RADAR program all perform this type of simple reduction. However there are other sensor–based applications that require a mechanism for the programmer to define generalized reduction operations. For example, a connected components algorithm can be written as a parallel loop over the rows of the image, where each iteration computes a segment table for its row. This is followed by a generalized reduction step that merges the segment tables.

5.1. Implications for HPF implementations

For most sensor–based applications the HPF SUM intrinsic is sufficient. However, the first version of HPF provides no support for operations like connected components that require generalized reductions. Achieving good performance in these cases will require sophisticated compiler analysis to recognize the reductions [5]. In Fx, we avoid this analysis by incorporating a mechanism for defining arbitrary binary associative reductions into the parallel loop construct [22].

5.2. Reduction efficiency test

A user or implementer can measure the quality of the parallel reduction loops generated by an HPF compiler using a test similar to the loop efficiency test in Section 4.2. Consider the following loop that adds the columns of an $N \times N$ array.

```
REAL a(N,N),v(N)
!HPF$ DISTRIBUTE (*,BLOCK):: a

v = 0.0
DO k=1,N
 v = v + a(:,k)
ENDDO
```

The performance of this loop on P nodes is bounded from below by the performance of the following sequential HPF DO loop:

```
REAL sa(N,N/P),v(N)

v = 0.0
DO k=1,N/P
 v = v + sa(:,k)
ENDDO
```

If $R_p(N,P)$ is running time of the parallel reduction of an $N \times N$ array on P nodes and $R_s(N,P)$ the running time of the corresponding sequential reduction of an $N \times N/P$ array, then $E_{reduce}(N,P) = R_s(N,P)/R_p(N,P)$ is the *reduction efficiency* of the parallel reduction generated by the compiler.[2] Reduction efficiency exposes the runtime overheads that are incurred by performing the reduction in parallel. Unlike loop efficiency, which is completely determined by the compiler, reduction efficiency is a function of overheads due to the compiler, as well as overheads due to the underlying communication system. Thus, in general it may be impossible to achieve a reduction efficiency of unity.

Figure 6 shows the results for a simple plus–reduction using the Fx compiler on Paragon. As with the loop efficiency graph, reduction efficiency is bounded from above by the curve for 1 node and from below by the curve for 32 nodes. Surprisingly, the reduction efficiency for a single node decreases as the problem

[2] An alternative formulation of the reduction efficiency test is to use the HPF SUM intrinsic for the parallel reduction. In this case, the sequential reduction must use the same local computational kernel as the SUM intrinsic.

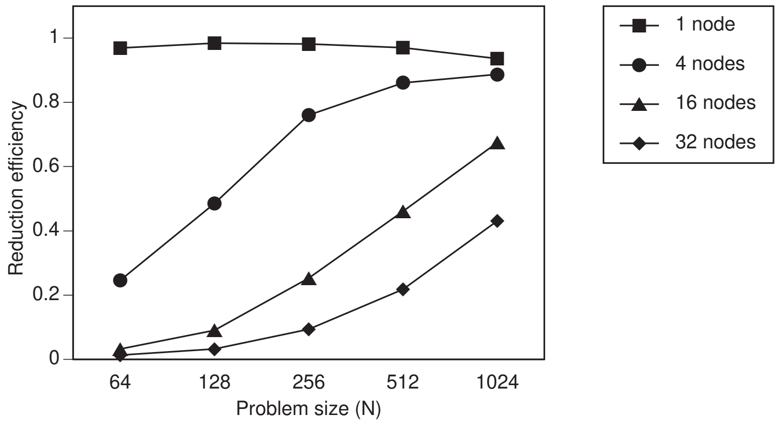


Figure 6: Fx/Paragon reduction efficiency ($N \times N \rightarrow N$ plus-reduction).

size increases. This suggests a problem in the Fx implementation of the reduction. Ideally, the efficiency on a single node should be close to unity. Another point of concern is the slow convergence of the curves for 16 and 32 nodes. Since the local computation step of each parallel reduction grows as roughly N^2/P and the communication step grows as roughly $N \log P$, we might expect these curves to converge faster than they do. Yet even for a relatively large N, the reduction efficiency is below 50%. While the reduction efficiency does not pinpoint the source of the overhead, it does point out an opportunity for improvement in the Fx implementation.

6. Index permutations

As we saw in Section 3, the following computational pattern occurs in many sensor–based applications: (1) operate independently along one dimension of an array, then (2) operate independently along another dimension. The FFT2, ABI, RADAR, SAR, and MR programs all exhibit this pattern. For example, FFT2 performs a local FFT on each column of an array, then performs a local FFT on each row. In order to exploit locality, this pattern is usually implemented with an *index permutation* (also referred to as a transpose or corner turn) between steps (1) and (2):

```
DO k=1,N
 call fft(a(:,k))
ENDDO
b = TRANSPOSE(a)
DO k=1,N
 call fft(b(:,k))
ENDDO
a = TRANSPOSE(b)
```

In general, an index permutation is an assignment from a d–dimensional array a to a d–dimensional array b such that after the assignment,

$$b(i_1, i_2, \ldots, i_d) = a(\pi(i_1, i_2, \ldots, i_d)), \quad 1 \leq i_1, i_2, \ldots, i_d \leq N$$

where each i_k is a unique array index and π is some permutation operator. For example, after the TRANSPOSE step in the FFT2 program, $b(i,j) = a(j,i)$, for $1 \leq i, j \leq N$.

Although most of the sensor–based applications that we have studied transpose 2D arrays, there are important cases where index permutations of higher–dimensional matrices are necessary. In particular, 2D arrays of complex variables are often implemented as 3D arrays of real variables, and for $d > 1$, a d–dimensional FFT must permute two indices of a d–dimensional complex array between each local FFT step.

6.1. Implications for HPF implementations

Efficient index permutation is crucial to achieving good performance in sensor–based applications. In general, an index permutation induces a complete exchange (i.e., an all–to–all personalized communication), where each node sends data to every other node. The Fortran 90 TRANSPOSE and RESHAPE intrinsics [1] adopted by HPF provide an opportunity to optimize this important operation, but unfortunately TRANSPOSE is only defined for 2D arrays. So for the general case, an HPF implementation must either provide a highly tuned RESHAPE intrinsic, or provide a generalized TRANSPOSE extrinsic (this is being considered for HPF-2), or be able to generate efficient code for index permutations that are implemented with a combination of array assignments and DO loops:

```
REAL a(N,N),b(N,N)
!HPF$ DISTRIBUTE (*,BLOCK):: a
!HPF$ DISTRIBUTE (BLOCK,*):: b

b = a
!HPF$ INDEPENDENT
DO k=1,N
 a(:,k) = b(k,:)
ENDDO
```

The Fx compiler provides an index permutation intrinsic with the same basic functionality as a FORTRAN 90 RESHAPE intrinsic with an ORDER argument. The advantage of using an intrinsic is that an intrinsic can leverage off of the existing code in the compiler for generating array assignment statements. Writing an extrinsic with the same functionality means duplicating the compiler's array assignment code in the run–time library. Furthermore, capturing the index permutation in an intrinsic allows the compiler to exploit significant optimizations on systems with toroidal interconnects [8].

6.2. Permutation efficiency test

Just as with loops and reductions, there is a simple test for measuring the efficiency of HPF index permutations. The execution time of a parallel index permutation of an $N \times N$ array (using either the TRANSPOSE intrinsic, a permutation extrinsic, or an assignment statement and a DO loop) is bounded from below by the time to sequentially permute an $N \times N/P$ array on a single node:

```
REAL sa(N/P,N),sb(N,N/P)

DO k=1,N/P
 sa(k,:) = sb(:,k)
ENDDO
```

If $T_p(N, P)$ is the running time of the parallel index permutation of an $N \times N$ array and $T_s(N, P)$ is the running time of the corresponding sequential permutation of an $N \times N/P$ array, then $E_{permute}(N, P) = T_s(N, P)/T_p(N, P)$ is the *permutation efficiency* of the parallel index permutation generated by the HPF compiler.[3] Permutation efficiency is a rough measure of the percentage of effective local memory bandwidth that is realized by the parallel permutation. Like reduction efficiency, permutation efficiency is influenced by overheads due to the compiler, as well as overheads due to the underlying communication system.

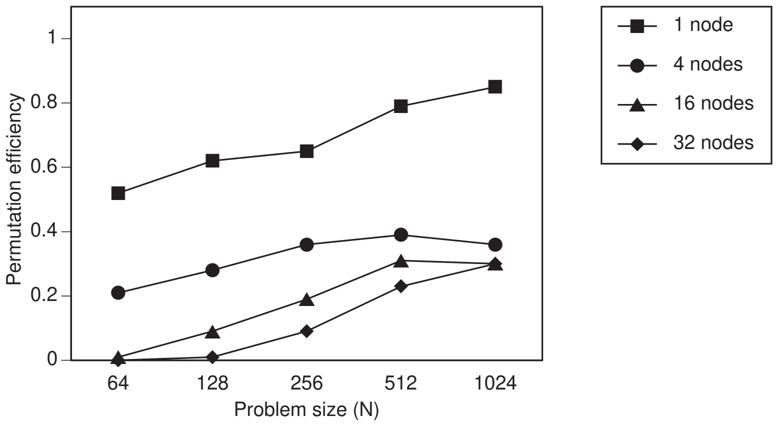


Figure 7: Fx/Paragon permutation efficiency ($N \times N$ 2D transpose).

Figure 7 shows the results for a 2D transpose of an $N \times N$ array using the Fx compiler on Paragon. The graph provides a couple of interesting insights. There is substantial overhead even for the single–node version of the parallel permutation, which achieves only 85% of the effective local memory bandwidth for large problems. The inefficiency is due to the fact that the Fx–generated code unnecessarily checks for communications that never occur (because the transpose is on a single node).

The multiple–node versions of the parallel permutation use only 30% of the effective local memory bandwidth for large problem sizes. This suggests that the parallel permutation on the Paragon is communication–bound, and that further improvements will require a new message passing layer. The inefficiency is largely due to overhead from the underlying communication system and it is tempting for us to wash our hands of responsibility for its performance. However, in our experience, significant performance benefits can be realized in compiler–generated code by tailoring the runtime communication libraries [17, 18]. HPF implementors need to be aware of the communication overheads for a particular target machine. The reduction and permutation efficiency tests are a simple and useful way to expose the performance impact of these overheads.

7. Scalability

Sensor–based applications are typically implemented as collections of functions that process continuous streams of data sets. The sizes of the data sets are determined by external factors such as the type of sensor, the number of sensors, and the frequencies of interest. For example, the image size of the STEREO application is fixed at 240×256 by the camera system and cannot be modified by the programmer. The magnetic resonance scanner used by the MR application processes 512×512 images (oversampled from

[3] As with the reduction efficiency test, if an intrinsic or extrinsic is used for the parallel permutation, then care must be taken to use the same local copy mechanism in the sequential version.

256 × 256 input). The radar subsystem used by the RADAR application produces 512 × 10 data sets.

The fixed size of the data sets is an important property of sensor–based applications that distinguishes them from typical scientific applications. Since the data set sizes are fixed, the amount of available computational work on each node decreases as the number of nodes increases. If a data parallel function performs a nontrivial amount of internal communication, then the efficiency of the function will tend to decrease as the number of nodes increases. This behavior is shown in Figure 7 for a 512 × 512 local FFT loop, a 512 × 512 image histogram, and a 512 × 512 transpose. The local FFT function contains no communication, and thus scales perfectly with the number of nodes. However, the histogram and transpose functions contain internal communication and their efficiency decreases significantly as the number of nodes increases.

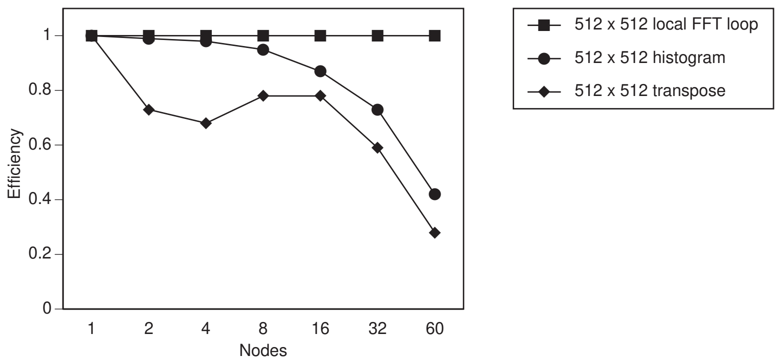


Figure 8: Scalability of various Fx/Paragon functions

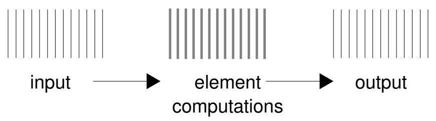
If efficient use of processing nodes is a goal (as it is in embedded systems where additional nodes increase the cost, size, power, and weight of the system) then we want to use a smaller number of nodes for functions like the histogram and transpose. But if we have a large parallel system with many nodes, how can we effectively use the remaining nodes? One approach that has been proposed is to use a mix of task and data parallelism [20, 2, 3, 6].

Task parallelism can significantly improve the performance of applications with functions that do not scale well. For example, using a mix of task and data parallelism doubled the throughput (compared to the most efficient data parallel code) of the 240 × 256 Fx STEREO program so that it was able to run in real–time [19]. Since HPF does not currently support task parallelism (although it is being considered by the HPF–2 committee) there is the risk that HPF sensor–based codes with smaller data sets will not run efficiently. This puts additional pressure on HPF developers to maximize the loop, reduction, and permutation efficiencies identified earlier.

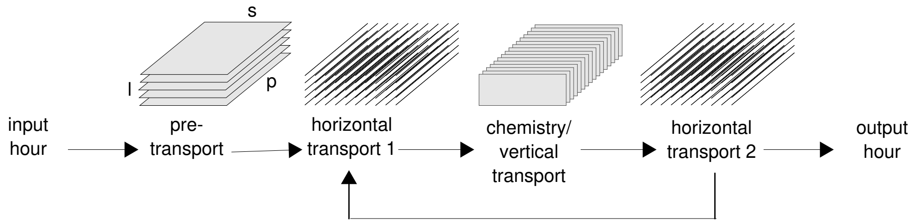
8. Relation to scientific codes

Although this paper does not specifically address scientific HPF codes, other groups have used Fx to implement two nontrivial physical simulations: QUAKE, a 3D earthquake ground motion simulation (based on the method of boundary elements) and AIR, a regional air quality modeling program [12]. Both codes are legacy F77 codes of about 10,000 lines that were ported to the Paragon version of Fx. QUAKE is especially interesting because it was ported in a few weeks by a seismologist from the Southern California Earthquake Center who had never written a parallel program.

Figure 9 shows the computational structure of the QUAKE and AIR programs. QUAKE is a single perfectly parallel DO loop. AIR is a sequence of DO loops that operate on different dimensions of an array, with 3D transposes interspersed between the loops. The interesting thing is that at a high level, the structure of these two moderately large scientific codes is almost identical to the FFT! So again, we see that parallel loops and index permutations are important operations for HPF developers to optimize.



(a) QUAKE – Boundary element earthquake ground motion modeling.



(b) AIR – Finite element air quality modeling.

Figure 9: Complex scientific codes with simple data parallel structure

The QUAKE and AIR programs reinforce an important point that we touched on in Section 4: complicated programs with complicated inner loops can nonetheless have a simple data parallel structure that is straightforward to parallelize. The AIR program takes this to extremes: each iteration of the parallel DO loop in each of the horizontal transport steps solves an independent *sparse and irregular finite element problem*. We normally assume that HPF is not a good platform for sparse irregular codes, but AIR is an example of a sparse code that is quite well suited for HPF.

9. Conclusions

We have identified sensor–based applications as an important class of applications that are generally well suited to HPF. Sensor–based applications operate on dense arrays of data and they reference this data in regular ways that are naturally expressed using HPF loops and array assignment statements. Typical sensor–based applications include fast Fourier transforms, radar and sonar processing, computer vision, and medical imaging.

In the process of studying realistic sensor–based applications compiled by a subset HPF compiler developed at Carnegie Mellon, we observed that performance is determined by the efficiency of three key operations: independent DO loops, reductions, and index permutations. Of course, good performance on these operations is important for many types of applications. The interesting point is that for sensor–based applications, good performance for a small set of operations is generally sufficient for achieving good overall performance.

Implementing an efficient compiler for all of HPF is difficult and expensive. We believe that focusing attention and effort on the performance of independent DO loops, reductions, and index permutations will allow for the rapid development of practically efficient compilers. Based on our experience with the Fx system, HPF developers who focus on these operations will reap large rewards in performance.

Given the importance of loops, reductions, and index permutations for sensor–based applications, we have introduced some simple and useful tests that users and implementers can use to measure the efficiency of the code generated by an HPF compiler for these key operations. The tests are written entirely at the user level, but can provide good insight into the quality of code generated by an HPF compiler. For example, the Fx compiler, as well as several commercial HPF compilers, exhibit a measured loop efficiency of close to 100% for the canonical DO loop. In other words, the loop bodies generated by these compilers are nearly as good as their sequential counterparts. On the other hand, several commercial HPF compilers exhibit a maximum loop efficiency of only 50% for the same DO loop, which suggests a problem in the way these compilers generate loops. A future paper will address the performance of commericial HPF compilers in more detail.

Finally, although HPF is a good match in general with sensor–based applications, scalability can be an issue because of the fixed sizes of the data sets. If the data sets are small, then a mix of task and data parallelism is often required to get good performance.

Acknowledgements

Keith Bromley at the Naval Oceans Systems Center encouraged us to search for similarities in signal and image processing applications. Dennis Ghiglia at Sandia Labs generously provided us with F77 SAR code, which Peter Lieu ported to Fx. Jim Wheeler at GE taught us about underwater sonar applications. Doug Noll at Pitt Medical Center developed the MR algorithm, and Claudson Bornstein, Bwolen Yang, and Peter Lieu implemented it in Fx. Yoshi Hisada from the USC Southern California Earthquake Center took a chance and implemented his 3D boundary element ground motion algorithm in Fx. Ed Segall, Chang-Hsin Chang, and Peter Lieu ported the air quality modeling application to Fx. Thomas Gross, Jim Stichnoth, Bwolen Yang, and Peter Dinda made major contributions to the Fx compiler. Thanks also to the anonymous referees.

References

[1] AMERICAN NATIONAL STANDARDS INSTITUTE, INC. *ANSI Systems Programming Language Fortran(Fortran 90),Draft S8, Version 114 (X3.9-1990)*, Jan. 1990.

[2] CHANDY, M., FOSTER, I., KENNEDY, K., KOELBEL, C., AND TSENG, C. Integrated support for task and data parallelism. *International Journal of Supercomputer Applications 8*, 2 (1994), 80–98.

[3] CHAPMAN, B., MEHROTRA, P., VAN ROSENDALE, J., AND ZIMA, H. A software architecture for multidisciplinary applications: Integrating task and data parallelism. Tech. Rep. 94-18, ICASE, NASA Langley Research Center, Hampton, VA, Mar. 1994.

[4] CHATTERJEE, S., GILBERT, J., LONG, F., SCHREIBER, R., AND TENG, S. Generating local addresses and communication sets for data-parallel programs. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Diego, CA, May 1993), pp. 149–158.

[5] GHULOUM, A., AND FISHER, A. Flattening and parallelizing irregular, recurrent loop nests. In *Proc. of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*

(Santa Barbara, CA, July 1995).

[6] GROSS, T., O'HALLARON, D., AND SUBHLOK, J. Task parallelism in a High Performance Fortran framework. *IEEE Parallel & Distributed Technology 2*, 3 (1994), 16–26.

[7] HIGH PERFORMANCE FORTRAN FORUM. High Performance Fortran language specification, version 1.0. Tech. Rep. CRPC-TR92225, Center for Research on Parallel Computation, Rice University, May 1993.

[8] HINRICHS, S., KOSAK, C., O'HALLARON, D., STRICKER, T., AND TAKE, R. An architecture for optimal all-to-all personalized communication. In *Proc. SPAA '94* (Cape May, NJ, June 1994), ACM, pp. 310–319.

[9] HIRANANDANI, S., KENNEDY, K., AND TSENG, C. W. Compiling Fortran D for MIMD distributed-memory machines. *CACM 35*, 8 (Aug 1992), 66–80.

[10] KANG, S., WEBB, J., ZITNICK, C., AND KANADE, T. A multibaseline stereo system with active illumination and real–time image acquisition. In *Proceedings of the International Conference on Computer Vision* (Cambridge, MA, 1995).

[11] KNIES, A., O'KEEFE, M., AND MACDONALD, T. High Performance Fortran: A practical analysis. *Scientific Programming 3*, 3 (Fall 1994), 187–200.

[12] KUMAR, N., RUSSEL, A., SEGALL, E., AND STEENKISTE, P. Parallel and distributed application of an urban regional multiscale model. Carnegie Mellon Dept. of Mech. Eng. and Dept. of Computer Science, 1995. working paper.

[13] NOLL, D., PAULY, J., MEYER, C., NISHIMURA, D., AND MACOVSKI, A. Deblurring for non-2D fourier transform magnetic resonance imaging. *Magnetic Resonance in Medicine 25* (1992), 319–333.

[14] PLIMPTON, S., MASTIN, G., AND GHIGLIA, D. Synthetic aperture radar image processing on parallel supercomputers. In *Proceedings of Supercomputing '91* (Albuquerque, NM, November 1991), pp. 446–452.

[15] SHAW, G., GABEL, R., MARTINEZ, D., ROCCO, A., POHLIG, S., GERBER, A., NOONAN, J., AND TEITELBAUM, K. Multiprocessors for radar signal processing. Tech. Rep. 961, MIT Lincoln Laboratory, Nov. 1992.

[16] STICHNOTH, J. Efficient compilation of array statements for private memory multicomputers. Tech. Rep. CMU-CS-93-109, School of Computer Science, Carnegie Mellon University, Feb. 1993.

[17] STICHNOTH, J., O'HALLARON, D., AND GROSS, T. Generating communication for array statements: Design, implementation, and evaluation. *Journal of Parallel and Distributed Computing 21*, 1 (Apr. 1994), 150–159.

[18] STRICKER, T., AND GROSS, T. Optimizing memory system performance for communication in parallel computers. In *Proc. 22nd Intl. Symposium on Computer Architecture* (Santa Marguerita di Ligure, Italy, June 1995), ACM, pp. 308–319.

[19] SUBHLOK, J., O'HALLARON, D., GROSS, T., DINDA, P., AND WEBB, J. Communication and memory requirements as the basis for mapping task and data parallel programs. In *Proc. Supercomputing '94* (Washington, DC, Nov. 1994), pp. 330–339.

[20] SUBHLOK, J., STICHNOTH, J., O'HALLARON, D., AND GROSS, T. Exploiting task and data parallelism on a multicomputer. In *Proc. of the ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)* (San Diego, CA, May 1993), pp. 13–22.

[21] WEBB, J. Latency and bandwidth consideration in parallel robotics image processing. In *Supercomputing '93* (Nov. 1993), pp. 230–239.

[22] YANG, B., WEBB, J., STICHNOTH, J., O'HALLARON, D., AND GROSS, T. Do&Merge: Integrating parallel loops and reductions. In *Proc. Sixth Workshop on Languages and Compilers for Parallel Computing* (Portland, OR, Aug. 1993), vol. 768 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 169–183.