HPFBench: A High Performance Fortran Benchmark Suite

Y. CHARLIE HU
Harvard University
GUOHUA JIN
Rice University
and
S. LENNART JOHNSSON, DIMITRIS KEHAGIAS, and NADIA SHALABY
Harvard University

The High Performance Fortran (HPF) benchmark suite HPFBench is designed for evaluating the HPF language and compilers on scalable architectures. The functionality of the benchmarks covers scientific software library functions and application kernels that reflect the computational structure and communication patterns in fluid dynamic simulations, fundamental physics, and molecular studies in chemistry and biology. The benchmarks are characterized in terms of FLOP count, memory usage, communication pattern, local memory accesses, array allocation mechanism, as well as operation and communication counts per iteration. The benchmarks output performance evaluation metrics in the form of elapsed times, FLOP rates, and communication time breakdowns. We also provide a benchmark guide to aid the choice of subsets of the benchmarks for evaluating particular aspects of an HPF compiler. Furthermore, we report an evaluation of an industry-leading HPF compiler from the Portland Group Inc. using the HPFBench benchmarks on the distributed-memory IBM SP2.

Categories and Subject Descriptors: D.3.2 [**Programming Languages**]: Language Classifications—*Concurrent, distributed, and parallel languages*; G.1.3 [**Numerical Analysis**]: Numerical Linear Algebra—*Linear systems* (direct and iterative methods); G.4 [**Mathematics of**

The programs in HPFBench suite were developed by Thinking Machines Corp. in Connection Machine Fortran with partial support from ARPA under subcontract DABT63-91-C-0031 with the Computer Science Department of Yale University and the Northeast Parallel Architectures Center (NPAC) at Syracuse University. Verification, debugging, and documentation were made in part by the Parallel Computation Research Group at Harvard University with support from Thinking Machines Corp. Porting to High Performance Fortran was partially supported by the Computer Science Department and the Center for Research and Parallel Computation of Rice University.
Authors' addresses: Y. C. Hu, Division of Engineering and Applied Sciences, Harvard University, 33 Oxford Street, Cambridge, MA 02138; G. Jin, Computer Science Department, Rice University, 6100 Main Street, Houston, TX 77005; S. L. Johnsson, D. Kehagias, and N. Shalaby, Division of Engineering and Applied Sciences, Harvard University, 33 Oxford Street, Cambridge, MA 02138.

Computing]: Mathematical Software—*Efficiency*; *Parallel and vector implementations*; I.6.3 [**Simulation and Modeling**]: Applications; J.2 [**Computer Applications**]: Physical Sciences and Engineering—*Astronomy*; *Chemistry*; J.3 [**Computer Applications**]: Life and Medical Sciences—*Biology and genetics*

General Terms: Languages, Measurement, Performance

Additional Key Words and Phrases: Benchmarks, compilers, High Performance Fortran

1. INTRODUCTION

High Performance Fortran (HPF) [High Performance Fortran Forum 1993; 1997] is the first widely supported, efficient, and portable parallel programming language for shared- and distributed-memory systems. It continues the Fortran tradition of providing a balanced mix of features for writing portable but efficient programs, and is realized by defining a set of standard extensions to Fortran 90. High-level constructs such as *FORALL* are provided where advanced compilers are believed capable of generating efficient code for different hardware. Programmer control (such as array layout directives) is provided in areas for which compiler optimization remains a challenging problem. Thus, HPF enables application developers to write portable and efficient software that will compile and execute on traditional vector multiprocessors, shared-memory machines, distributed shared-memory machines, message passing distributed-memory machines, and distributed systems, such as networks of workstations.

Since the first release of the HPF specification in 1994, a growing number of vendors have made commercially available HPF compilers, with more vendors announcing plans to join the effort. However, there has not been a systematically designed HPF benchmark suite for evaluating the qualities of the HPF compilers, and the HPF compiler vendors have mostly relied on individual application programmers to feedback their experience often with some particular type of applications on a particular type of architecture (for example, see Hu et al. [1997]). The development of the HPFBench benchmark suite was a first effort to produce a means for evaluating HPF compilers on all scalable architectures.

The functionality of the HPFBench benchmarks covers linear algebra library functions and application kernels. The motivation for including linear algebra library functions is for measuring the capability of compilers in compiling the frequently time-dominating computations in applications.

One motivation for building libraries, in particular in the early years of new architectures, is that they may offer significantly higher performance by being implemented, at least in part, in lower-level languages to avoid deficiencies in compiler technology, or in the implementation of compilers and run-time systems. However, though the functionality of libraries is limited compared to that of applications being run on most computers, implementing libraries in low-level languages tend to be very costly, and often means that high or even good performance may not be available until

late in the hardware product cycle. This in turn implies that following the rapid advances in hardware technology is very difficult, since the older generation hardware often competes successfully with the new generation because of the difference in the quality of software. Thus, it is important to minimize the amount of low-level code also in software libraries, and shift the responsibility of achieving high efficiency to the compiler.

In addition to some of the most common linear algebra functions that are frequently occurring in many science and engineering applications, the HPFBench benchmark suite also contains a set of small application codes containing typical "inner loop" constructs that are critical for performance, but that are typically not found in libraries. An example is stencil evaluations in explicit finite difference codes. The benchmarks were chosen to complement each other, such that a good coverage would be obtained of language constructs and idioms frequently used in scientific applications, and for which high performance is critical for good performance of the entire application. Much of the resources at supercomputer centers are consumed by codes used in fluid dynamic simulations, in fundamental physics, and in molecular studies in chemistry or biology. The selection of application codes in the HPFBench benchmark suite reflects this fact.

The two groups of HPFBench are listed as follows. The names by which we refer to the codes are given in parenthesis.

Linear algebra library functions

(1) *Triangular solvers:*
   (a) *Conjugate Gradient* (`conj-grad`)
   (b) *Parallel cyclic reduction* (`pcr`)
(2) *Fast Fourier transform* (`fft`)
(3) *Gauss-Jordan matrix inversion* (`gauss-jordan`)
(4) *Jacobi eigenanalysis* (`jacobi`)
(5) *LU factorization* (`lu`)
(6) *Matrix-Vector multiplication* (`matrix-vector`)
(7) *QR factorization and solution* (`qr`)

Applications kernels cover the following applications and methods:

(1) *Boson:* many-body simulation (`boson`)
(2) *Diffusion equation:* in three dimensions using an explicit finite difference algorithm (`diff-3d`)
(3) *Poisson's equation* by the Conjugate Gradient method (`ellip-2d`)
(4) *Solution of the equilibrium equations in three dimensions by the finite-element method* (`fem-3d`)
(5) *Seismic processing:* generalized moveout (`gmo`)
(6) *Spectral method:* integration of Kuramoto-Sivashiniski equations (`ks-spectral`)
(7) *Molecular dynamics, Leonard-Jones force law:*
   (a) with local forces only (`mdcell`)
   (b) with long-range forces (`md`)
(8) *Generic direct N-body solvers* with long-range forces (for vortices) (`n-body`)

(9) *Particle-in-cell* in two dimensions:
  (a) straightforward implementation (`pic-simple`)
  (b) sophisticated implementation (`pic-gather-scatter`)
(10) *QCD kernel:* staggered fermion Conjugate Gradient method (`qcd-kernel`)
(11) *Quantum Monte-Carlo* (`qmc`)
(12) *Quadratic programming* (`qptransport`)
(13) *Solution of nonsymmetric linear equations* using the Conjugate Gradient Method (`rp`)
(14) *Euler fluid flow in two dimensions using an explicit finite difference scheme* (`step4`)
(15) *Wave propagation in one dimension* (`wave-1d`)

The concept of measuring performance using benchmark codes is not new. Earlier efforts on benchmarking supercomputer performance (see, for example Wueller-Wichards and Gentzsch [1982], Lubeck et al. [1985], and Dongarra et al. [1987]) focused on ad hoc approaches to the evaluation of systems rather than on potential standardization of the benchmark process. Later benchmarking efforts emphasized more on the methodology and metrics applied to the evaluation of supercomputing systems. In particular, the well-known sets include the Livermore Fortran kernels [McMahon 1988], the LINPACK [Dongarra 1989], and the NAS kernels from NASA/Ames [Bailey and Barton 1985]. Each of these benchmark sets contains codes that closely relate to one particular type of science and engineering research in a particular environment. Furthermore, the benchmarks are mostly designed for measuring uniprocessors though the NAS parallel benchmarks [Bailey et al. 1994] are "paper and pencil" benchmarks that specify the task to be performed and allow the implementor to choose algorithms as well as programming model, and the newer NAS parallel benchmarks 2.0 [Bailey et al. 1995] consists of MPI-based source implementations. The Perfect Club benchmarks [Berry et. al. 1989; Cybenko et al. 1990; Sinvhal-Sharma et al. 1991] is a collection of Fortran 77 application codes originally designed for the evaluation of sequential architectures, though there have been efforts in porting the codes to parallel machines [Cybenko et al. 1990]. Different from earlier benchmarks, the Perfect Club benchmarks focus on whole application codes from several areas of engineering and scientific computing.

The benchmark package most closely related to HPFBench is the PARKBENCH benchmark suite [Hockney and Berry 1994], which represents an international collaborative effort on providing a focused parallel machine benchmarking activities and setting standards for benchmarking methodologies. PARKBENCH suite defines four areas of benchmarking focus, and is collecting actual benchmarks from existing benchmarks suites or from users' submissions. The programming models of the benchmark codes are planned to be both message passing and HPF. The four benchmarking focuses include low-level benchmarks for measuring basic computer characteristics, kernel benchmarks to test typical scientific subroutines, compact

applications to test complete problems, and HPF kernels to test the capabilities of HPF compilers. The motivation for kernel benchmarks and compact applications resembles that of the linear algebra functions and applications kernels in our HPFBench suite, while the HPF kernels in PARKBENCH are for testing different phases of compilations rather than benchmarking whole HPF codes. At the release of the first PARKBENCH report in 1994, the suite contained very few actual codes, and the compact applications were largely missing. The HPFBench efforts were first started in 1990 and therefore overlapped with the PARKBENCH effort. Furthermore, to our knowledge, the HPFBench suite is the first benchmark suite that focuses entirely on the High Performance Fortran programming environments.

In this article, we strive to provide sufficient insight into the benchmark codes for prospective users to choose one or a subset of codes that would best expose and measure specific features of a compiler or system. The source code and the details required for the use of the suite are covered online at http://dacnet.rice.edu/Depts/CRPC/HPFF/benchmarks. The online documentation gives information about how to run the programs and, for each code, the meaning of the arguments, memory requirements as derived from array declarations and layout directives (excluding temporary arrays generated by the compiler and the run-time system), and floating-point operations as a function of input arguments in order to allow for an estimate of the resources and time required to run the code. In all, there are 25 benchmarks in the suite, comprising about 16,000 lines of source code. The full HPFBench benchmark suite (including the sample data files) occupies 2.64MB.

Section 2 describes the methodology of HPFBench. Section 3 summarizes, for each of the HPFBench benchmarks, the employed data structures and their layout, the floating-point operation count, the dominating communication patterns, and some characteristics of the implementation. Section 4 reports on an evaluation of an industry-leading HPF compiler from the Portland Group Inc. using the HPFBench benchmarks on the distributed-memory IBM SP2. Finally, Section 5 summarizes the article.

2. THE HPFBENCH METHODOLOGY

2.1 Language Aspects

The HPFBench benchmark codes are written entirely in High Performance Fortran 1.0 standard [High Performance Fortran Forum 1993]. High Performance Fortran is an extension of Fortran 90. The main differences are a set of data-mapping compiler directives for explicit management of data distribution among processor memories and some parallel constructs for expressing additional parallelism.

2.1.1 *Fortran 90*. Compared to constructs in Fortran 77, some of the new constructs in Fortran 90 specify data references on whole arrays, or segments thereof, such as *CSHIFT*, *EOSHIFT*, *SPREAD*, and *SUM*. In the

context of distributed-memory architectures, these functions define *collective communication* patterns, and are often implemented as library functions, whether explicitly available to the user (as part of the run-time system supporting the compiler) or only callable by the compiler. By specifying the collective communication explicitly, path selection and scheduling can be optimized without sophisticated code analysis. Another Fortran 90 construct is the triplet notation for array sectioning in expressions, which, in general, imply both local memory data motion and communication.

Fortran 90 memory management is more complex than in Fortran 77, and its effective handling is critical for any compiler and associated run-time system for high-performance systems. Thus, Fortran 90 offers both heap- and stack-based dynamic array allocation, and allows the programmer to declare arrays as static, allocatable, or automatic. Dummy array arguments passed in subroutine calls allow for run-time memory management in the form of adjustable, assumed-shape, and assumed-size arrays. The HPFBench codes all declare explicit-shape arrays that either are static or automatic, and the dummy arrays are all adjustable, except in a few cases where static arrays are declared.

In some execution models, like vector architectures, one processor, the Control Processor, (for vector architectures the scalar processor) performs scalar operations and instruction storage and broadcast. Therefore, the use of scalar variables in array expressions often implies communication between the Control Processor and the processing elements. Since this communication is critical for all operations, the communication network between the Control Processor and the processing elements is often of a higher capacity than the network between the processing elements, but not of as high a capacity as that between individual processing elements and their associated memory. Several of the HPFBench codes contain constructs that imply communication between the Control Processor and the processing elements. The HPFBench `n-body` code contains constructs that allow for a clear comparison between the two networks: Control Processor to and from processing elements, and between processing elements.

2.1.2 *HPF Extensions*. The main extensions of HPF on top of Fortran 90 are

(1) data-mapping directives,

(2) parallel *FORALL* statements and constructs and *INDEPENDENT DO*,

(3) a set of library procedures, and

(4) interfaces to extrinsic procedures.

Data-mapping directives, including data alignment and distribution directives, allow the programmer to advise the compiler how to assign array elements to processor memories. HPF 1.0 standard defines three kinds of distributions for each array axis: local to a processor, block, and cyclic. Tables V and IX list the distribution of arrays used in dominating compu-

Table I. Summary of Fortran 90 Constructs in the HPFBench Benchmark Codes

Language Constructs	Array Data Structures			
	1D	2D	3D	(4-7)-D
CSHIFT	fft-1d conj-grad jacobi pcr(1) qptransport wave-1d	fft-2d ks-sprectral jacobi pcr(2) ellip-2d n-body step4, pic-simple	fft-3d boson rp pcr(3)	mdcell qcd-kernel
Array Sections			diff-3d	
SPREAD		matrix-vector(1) qr:factor/qr:solve gauss-jordan jacobi lu:nopivot/lu:pivot md, n-body	matrix-vector(2,3,4)	
SUM		matrix-vector(1) qr:factor/qr:solve ellip-2d ks-spectral md qmc	matrix-vector(2,3,4) rp	
MAXVAL,MINVAL			mdcell	qmc
MAXLOC		gauss-jordan lu:pivot		
WHERE	fft-1d	fft-2d jacobi qr:factor/qr:solve	fft-3d fem-3d mdcell pic-gather-scatter	
Mask in SUM		qr:factor/qr:solve		
Mask in MAXLOC		gauss-jordan		

tations of the linear algebra and application kernel codes, respectively. All codes using arrays of two or more dimensions include layout directives, which affect the storage-to-sequence association.

Parallel *FORALL* statements and constructs and *INDEPENDENT DO* allow fairly general array sectioning and specifications of parallel computations. The HPFBench benchmark suite contains the *FORALL* construct in a variety of situations: applied to data within individual processors only (i.e., no communication required), applied to a mixture of local and remote data, with and without masks, etc.

HPF also defines a set of library procedures which serve as interfaces to run-time systems for collective communications. A subset of HPF library

Table II. Summary of HPF Language Constructs in the HPFBench Benchmark Codes

Language Constructs	Array Data Structures			
	1D	2D	3D	(4-7)-D
Layout Directives	All Codes			
FORALL		jacobi diff-2d gmo pic-simple	mdcell pic-gather-scatter	
INDEPENDENT DO			mdcell	
Mask in FORALL	boson	pic-simple	pic-gather-scatter	
sum_prefix/copy_prefix	qptransport	qmc	pic-gather-scatter	
sum_scatter		pic-simple	pic-gather-scatter	
copy_scatter	wave-1d	lu:pivot gauss-jordan mdcell pic-simple, qmc		
grade_up	pic-gather-scatter qptransport			

procedures are tested in a few HPFBench codes. These include scatter and vector scatter operations, parallel prefix operations, and the sort operation.

Lastly, HPF defines extrinsic mechanism by which an HPF program can interoperate with program units written in other programming paradigms, such as explicit message-passing SPMD style. Extrinsic procedures are not used by any of the HPFBench benchmark codes.

The Fortran 90 and HPF unique language constructs used in the HPF-Bench benchmark suite are summarized in Tables I and II.

2.2 Performance Metrics

The primary performance metrics that are output by each HPFBench benchmark code are as follows:

—*Elapsed time (in seconds):* total wall clock time spent in executing the benchmark.

—*Elapsed FLOP rate (in MFLOPs):* number of million floating-point operations per second obtained by dividing the stated FLOP count for the benchmark by the elapsed time.

—*Communication time breakdowns:* the amount of wall clock time spent in each of the different communications performed in the benchmark.

A frequently used performance measurement, arithmetic efficiency, is computed by dividing the FLOP rate by the peak FLOP rate of all the

Table III. Number of FLOPs Accounted for Each Operation Type

Operation Type	FLOPs
$+, -, \times$	1
$\div, \sqrt{}$	4
log, exp, sin, cos, ...	8

participating processors. Since the peak FLOP rate varies with the underlying machines and can be easily calculated once fixing a target machine, we leave this measurement out from the output of the benchmarks.

For benchmarks with different subroutines, performance metrics for different modules of a benchmark are reported separately. For instance, the factorization and solution times for `qr` are reported separately.

In addition to the above performance metrics that each benchmark outputs at the run-time, the following metrics of the benchmarks are detailed in Section 3 which gives the benchmark descriptions. Such metrics characterize the benchmarks in the HPFBench suite and can be used to assist a user in choosing appropriate benchmarks for his or her specific evaluation needs.

—*FLOP count:* In counting the number of FLOPs we adopt the operation counts suggested in Hennessy and Patterson [1990] which were also used in Livermore Fortran kernels [McMahon 1988] and PARKBENCH [Hockney and Berry 1994], and summarized in Table III, for addition, subtraction, multiplication, division, square root, logarithms, exponentials, and trigonometric functions. For reduction and parallel prefix operations, such as the intrinsic *SUM* and segmented *Scan*s, we use the sequential FLOP count, i.e., $N - 1$ for N element one-dimensional arrays.

The performance evaluation and analysis are based on the execution semantics of HPF. Thus, the execution of the statement `vtv = sum(v*v, mask)` implies that the self inner product of the vector `v` is executed for all elements, rather than only the unmasked ones. As for the summation that is performed under mask, we only count operations associated with mask elements being *true* when the mask is data independent (predictable at compile time). Otherwise, we count operations as if all elements of the mask were true.

—*Memory usage (in bytes):* The reported memory usage only covers user-declared data structures including all the auxiliary arrays required by the algorithm's implementation, given the data type sizes of Table IV. Temporary variables and arrays that may be generated by the compiler are not accounted for.

In the case where a lower-dimensional array L is aligned with a higher-dimensional array H, L effectively takes up the storage of $size\{H\}$, and we report the collective memory of L and H to be $2 * size\{H\}$.

—*Communication pattern:* We specify the types of communication an algorithm exhibits, as well as the language constructs with which they

Table IV. Data Type Sizes (in bytes) for Standard 32–Bit-Based Arithmetic Architectures

Data Type	Size
integer	4
logical	4
double-precision real	8
double-precision complex	16

are expressed. These communication types include stencils, gather, scatter, reduction, broadcast (*SPREAD*), all-to-all broadcast (AABC) [Johnsson and Ho 1989], all-to-all personalized communication (AAPC) [Johnsson and Ho 1989], butterfly, *Scan*, circular shift (*CSHIFT*), send, get, and sort. It should be noted that more complex patterns (such as stencils and AABC) can be implemented by more than one simpler communication function (for instance *CSHIFT*s, *SPREAD*s, etc.).

—*Operation count per iteration (in FLOPs):* We give the number of floating-point operations for one iteration in the main loop. This metric serves as the first order approximation to the computational grain size of the benchmark, giving insight into how the program scales with increasing problem sizes.

—*Communication count per loop iteration:* We group the communication patterns invoked by this benchmark and specify exactly how many such patterns are used within the main computational loop. This metric, together with the operation count per iteration, give the relative ratio between computation and communication in the benchmark.

2.3 Benchmark Guide

In this section, we provide some guidelines on how to select benchmarks from the HPFBench suite for evaluating specific features of an HPF compiler.

First, the salient features of the HPFBench benchmark listed in the previous section codes are summarized in a set of tables through out the article as follows.

—*Language constructs:* Tables I and II.

—*Data Layout:* Table V for linear algebra functions and Table IX for application kernels.

—*Collective communications:* The use of certain types of collective communications is covered by Tables VI and X for linear algebra and application kernels, respectively. Implementation techniques for some of the communications in the application kernels are covered in Table XI.

—*Computation/communication ratio, memory usage:* The number of arithmetic operations per communication as well as the benchmark's memory usage is listed in Table VII for library codes and in Table XII for application kernels.

We now discuss how one can use the features summarized in various tables to help choosing appropriate benchmarks to evaluate specific features of a compiler.

Since individual processor performance is critical for high performance on any scalable architecture, the HPFBench benchmark suite includes two codes that do not invoke any interprocessor communication: `matrix-vector(3)` and `gmo`.

Broadcast, reduction, and nearest-neighbor array communication are important programming primitives that appear in several benchmarks, some of which largely depend on these primitives for efficient execution. For instance, on most architectures, the matrix-vector multiplication benchmark will be dominated by the time for broadcast (*SPREAD*) and reduction (*SUM*). *SPREAD* is also likely to dominate the execution time for the Gauss-Jordan matrix inversion benchmark, while *CSHIFT* is expected to have a significant impact on the performance for both tridiagonal system solvers, the FFT, and several of the application kernels implementing stencils through *CSHIFT*s, such as `boson`, `ellip-2d`, `mdcell`, `rp`, and `wave-1d`.

The linear algebra subset of the HPFBench benchmark suite is provided to enable testing the performance of compiler-generated code against that of a highly optimized library, such as the ESSL and PESSL [IBM 1996] for the IBM SP2. Performance attributes for the linear algebra codes are presented in Tables V–VII, tabulating the data representation and layout for dominating kernel computations, the communication pattern along with their associated array ranks, and the computation-to-communication ratio in the main loop. These tables can be used to decide on an appropriate benchmark code according to a given testing criteria. For instance, if a user desires to evaluate how a particular compiler implements *CSHIFT* on a two-dimensional array, Table VII indicates that there are three choices: `jacobi`, `fft-2d`, and `pcr(2)`. If having local axis (and therefore local memory addressing) in the main computational kernel is desired (or not), then by Tables V and VII the `pcr(2)` is picked (or ruled out). In the latter case, to decide between the `jacobi` and `fft-2d` codes, Table VII shows the other communication patterns associated with each. Hence, if minimizing other communication patterns is the goal, the `fft-2d` code would be selected. Conversely, if evaluating broadcasts and sends is also of interest, then `jacobi` would be the appropriate choice.

The application kernels of the benchmark suite are intended to cover a wide variety of scientific applications typically implemented on parallel machines. Table VIII captures some of the essential features by which many application codes are classified. Many finite difference codes using explicit solvers require emulation of grids on one or several dimensions. Depending on the layout of the associated arrays, the interprocessor may or may not correspond directly to the dimensionality of the data arrays. Table VIII specifies the interprocessor communication implied in the application kernels. The application kernels contain one code for unstructured grid computations (`fem-3d`), pure particle codes (`n-body` and `md`), and codes

that make use of both regular grid structures and particle representations (`mdcell`, `pic-simple`, and `pic-gather-scatter`). Tables VIII–XII can be used to aid in the selection of one code or a subset of benchmark codes for a specific task. If, for example, an application code with an AABC is desired, then Table X yields the codes `md` and `n-body`. Table IX shows that only `n-body` is guaranteed to perform an AABC with respect to the processing elements, since one of the axes is local while both are distributed in `md`. Both codes perform AABC communication with respect to the index space. On the other hand, Table XI states that if the implementation of AABC in the form of *SPREAD*s is of interest, then `md` is the code of choice. Alternatively, `n-body`'s AABC includes *CSHIFT*s and a *Broadcast* from one processor or the Control Processor.

2.4 Performance Evaluation

The HPFBench benchmarks are entirely written in HPF. To measure the overhead of HPF compiler-generated code on a single processor, we also provide an Fortran 77 version for each of the HPFBench benchmarks. On parallel machines, an ideal evaluation of performance of the compiler-generated code is to compare with that of a message-passing version of the same benchmark code, as message-passing codes represent the best performance tuning effort from low-level programming. Message passing implementations of LU, QR factorizations, and matrix-vector multiplication are available from IBM's PESSL library [IBM 1996], and in other libraries such as ScaLAPACK [Blackford et al. 1997] from University of Tennessee, among others. However, the functions in such libraries often use blocked algorithms for better communication aggregation and BLAS performance. A fair comparison with such library implementations thus requires sophisticated HPF implementations of the blocked algorithms.

In the absence of message-passing counterparts that implement the same algorithms as the HPFBench codes, we adopt the following two-step methodology in benchmarking different HPF compilers.

—First, we compare the single-processor performance of the code generated by each HPF compiler versus that of the sequential version of the code compiled under a native Fortran 77 compiler on the platform. Such a comparison will expose the overhead of the HPF compiler-generated codes.

—Second, we measure the speedups of the HPF code on parallel processors relative to the sequential code. This will provide a notion on how well the codes generated by an HPF compiler scale with the parallelism available in a scalable architecture.

2.5 Benchmarking Rules and Submission of Results

We encourage online submission of benchmarking results for the HPFBench codes. The ground rule for performance measurement is as follows: only HPF directives, including data layout for arrays and computation

partitionings, and source code segments for collective communications such as those listed in Tables VI and X can be modified (but still using HPF/F90 language features) so that maximum performance can be achieved by compiler-generated codes on a particular machine.

We encourage the submission of performance measured on both single-processor and parallel systems. The relative single-processor performance on the same architecture would measure the overhead of different HPF compilers, while the parallel performance reflects the scalability of the generated code.

Benchmark results can be submitted through the HPF Forum Web server accessible from the HPFBench Web page http://dacnet.rice.edu/Depts/CRPC/HPFF/benchmarks. A complete submission of benchmarking results should include

—a detailed description of the hardware configuration including machine model, CPU, memory, cache, interconnect, and software support including operating system, compiler, and run-time libraries used for the benchmark runs;

—directive and source code changes to the original benchmark codes; and

—problem size used and output results from the benchmarks.

Due to the large numbers of benchmark codes in the HPFBench suite, we define a subset of representative benchmarks, or core benchmarks, in Section 3.4. Results for the core benchmarks are encouraged as the minimum required test set for benchmarking.

3. THE HPFBENCH BENCHMARK SUITE

The functionality of the HPFBench benchmarks covers linear algebra library functions that frequently appear as the time-dominant computation kernels, and application kernels that contain time-dominant computation kernels that are not linear algebra kernels. This section gives a brief description of each of the benchmark codes, together with the main data structures used, the algorithms employed, and the communication patterns.

3.1 Library Functions for Linear Algebra

Linear algebra functions frequently appear as the time-dominant computation kernels of large applications, and often hand-optimized as mathematical library functions by the supercomputer vendors (e.g., ESSL and PESSL [IBM 1996] from IBM). These hand-optimized implementations attempt to make efficient use of the underlying system architecture through efficient implementation of interprocessor data motion and management of local memory hierarchy and data paths in each processor. Since these are precisely the issues investigated in modern compiler design for parallel languages and on parallel machine architectures, the library subset of the HPFBench benchmark suite is provided to enable testing the performance

Table V. Data Distributions of Arrays in Dominating Computations of the Linear Algebra Functions, as Specified Using the DISTRIBUTE Directive. The distribution of an axis is either local to a processor, denoted as "*", or sliced into uniform blocks and distributed across the processors, denoted as "b", short for "block," or distributed across the processors in a round-robin fashion, denoted as "c", short for "cyclic. Benchmarks lu and qr are the only benchmarks that would benefit from "Cyclic" data distribution from better load balancing.

		Arrays			
Code		1D	2D	3D	4D
conj-grad		X(b)			
fft:	1d	X(b)			
	2d		X(b,b)		
	3d			X(b,b,b)	
gauss-jordan		X(b)	X(b,b)		
jacobi		X(b)	X(b,b)		
lu			X(b,b) or X(c,c)		
matrix-vector:	(1)	X(b)	X(b,b)		
	(2)		X(b,b)	X(b,b,b)	
	(3)		X(*,b)	X(*,*,b)	
	(4)		X(b,b)	X(*,b,b)	
pcr:	(1)	X(b)	X(*,b)		
	(2)		X(b,b)	X(*,b,b)	
	(3)			X(b,b,b)	X(*,b,b,b)
qr			X(b,b) or X(c,c)		

of compiler-generated code against that of any highly optimized library, such as the PESSL.

The linear algebra library functions subset included in the HPFBench suite is comprised of matrix-vector multiplication, dense matrix solvers, two different tridiagonal system solvers (based on parallel cyclic reduction and the Conjugate Gradient method respectively), a dense eigenanalysis routine, and an FFT routine. Most of them support multiple instances, e.g., multiple instances of tridiagonal systems are solved concurrently by calling the appropriate pcr function once.

We summarize some of the important properties of our implementations of the linear algebra benchmarks by means of three tables. Table V gives an overview of the data representation and layout for the dominating computations. Table VI shows the communication operations used along with their associated array ranks. Table VII tabulates the computation-to-communication ratio in the main loop of each linear algebra benchmark.

3.1.1 *Conjugate Gradient (conj-grad).* This benchmark uses the Conjugate Gradient method [Golub and van Loan 1989] for the solution of a

Table VI. Communication Pattern of the Linear Algebra Functions (the Array Dimensions for Reduction and Broadcast Are of Source and Destination, Respectively). MAXLOC is a type of reduction that returns the index of the largest array element.

	Arrays		
Communication Pattern	1D	2D	3D
Cyclic shift	fft-1d pcr (1) jacobi conj-grad	fft-2d pcr (2) jacobi	fft-3d pcr (3)
Broadcast		matrix-vector(1) qr: factor/qr: solve gauss-jordan jacobi	matrix-vector(2,3,4)
Reduction		matrix-vector(1) qr: factor/qr: solve jacobi	matrix-vector(2,3,4)
MAXLOC		gauss-jordan lu: pivot	
Scatter		gauss-jordan fft-2d lu: pivot	fft-3d
Gather		gauss-jordan	

single instance of a tridiagonal system. The tridiagonal system is stored in three 1D arrays. The tridiagonal matrix-vector multiplication required in the Conjugate Gradient method corresponds to a three-point stencil in one dimension. It is implemented using *CSHIFT*s. Unlike the *CSHIFT*s in the parallel cyclic reduction method, the *CSHIFT*s in the Conjugate Gradient method are only for nearest-neighbor interactions.

3.1.2 *Fast Fourier Transform (`fft`).* These routines compute the complex-to-complex Cooley-Tukey FFT [Cooley and Tukey 1965]. One-, two-, or three-dimensional transforms can be carried out. In the HPFBench benchmark, the twiddle computation is included in the inner loop. It implements the butterfly communication in the FFT as a sequence of *CSHIFT*s with offsets being consecutive powers of two. The structures of the code for two- and three-dimensional transforms are similar.

In addition to *CSHIFTs* used to implement the butterfly communication, the bit-reversal operation is performed to reorder the output data points. Bit-reversal defines a communication pattern that has large demands on the network bandwidth and often cause severe network contention for many common networks and routers. With the data layout in the HPF-Bench benchmark, the bit-reversal constitutes an all-to-all personalized communication (AAPC) whenever the size of the local data set of the axis subject to bit-reversal is at least as large as the number of processing nodes

Table VII. Computation-to-Communication Ratio and Memory Usage in the Linear Algebra Functions. In general, 1D, 2D, and 3D arrays are of size n, n^2, and n^3, respectively, except `matrix-vector` and `qr` which use 2D arrays of size mn. `matrix-vector`, `lu`, `qr`, and `pcr` operate on multiple instances of matrices or linear systems, and the number of instances is denoted using i. Finally, r denotes the number of right-hand sides of linear systems as in `lu`, `pcr`, and `qr`.

Code	FLOP Count (per iteration)	Memory Usage (in bytes)	Communication (per iteration)
conj-grad	$26n$	$40n$	4 CSHIFTs, 3 Reductions
fft-1d	$5n$	$100n$	2 CSHIFTs
fft-2d	$10n^2$	$115n^2$	4 CSHIFTs
fft-3d	$15n^3$	$136n^3$	6 CSHIFTs
gauss-jordan	$n(n + 2 + 2n^2)$	$32n^2+16n$	n Reduction, 3n Sends, 2n Gets, 2n Broadcasts
jacobi	$n(6n^2+26n)$	$88n^2+4n$	2n CSHIFTs on 1D arrays, 2n CSHIFTs on 2D arrays, 2n Sends, 4n 1D to 2D Broadcasts
lu: nopivot	$2/3n^3i$	$8n(n + 2r)i$	n Reduction, n Broadcast
lu: pivot	$2/3n^3i$	$8n(n + 2r)i$	n Reduction, n Broadcast
matrix-vector	$2nmi$	$8(n + nm + m)i$	1 Broadcast, 1 Reduction
pcr	$(5r + 12)ni$	$8(r + 4)ni$	$(2r + 4)$ CSHIFTs
qr: factor	$(5.5m - 0.5n)n^2$	$36mn$	2n Reductions, 2n Broadcasts
qr: solve	$(8m - 1.5n)n^2$	$44mn + 8m(r + 1)$	2n Reductions, 4n Broadcasts

along the axis subject to bit-reversal. A detailed analysis of the parallel FFT can be found in Johnsson et al. [1992].

The FFT is one of the most widely used algorithms in science, engineering design, and in signal processing. Being a very efficient algorithm, FFT has relatively low operation count per data point, namely, $O(\log n)$, but its communication is global and extensive. Hence, FFTs tend to expose weaknesses in communication systems, in particular a low bisection bandwidth. It is also a good benchmark for the handling of complex arithmetic, and (local) memory hierarchies.

3.1.3 *Gauss-Jordan Matrix Inversion (`gauss-jordan`).* Given a square matrix A, the *Gauss-Jordan* routines compute the inverse matrix of A, A^{-1}, via the Gauss-Jordan elimination algorithm with partial pivoting [Golub and van Loan 1989; Wilkinson 1961]. Pivoting is required if the system is not symmetric positive definite. The pivot element is chosen from the pivot row, and the columns are permuted. At each pivoting iteration, this variant of the algorithm subtracts multiples of the pivot row from the rows above as well as below the pivot row. Thus, both the upper and lower triangular

Table VIII. Characterization of the Application Kernels

Function		HPFBench Code
Embarrassingly parallel		gmo
Structured grid emulation	1D	wave-1d
	2D	boson, ellip-2d, step4
	3D	diff-3d, rp, mdcell
	4D	qcd-kernel
Unstructured grid emulation		fem-3d
Particle-particle interaction		
global	2D	n-body
	3D	md
local	3D	mdcell
Particle-grid interaction	2D	pic-simple
	3D	pic-gather-scatter, mdcell
FFT	1D	wave-1d
	2D	ks-spectral, pic-simple

matrices are brought to zero. An analysis of the numerical behavior of the algorithm can be found in Dekker and Hoffman [1989]. Rather than replacing the original matrix with the identity matrix, this space is used to accumulate the inverse solution.

Since there is no alignment between the layout of 1D arrays used as temporary arrays in swapping rows and columns of the 2D arrays during total pivoting, data motion occurs in the swappings. *SPREAD* communication is used for spreading pivot rows and columns to 2D temporary arrays.

3.1.4 *Jacobi Eigenanalysis (`jacobi`).* The HPFBench routines are only valid for real symmetric matrices. Given a real symmetric matrix A of size $n \times n$, the benchmark uses the Jacobi method to compute the *eigenvalues* of the matrix A. *Eigenvectors* are *not* computed within the benchmark. The Jacobi method makes iterative sweeps through the matrix. In each sweep, successive rotations are applied to the matrix to zero out each off-diagonal element. A sweep consists of the application of $n(n - 1)/2$ rotations. As each element is zeroed out, the elements previously zeroed out generally become nonzero again. However, with each step, the square root of the sum of the squares of the off-diagonal elements decreases, eventually approaching zero. Thus, the matrix approaches a diagonal matrix, and the diagonal elements approach the eigenvalues. For a detailed description of this method see Golub and van Loan [1989] and Schroff and Schreiber [1988].

The Jacobi eigenanalysis benchmark is interesting in that it uses both 1D and 2D arrays with an extraction of the diagonal taking place in computing rotation factors and an alignment and broadcast taking place in applying the rotation factors. Aligning the 1D arrays with the 2D arrays

Table IX. Data distributions of arrays in Dominating Computations of the Application Kernels, as Specified Using the DISTRIBUTE Directive. The distribution of an axis is either local to a processor, denoted as "*", or sliced into uniform blocks and distributed across the processors, denoted as "b", short for "block." "Cyclic" distribution is not used in any of the applications benchmarks.

	Arrays				
Code	1D	2D	3D	4D, 6D, 7D	Unstructured Grid
boson			X(*,b,b)		
diff-3d			X(b,b,b)		
ellip-2d		X(b,b)			
fem-3d					X(*,b,b), X(*,*,b)
gmo	X(b)	X(*,b)			
ks-spectral		X(b,b)			
mdcell				X(*,b,b,b)	
md	X(b)	X(b,b)			
n-body		X(*,b)			
pic-simple		X(*,b)	X(*,b,b)		
pic-gather-scatter		X(*,b)	X(*,b,b)		
qcd-kernel				X(*,b,b,b,b,b)	
				X(*,*,b,b,b,b,b)	
qmc		X(b,b)		X(*,*,b,b)	
qptransport	X(b)				
rp			X(b,b,b)		
step4			X(*,b,b)		
wave-1d	X(b)				

result in poor load balance for the computation of rotation factors, while not aligning the arrays yields good load balance, but results in a potentially high communication cost.

The main communication patterns include nearest-neighbor *CSHIFT*s under masks on 1D rotation vectors, *SPREAD*s for duplicating 1D rotation vectors into 2D arrays, and *CSHIFT*s on 2D arrays.

3.1.5 *LU Factorization (lu).* Given a dense square matrix A of size $n \times n$, and, a right-hand-side vector of size n, these routines solve the dense system of equations $AX = B$ by factoring the matrix A into a lower triangular matrix L and an upper triangular matrix U, such that $A = LU$. The factorization method is Gaussian elimination with or without partial pivoting. Load balance is a well-known issue for LU factorization, and the desired array layout is cyclic distribution. Thus the lu benchmark codes uses two-dimensional arrays with cyclic distributions.

3.1.6 *Matrix-Vector Multiplication (matrix-vector).* This HPFBench benchmark is a collection of routines computing one or more matrix-vector products. Given arrays **x**, **y**, and **A** containing *multiple instances* [Johnsson et al. 1989] of the vectors x and y and the matrix A, respectively, the matrix-vector routines perform the operation $y \leftarrow y + Ax$ for each instance. The matrix-vector multiplication is implemented for the following array layouts:

(1) one instance of the three operands with each instance spread over all the processors,

(2) multiple instances with each instance of each operand occupying a subset of the processors,

(3) multiple instances with each instance of the corresponding operands allocated to the memory unit associated with one processor. This layout requires no communication and represents a truly embarrassingly parallel case,

(4) multiple instances with the row axis (the axis crossing different rows) of array **A** allocated local to the processors, and the other axis of **A** as well as the axes of the other operands spread across the processors. This layout only requires communication during the reduction.

For all cases, the spread-and-reduction algorithm is used, i.e., $y = sum(A * spread(x, dim = dim2), dim = dim1)$. Since a compiler typically allocates some temporary arrays to store the intermediate results when compiling the expression, the execution time for this implementation will on most architectures be dominated by the *SPREAD* and *SUM* operations, and the implicit alignments of the temporary arrays with input arrays **x** and **A** and output array **y**.

Matrix-vector multiplication is a typical level-2 BLAS operation. It is the dominating operation in iterative methods for the solution of linear systems of equations. It only requires two floating-point operations per matrix element, and its performance is very sensitive to data motion. In case one above, each operand is distributed across all nodes such that the input vector must be aligned with the matrix, and the result vector aligned with the output vector as part of the computation.

3.1.7 *Parallel Cyclic Reduction (`pcr`).* Parallel Cyclic Reduction is one of the two tridiagonal solvers in HPFBench. It is different from the other tridiagonal solver, `cond-grad`, both in the systems to be solved and in the methods used. While `cond-grad` solves a single-instance tridiagonal system, this code handles multiple instances of the system $AX = B$. The three diagonals representing A have the same shape and are 2D arrays. One of the two dimensions is the *problem axis* of extent n, i.e., the axis along which the system will be solved. The other dimension is the *instance axis*. For multiple right-hand sides, B is 3D. In this case, its first axis represents the right-hand sides, is of extent r, and is local to a processor. Excluding the first axis, B is of the same shape as each of the arrays for the diagonal A. The HPFBench code tests two situations, with the problem axis being the left and the right parallel axis, denoted as `coef_inst` and `inst_coef`, respectively.

While `cond-grad` uses Conjugate Gradient method to solve tridiagonal systems, the `pcr` benchmark solves the irreducible tridiagonal system of linear equations $AX = B$ using the *parallel cyclic reduction* method [Golub

and van Loan 1989; Hockney 1965; Hockney and Jesshope 1988], which performs the reduction and obtains the solution in one pass. Parallel implementation issues are discussed in Johnsson [1985] and Johnsson and Ho [1990]. The communication consists of circular shifts with offsets being consecutive powers of two and implemented by the intrinsic *CSHIFT*.

Like matrix-vector multiplication, few operations per data element are performed. Each element in each solution vector is updated $\log n$ times before its final value is available. However, only a few operations are performed on an element after each *CSHIFT* communication, regardless of n, and no replication or reduction is performed. Tridiagonal solvers expose communication overhead to a much greater extent than dense matrix-vector multiplication. In the latter case, the number of operations per communicated element scales as $O(\sqrt{M})$, where M is the number of elements of each submatrix residing on each processor [Johnsson et al. 1989].

3.1.8 *QR Factorization and Solution (qr).* This benchmark solves dense linear systems of equations using Householder transformations [Dahlquist et al. 1974; Golub and van Loan 1989]. Given an $m \times n$ coefficient matrix A, where $m \geq n$, and a set of r right-hand-side vectors in the form of an $m \times r$ matrix B, the QR routines factorize and solve the system of equations $AX = B$. The matrix A is factored into an orthogonal matrix Q and an upper triangular matrix R, such that $A = QR$. Then, the solver uses the factors Q and R to calculate the least squares solution to the system $AX = B$, i.e., to compute the set of r vectors in array X (each corresponding to a particular right-hand side). The HPFBench version of the QR routines only supports single-instance computation and performs the Householder transformations *without* column pivoting.

Both the factorization and the solution routines make use of masks. An alternative would be to use array sections. Whichever approach yields the highest performance and requires the least memory depends upon how the compiler handles masked operations, the penalty for carrying out operations under masks, and how array sections are implemented, in particular with respect to temporary storage.

The communication patterns appearing are *SPREAD*s and reductions. The reductions are performed within the intrinsic function *SUM*. In the solution routines, the right-hand-side matrix B is aligned with A through assignment to another array (`rhs`) of the same shape as A. Therefore, the misalignment overhead is reduced, at the expense of additional memory space. Due to this alignment, $n \geq r$ must hold. Thus the matrix sizes must satisfy the inequality $m \geq n \geq r$.

3.2 Application Kernels

The application kernel benchmarks are intended to cover computations (including communication) that dominate the running time of a wide

variety of scientific applications frequently implemented on scalable architectures. We characterize these benchmarks to assess their performance according to some inherent properties that inevitably dictate their computational structure and communication pattern.

Single-node performance is at least as important as the communication related performance on most architectures. The HPFBench application kernels contain one "embarrassingly parallel" code, `gmo`, which does not include any interprocessor communication.

A large number of production codes are based on structured (regular) discretizations of space or time. Many solution methods imply data references along the axes of grids resulting from such discretizations. For methods of relatively low order with respect to accuracy, the data references associated with updating variables at grid points are confined to neighborhoods that extends one or two grid points in all directions. For computations of this nature, the efficient support of communication as defined by the grid is often crucial. The HPFBench application kernels contain codes that depend on the emulation of grids distributed across processors as follows: one-dimensional grids: `wave-1d`; two-dimensional grids: `boson`, `ellip-2d`, and `step4`; three-dimensional grids: `diff-3d`, `rp`, and `mdcell`; four-dimensional grids: `qcd-kernel`.

Since the geometries involved are often complex, unstructured (irregular) grids are the most common form of spatial discretizations in engineering applications, and the finite-element method is a common solution technique. The collection of elements is typically represented as a list of elements with arrays describing the connectivity or adjacency. The `fem-3d` HPFBench benchmark is intended to cover some of the aspects of this type of codes. This benchmark uses an iterative solver for the equilibrium equations. The execution time is dominated by matrix-vector multiplication, which is in turn dominated by *gather* and *scatter* operations on the unstructured grid (unassembled stiffness matrix).

In addition to codes solving field problems for a variety of domain shapes and media, many computations also involve discrete entities, like particles. The so-called *N-body* codes solve field equations based on particle-particle interactions with or without spatial discretizations. Traditional N-body codes do not use a spatial discretizations and have an arithmetic complexity of $O(N^2)$. The so-called *particle-in-cell* (pic) codes make explicit use of a grid for long-range interaction and require the representation of both particle attributes and grid-point data. Hierarchical N-body codes of arithmetic complexity $O(N \log N)$ or $O(N)$ construct a hierarchy of grids.

The HPFBench benchmark suite contains two codes that carry out direct interaction between all particles: `md` for particle interaction in three-dimensional space, and `n-body` for particle interaction in two-dimensional space.

The HPFBench codes involving both grid and particle representations are `pic-gather-scatter` and `mdcell` (with three-dimensional grids) as well as `pic-simple` (with a two-dimensional grid).

A particularly important computational kernel is the FFT, for which efficient implementations appear in many libraries. The HPFBench suite includes three application kernels that make extensive use of the FFT: `ks-spectral` and `pic-simple` perform FFTs on two-dimensional grids, while `wave-1d` performs FFTs on a one-dimensional grid.

Several benchmarks (`pic-simple`, `qmc`, `qptransport`, and `wave-1d`) rely on a random-number generator for problem initialization. Although Fortran 90 provides an intrinsic function (*RANDOM NUMBER*) for random-number generation, the sequence the intrinsic function generates seems to vary with different F90 compilers. To ensure the same problem is solved for these benchmarks under different HPF compilers and on different platforms, we provide our random-number generator wrapper as an HPFBench utility function. The wrapper calls the UNIX C library function `drand48()` which is both compiler- and architecture-independent. One consequence of this change is that initializing a distributed array will be serialized. Since a random-number generator is only called during initialization, we consider this serialization to be acceptable.

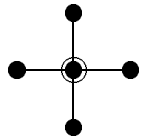
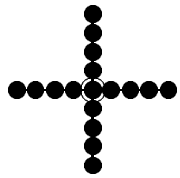
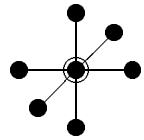
The above characterization of the application kernels is summarized in Table VIII. Table IX lists the data representation and layout for the arrays used in the dominating computations in the application kernels, and Table X summarizes the communication patterns in the codes. Table XI tabulates the implementation techniques for the stencil, gather/scatter, and AABC communication patterns, whereas Table XII lists the computation-to-communication ratio for the main loop of application codes, as well as the memory usage.

3.2.1 *Boson: Many-Body Simulation (`boson`).* This benchmark performs quantum many-body computations for bosons on a two-dimensional lattice using a grid-based Monte Carlo technique. The code uses a Cartesian lattice with periodic boundary conditions and uniform site connectivity resulting in stencil communication. The algorithms are outlined and demonstrated in Batrouni and Scalettar [1992], Hirsch et al. [1982], and Tobochnik et al. [1992].

The implementation uses 3D arrays, with the first axis representing the time axes, and the other two axes representing the two spatial axes of the grid. Nearest-neighbor *CSHIFT*s are heavily used along the spatial axes. The *CSHIFT*s are interleaved with computations in a way that precludes the use of a linear stencil formulation as well as the use of *polyshift* (*PSHIFT* [George et al. 1994]). The main computations are scalar operations among 2D parallel arrays, i.e., the 3D arrays with the time axes locally (and sequentially) indexed.

3.2.2 *3D Diffusion Equation: Explicit Finite Difference (`diff-3d`).* This diffusion equation simulation is the integration of the three-dimensional heat equation using an explicit finite difference algorithm, which is stable subject to the Courant condition $dt < dx^2/D$, where D is the diffusivity. The benchmark code fully exploits the fact that the stencil

Table X. Communication Patterns in the Application Kernels

Communication Pattern	Arrays			
	1D	2D	3D	(4,5,6,7)-D
Stencil	wave-1d 	ellip-2d  step4 	rp, diff-3d 	
Gather		pic-simple	fem-3d pic-gather-scatter	
Scatter	qptransport		pic-gather-scatter	mdcell
Scatter w/combine		qmc	fem-3d pic-gather-scatter	
Reduction	qptransport	ellip-2d ks-spectral md qmc rp		
Broadcast		ellip-2d,md n-body,rp		qmc
AABC		md, n-body		
Butterfly (FFT)	wave-1d	pic-simple ks-spectral		
Scan	qptransport	qmc	pic-gather-scatter	
Cyclic shift	wave-1d	ellip-2d n-body step4	boson rp	mdcell qcd-kernel
Sort	pic-gather-scatter qptransport			

coefficients are the same for all grid points (constant coefficients). Thus neither left-hand-side nor right-hand-side matrices are stored explicitly in

Table XI. F90/HPF Constructs Used in Implementing Some Common Communication Patterns in the Application Kernels

Communication Pattern	Code	Implementation Techniques
Stencil	boson	CSHIFT
	ellip-2d	CSHIFT
	mdcell	CSHIFT
	rp	CSHIFT
	wave-1d	CHSIFT
	step4	chained CSHIFTs
	diff-3d	Array sections
Gather	fem-3d	FORALL with indirect addressing
	pic-gather-scatter	FORALL with indirect addressing
	pic-simple	FORALL with indirect addressing
Scatter	mdcell	INDEPENDENT DO w/indirect addressing
	pic-gather-scatter	FORALL with indirect addressing
	qptransport	indirect addressing
Scatter w/combine	fem-3d	HPF sum_scatter library procedure
	pic-gather-scatter	HPF sum_scatter library procedure
	qmc	HPF copy_scatter library procedure
AABC	md	SPREAD
	n-body	CSHIFT, SPREAD, scalar to array assignment

this benchmark. Only the solution variables $F(x, y, z, t)$ are stored in array form for one time step as a 3D array of shape $n_x \times n_y \times n_z$.

The operations in this benchmark are dominated by the evaluation of a seven-point centered difference stencil in three dimensions. The communication is implemented by array sections. Instead of array sections, *CSHIFT*s could have been used. This implementation technique is used in `ellip-2d`, which evaluates five-point stencils in two dimensions, and the `rp` benchmark which also performs a seven-point centered stencil in three dimensions, just as `diff-3d`. Stencil evaluations are interesting from a compiler evaluation perspective because they occur frequently in a variety of application codes, and there are many opportunities for optimization of data motion between processing elements, and between memory and registers. Some of these issues, related to the data-parallel programming paradigm, are explored in Hu and Johnsson [1996], while some compiler techniques are explored in Brickner et al. [1993].

This benchmark is an example of applications with structured Cartesian grids, solving homogeneous linear differential equations with constant boundary conditions.

3.2.3 *Solution of Poisson's Equation by the Conjugate Gradient Method (`ellip-2d`).* This benchmark uses the preconditioned Conjugate Gradient method to solve Poisson's equation on a regular two-dimensional grid

Table XII. Computation-to-Communication Ratio and Memory Usage in Main Loop of Application Kernels. In applications involving structured grids, n_x, n_y, n_z denote the number of mesh points along x-, y-, and z-axes. In particle simulation applications, n_p denotes the number of particles. n_t denotes the extent of time axis in quantum dynamics applications `boson` and `qcd`. In `fem-3d`, n_{ve} and n_e denote the number of vertices per element and the number of elements in an unstructured mesh. Problem sizes in `gmo` and `qmc` are more involved and are detailed in the code.

Code	FLOP Count (per iteration)	Memory Usage (in bytes)	Communication (per iteration)
boson	$4(258 + 36/n_t) \cdot n_t n_x n_y$	$40n_x n_y + 128n_t + 12000 + 4000m_b + 1536n_t n_x n_y$	38 CSHIFTs
diff-3d	$9(n_x - 2) \cdot (n_y - 2)(n_z - 2)$	$8n_x n_y n_z$	1 7-point Stencil
ellip-2d	$38n_x n_y$	$96n_x n_y$	4 CSHIFTs, 3 Reductions
fem-3d	$18n_{ve}n_e$	$56n_{ve}n_e + 140n_v + 1200n_e$	1 Gather, 1 Scatter w/combine
gmo	$6000n_{vec}$	$2n_{vec} \cdot (8 \cdot \mathrm{ns}_{out} \cdot (\mathrm{ntr}_{out}+1) + 8 + 8 \cdot n_{vec})$	N/A
ks-spectral	$(76 + 40\log_2 n_x) \cdot n_x n_y$	$144n_x n_y$	8 1D FFTs on 2D arrays
mdcell	$(101 + 392n_p) \cdot n_p n_x n_y n_z$	$(184 + 160n_p) \cdot n_x n_y n_z$	195 CSHIFTs, 7 Scatters on local axis
md	$(23 + 51n_p)n_p$	$160n_p + 80n_p^2$	6 1D to 2D SPREADs, 3 1D to 2D sends, 3 2D to 1D Reductions
n-body			
Broadcast	$17n_p^2$	$72n_p$	3 Broadcasts
SPREAD	$17n_p^2$	$72n_p$	3 SPREADs, 3 SPREADs
CSHIFT	$17n_p(n_p - 1)$	$72n_p$	3 CSHIFTs, 3 CSHIFTs
CSHIFT-sym.	$13.5n_p(n_p - 1)$	$96n_p$	3 CSHIFTs

with Dirichlet boundary conditions. The Poisson's equation is discretized with a centered five-point stencil. The matrix-vector product in the Conjugate Gradient algorithm [Dahlquist et al. 1974; Golub and van Loan 1989; Press et al. 1992] takes the form of a stencil evaluation on the two-dimensional grid. In addition, one reduction and one broadcast is required for the Conjugate Gradient method.

Compared to `diff-3d`, this HPFBench benchmark is dominated by the evaluation of a two-dimensional stencil instead of a three-dimensional stencil. Hence, for a given subgrid size, the optimum computation-to-communication ratio (volume/surface) is higher for `ellip-2d`. However, the inner products and broadcast require some communication, but it can

Table XII. *Continued*

Code	FLOP Count (per iteration)	Memory Usage (in bytes)	Communication (per iteration)
pic-simple	$n_p + 15n_xn_y \cdot (\log n_x + \log n_y)$	$64n_p + 72n_xn_y$	1 Scatter w/ add 1D to 2D, 3 FFT, 1 Gather 3D to 2D
pic-gather-scatter	$270\ n_p$	$12n_xn_yn_z + 88n_p$	81 Scans, 27 Scatters w/add, 27 1D to 3D Scatters, 27 3D to 1D Gather
qcd-kernel	$606n_xn_yn_zn_t$	$720n_xn_yn_zn_ti$	4 CSHIFTs
qmc	$[(42 + 2n_on_{maxw}) \cdot n_pn_dn_wn_e + (142n_o + 251) \cdot n_wn_e]n_b$	$16n_pn_d + 96n_wn_e$	n_{maxw} SPREADs 3D to 1D, 5 Reductions 2D to 1D, $(n_pn_d + 4)$ Scans on 2D, $(n_pn_d + 1)$ Sends, 3 Reductions 2D to scalar
qptransport	$34n$	$160n$	10 Scatters 1D to 1D, 1 Sort, 5 Scans, 1 CSHIFT, 1 EOSHIFT, 3 Reductions
rp	$44n_xn_yn_z$	$120n_xn_yn_z$	2 Reductions, 12 CSHIFTs (2 7-point Stencils)
step4	$2500n_xn_y$	$1000n_xn_y$	128 CSHIFTs (8 16-point Stencils)
wave-1d	$29n_x + 10n_x\log n_x$	$64n_x$	12 CSHIFTs, 2 1D FFTs

be made very efficient. The three dominating operations with respect to performance on many scalable architectures in this benchmark are the reduction (*SUM*), broadcast, and the evaluation of the five-point centered difference stencil.

This benchmark is an example of a class of applications with structured grids, involving the solution of linear nonhomogeneous differential equations with Dirichlet boundary conditions by means of an iterative solver. From a performance point of view, like `diff-3d`, this benchmark is interesting, since it discloses how efficiently stencil communication and evaluation are handled by the compiler.

3.2.4 *Finite-Element Method in Three Dimensions (`fem-3d`).* This benchmark uses the finite-element method on trilinear brick elements to solve the equilibrium equations in three dimensions. The mesh is represented as an unstructured mesh. Specifically, this code evaluates the elemental stiffness matrices and computes the displacements and the stresses at the quadrature points using the Conjugate Gradient method on unassembled stiffness matrices [Johnson 1987; Johnson and Szepessy 1987].

The unstructured mesh is read from a file and stored in the element nodes array. A partitioning of the unstructured mesh is performed in an attempt to minimize the surface area (communication) for the collection of elements stored in the memory of a processor. The partitioning is carried out by Morton ordering (implemented in HPF as a general partitioning method in Hu et al. [1997]) of the elements using the coordinates of the element centers. The array of pointers derived from the unstructured mesh is reordered so that the communication required by the subsequent gather and scatter operations is reduced. The nodes are also reordered via Morton ordering. The nodes of the mesh are stored in a 2D array whose first axis is local to a processing element. It represents the coordinates of each node. The computations of the benchmark are dominated by the sparse matrix-vector multiplication required for the Conjugate Gradient method. It is performed in three steps: a gather, a local elementwise dense matrix-vector multiplication, and a scatter. The elementwise matrix-vector multiplication exploits the fact that the elemental stiffness matrices are symmetric.

The `fem-3d` is a good example of finite-element computations on unstructured grids. In order to preserve locality of reference and allow for blocking, and the use of dense matrix optimization techniques, finite-element codes are often based on unassembled stiffness matrices. Moreover, by exploiting symmetry in the elemental stiffness matrices, this code is a good test case for compiler optimizations with respect to local memory references for a somewhat more complex situation than a dense nonsymmetric matrix. Through the use of the reordering of the mesh points and nodes to improve locality, the gather and scatter operations are expected to be efficient.

3.2.5 *Seismic Processing (`gmo`).* This benchmark is a highly optimized HPF code for a *generalized moveout* seismic kernel for all forms of Kirchhoff migration and Kirchhoff DMO (also known as $x - t$ migration and $x - t$ DMO). For each vector unit, the code explicitly strip-mines the main computational loop into vector chunks of length *nvec*. In order to perform the strip-mining, the indices `isamp`, `ksamp`, and the real variable `del` have a local axis of extent equal to the vector length *nvec*. `independent do` loops are used to express each calculation in the loop for one whole vector chunk and across all the vector units. An outer do-loop steps through all the vector chunks sequentially.

The `gmo` benchmark is a good test case of a compiler's local memory management. The code is written explicitly with efficient local memory management in mind.

3.2.6 *Spectral Method: Integration of Kuramoto-Sivashiniski Equations (`ks-spectral`).* This benchmark uses a spectral method to integrate the Kuramoto-Sivashiniski equations on a two-dimensional regular grid. The two-dimensional grid is stored in a two-dimensional array with block distribution. The code performs the integration using a fourth-order Runga-Kutta integration of the equation $dX/dt = f(X, t)$ with step size dt. Within an integration step, the core computation is a 2D FFT performed on

the grids. In fact, all the communication in the benchmark happens inside the 2D FFT.

3.2.7 *Molecular Dynamics, Lennard-Jones Force Law (md, mdcell).* Many molecular dynamics codes make use of the Lennard-Jones force law for particle interaction. With few exceptions, this force law is applied with a cut-off radius. A grid is superimposed on the domain of particles, and the cell size of the grid is chosen such that the cut-off radius is smaller than the cell size. This guarantees that interaction only involves particles in adjacent cells. Furthermore, if the cell size is only moderately larger than the cut-off radius, then, for simplicity in implementation, all particles in adjacent cells are assumed to interact, and no separate interaction lists are maintained. The loss of efficiency is modest. The HPFBench `mdcell` code is designed with these assumptions. Accurate modeling of electrostatic forces does, however, require long-range interaction between particles, in which case the radius cannot be cut off. Thus, molecular dynamics codes that attempt to include electrostatic forces must evaluate interactions between all particles, i.e., such codes have many aspects in common with general N-body codes. The HPFBench code `md` carries out an evaluation of interactions between all particles using a Lennard-Jones potential without cut-off. The `mdcell` and the `md` codes both use a Maxwell initial distribution and a Verlet integration scheme. Both benchmarks make heavy use of parallel indirection.

—*Local forces only:* In the `mdcell` code particles only interact with particles in nearby cells; thus no neighbor tables are required. The benchmark assumes periodic boundary conditions and initializes particles with the Maxwell distribution.

The particles are initialized and manipulated in 4D arrays, i.e., the first axis is local and is used for storing particles in the same cell, while the other axes are parallel, representing the 3D grid of cells. Utilizing the symmetry, each cell only needs to fetch directly 13 neighbor cells instead of all 26 in the neighbor-cell interaction. The fetching of neighbor cells is performed via *CSHIFT*. No linear ordering [Lomdahl et al. 1993] is imposed on the order of fetching the 13 neighbors; therefore the number of *CSHIFT*s is not minimized. The migration of particles are implemented by slicing through the 4D particle arrays and copying particles that need to move to different cells out to a newly allocated slice of the 4D array, followed by a scatter operation within the local axis of the 4D array to squeeze out the "bubbles" in the array. Since particle can at most move to a neighboring cell, the copying operation is implemented using a *CSHIFT*.

—*Long-range forces:* The `md` code carries out a Lennard-Jones force law without cut-off. The interaction is evaluated directly through an $O(N^2)$ method. The all-to-all communication is implemented with a send-spread-reduce algorithm. Each particle attribute is implemented as a 1D array. For instance, particle positions are implemented as three 1D

arrays with a location for each particle. The all-to-all communication makes use of 2D temporary $n \times n$ arrays, where n is the number of particles. The particle information is spread across both dimensions using *SPREAD* operations.

3.2.8 *A Generic Direct N-Body Solver, Long-Range Forces (`N-Body`).* This benchmark consists of a suite of two-dimensional N-body solvers all of which directly compute all pairwise interactions [Hockney and Eastwood 1988]. The different codes in the suite allows for a comparison of different ways of programming the all-to-all communication. The different ways of programming the direct method are described in detail with code listings in Greenberg et al. [1992]. The data structures consist of 2D arrays with the first dimension local to a node, holding the particle information, and the second parallel, representing the number of particles. The all-to-all communication for the direct method is implemented with four different communication patterns:

(1) Broadcast, implemented as an assignment of a single array element to an array,

(2) *SPREAD*, implemented using *SPREAD*,

(3) *CSHIFT*, implemented using *CSHIFT*,

(4) *CSHIFT*, implemented using *CSHIFT*, and exploiting symmetry.

3.2.9 *Particle-in-Cell Codes (`pic-simple, pic-gather-scatter`).* The HPFBench benchmark suite contains two codes for particle-in-cell methods. Both codes represent a structured grid in the form of 3D arrays, but differ in their representation of particle attributes. The `pic-simple` code implements a two-dimensional spatial grid with a third local axis for vector values at the grid points and a 2D array for particles, with the attributes for each particle forming a local axis. The `pic-gather-scatter` code implements a three-dimensional spatial grid for a scalar field, while each particle attribute is represented by its own 1D array.

The long-range particle interaction in particle-in-cell codes is determined by solving field equations on the grid by some efficient method, like the FFT, interpolating the field to particles in a cell, moving the particles, and then projecting back to the grid points. The `pic-simple` code scatters field data from the particles to the grid with *send-with-add*, whereas the `pic-gather-scatter` code accomplishes this task with both *send-with-add* and *sort-scan-with-add*.

—*Straightforward implementation:* The `pic-simple` benchmark represents a two-dimensional electrostatic plasma simulation. It is meant to be a naive implementation of the method, representative of the sort of parallel code that users would write on their first attempt.

The particle information is represented by 2D arrays, with the first axis being local to a processing element and storing the relevant attributes for each particle. Initially, the particles are placed randomly according to a

uniform distribution inside a square box of shape $c_r \times c_r$. This box is centered in the domain. Each particle is given a random velocity in the range $[-c_v/2, c_v/2]$ along the x- and y-axes, which is modified with a component normal to, and proportional to, the vector from the box center to the particle. The magnitude of this modification vector is c_l. The two-dimensional grids are represented by 3D arrays, again with the first axis being local and representing the 2D vector field at each gridpoint.

The computations consist of scattering the charge data from the particles to the grid points using *sum_scatter*. Charge data sent to the same gridpoint are combined on-the-fly, and no sophisticated interpolation scheme is used. Then, a two-dimensional FFT is used to solve Poisson's equation for the electrostatic field on the two-dimensional grid. This is followed by invoking a *gather* operation to determine the field at the particles. Then, the particle positions and velocities are advanced in time, using a leap-frog method.

—*Sophisticated implementation:* The `pic-gather-scatter` benchmark tests the gather/scatter operations between data structures for particles and for cells in a typical 3D particle-in-cell application. The solver for the field equations on the grid is *not* included. Two different techniques for particle-grid interactions are used: *send-with-add* and *sort-scan-send*. The sort-scan-send version avoids congestion, which occurs in the send-with-add version. The sort-scan-send version also uses a more sophisticated interpolation function in distributing the charges (masses, etc.) to the grid.

The particle positions are stored in a 2D array, with the first axis being local and representing the three coordinates of a particle position. The field at the grid points is stored in 3D arrays, the first axis of which is local while the other two are parallel. Hence, this layout does not minimize the surface to volume ratio.

The benchmark makes extensive use of HPF library code for many communication and the sorting functions, including *sum_scatter*, *sum_prefix*, *copy_prefix*, and *grade_up*.

The scan-and-send sorts particles into canonical ordering using the concatenated coordinates of a particle, i.e., $(z|y|x)$, as its key. This ordering differs from the ordering of the cells and gridpoints in that the address space for the cells and grid points is three-dimensional. Using a coordinate-sort for the particles as described in Hu and Johnsson [1996] should improve the locality in the send and get operations between the 1D particle arrays the 3D grid arrays.

3.2.10 *QCD Kernel: Staggered Fermion Conjugate Gradient Method (`qcd-kernel`).* This benchmark represents the kernel of a staggered fermion Conjugate Gradient algorithm for Quantum Chromo-Dynamics (QCD). The operation performed is

$$r(x) = mc + \sum_{\mu=1}^{4} [u(x)^{\dagger} c(x + \mu) - u(x - \mu)c(x - \mu)]$$

where all the entities are double complex, except m which is a double constant. r and c represent three-element "vectors" (the three colors), and u represents a 3×3 operator between colors; † denotes the adjoint operator. The aforementioned operation is carried out on a four-dimensional lattice of shape $n_x \times n_y \times n_z \times n_t$ (three spatial and one time dimension). The integer μ loops over the four dimensions of the lattice, so $x \pm \mu$ is a neighboring lattice point along dimension μ. This method is described in detail in Aoki et al. [1991], which is also an excellent source of bibliography on the subject.

The `qcd-kernel` benchmark is implemented as a multiple-instance code, with 5D arrays for lattice data, or more for nonscalar fields. The kernel is highly optimized in the following ways:

—All the complex entities are declared as a pair of real values (real and imaginary parts separately) to avoid operations on complex data types (and relying on the compiler to translate them).

—The 3×3 operator u is represented as a vector of length nine, so that both u and its transpose can be accessed without any extra space or mapping overheads.

—Matrix operations are calculated element by element by straight-line code; the only loop overhead is the sequential loop for μ over the four dimensions.

—Only the five dimensions for the multiple-instance lattice (four dimensions for space and time and one dimension for the n_{ins} instances) are parallel; other dimensions are local. Thus, the only communication is the circular shifts $x \pm \mu$.

The `qcd-kernel` benchmark is interesting as an example of how efficient local arithmetic can be carried out on small matrices, and it is a good reference for compiler optimizations local to a node.

3.2.11 *Quantum Monte-Carlo (`qmc`).* This benchmark evaluates the ground-state energy for two fermions confined in a one-dimensional square potential well using a Green's function quantum Monte Carlo method. The basic algorithm consists of moving random walkers in configuration space (two-dimensional in this model problem) by sampling their steps from the Green's function for the Helmholtz equation, and multiplying their weights by a factor that is a function of configuration space position. For a thorough description of the algorithmic methodology see Anderson et al. [1991].

To load balance the calculation high-weight particles are split, and low-weight particles are killed at each step. This kill-and-split algorithm is

implemented with *scan*s and sends and is described in Traynor et al. [1991].

The manifestation of the fermion sign problem in this calculation is the tendency of large numbers of walkers of opposite sign to collect in the same region of configuration space. To mitigate this problem, a walker-cancellation scheme is used, as described in Anderson et al. [1991]. This amounts to an N-body problem on the walkers, and it is implemented using the sequential spread algorithm every n_c (cancellation number) iteration steps. Thus, by making n_c large, this code is a good test of the *scan-and-send* load-balancing algorithm. By making n_c small, it is a good test of the sequential *SPREAD* N-body algorithm. Both limits are examined in this test suite.

The main data structures of the code are 4D arrays, with the first two axes being local to a node and representing a walker, and the other two axes being parallel, representing the independent ensembles of walkers. Most computations, however, are performed on 2D arrays corresponding to the two parallel axes of the 4D arrays. The communication includes *reductions* along one axis of these 2D arrays, *scatter* with combine on 2D arrays implemented via *copy_scatter*, and *scan* operations by calling the HOF intrinsic functions *copy_prefix* and *sum_prefix* along one dimension of the 2D arrays.

3.2.12 *Quadratic Programming Problem (`qptransport`).* This benchmark is an optimization code for finding optimum paths on a bipartite graph with a quadratic cost function. The benchmark generates random sparse quadratic transportation problems and solves them using an unusual alternating direction method. Each problem is based on a large, sparse, bipartite graph. The graph vertices are either "sources" (with an exogenous supply of some commodity) or "sinks" (with a demand). The edges of the graph connect the sources and sinks, and each one has a quadratic cost function. The problem is to route the commodity from the source nodes to the sink nodes so as to exactly use all of the supplies and satisfy all demands with the minimum possible total cost. The example code generates problems, solves them, and checks that the solutions seem approximately correct. However only the solution phase is timed. It represents the graph as one long vector with many short segments of average length 16.

The code performs very few FLOPs and is dominated by many rank, *scan*, and scatter operations on 1D arrays by calling the HPF library functions, *grade_up*, *sum_prefix*, *min_prefix*, *copy_prefix*, and indirect addressing. It is believed that *rank* and *scan* operations that could recognize and take advantage of all the short segments being short would substantially speed up this benchmark.

3.2.13 *Solution of Nonsymmetric Linear Equations Using the Conjugate Gradient Method (`rp`).* This benchmark solves a nonsymmetric linear system of equations that result from seven-point centered difference approximations of the differential operators in the following equation:

$$\frac{d^2u}{dx^2} + \frac{d^2u}{dy^2} + \frac{d^2u}{dz^2} + d(x)\frac{du}{dx} + 100.0\frac{du}{dy} + 100.0\frac{du}{dz} = F$$

where $d(x)$ is a polynomial function of x. F is never explicity specified. Instead, a solution vector u is set up as a vector of all ones, and then the right-hand-side vector is derived from the linear system of equations. The PDE assumes Dirichlet boundary condition, and the domain is descretized by a regular grid in three dimensions. The particular method employed uses the single-node version of Conjugate Gradient accelerated Cimmino row projections [Bramley and Sameh 1992; Cimino 1939]. That is, the Conjugate Gradient method is applied to the normal equations $A^TAx = A^Tb$, where A is a nonsymmetric matrix with seven diagonals. Each unknown in the equation corresponds to a gridpoint in the three-dimensional grid, and the matrix A is represented in seven 3D arrays.

The multiplication A^TAx is implemented as two matrix-vector multiplications, i.e., $A^T(Ax)$. Since the matrix A is represented in seven 3D arrays, each matrix-vector multiplication becomes a seven-point stencil evaluation. The stencil is implemented using *CSHIFT*s, which involve movement of data interior to the subgrid on each processor, as well as actual interprocessor communication of data on the boundary of each subgrid. The local data movement can be eliminated via restructuring the code using the array aliasing feature of CMF [Hu and Johnsson 1999].

This code is dominated by two seven-point centered stencils in three dimensions, global reduction, and broadcast. One of the two stencil evaluations has variable coefficients with a single-source array, while the other stencil evaluation has a different source array for each shift. Stencils with multiple sources are common in the solution of Navier-Stokes equations.

3.2.14 *Explicit Finite Difference in Two Dimensions (`step4`).* This benchmark solves a subsonic Euler flow over a backward-facing step. The method employed uses fourth-order accurate finite difference equations. Artificial viscosity is used to stabilize the numerical method.

The communication pattern for the difference operations involves a 16-point stencil with four stencil points coming from each direction: north, east, south, and west, as shown in Table X. The 16-point stencil is implemented as four four-point stencils, one along each axis. Each four-point stencil is in turn implemented using chained *CSHIFT*s. For boundary points, some stencil points are truncated at the boundary, and fewer stencil points are employed. As a consequence, a combination of third-order accurate (for boundary points) and sixth-order accurate (for interior points) local approximations is used which guarantees fourth-order global convergence [Olsson 1994; 1995a; 1995b].

3.2.15 *Wave Equation (`wave-1d`).* This benchmark simulates the inhomogeneous one-dimensional wave equation using the method of characteristics. It does a spline fit to the function at every time step and makes

Table XIII. HPFBench Benchmark Problem Size A

Benchmark	Problem Size A		
	Problem Size	Mflops	Iteration
conj-grad	2^{17}	3278.4	962
fft-1d	2^{17}	111.4	10
fft-2d	$2^9 \times 2^8$	111.4	10
fft-3d	$2^6 \times 2^6 \times 2^5$	111.4	10
gauss-jordan	$2^9 \times 2^9$	270.3	1
jacobi	$2^8 \times 2^8$	614.2	6
lu: nopivot	$(2^9+1) \times (2^9+1)$	89.5	1
lu: pivot	$(2^9+1) \times (2^9+1)$	89.5	1
matrix-vector(1)	$2^8 \times 2^8$	65.5	500
matrix-vector(2)	$2^8 \times 2^8 \times 2^4$	1048.6	500
matrix-vector(3)	$2^8 \times 2^8 \times 2^4$	1048.6	500
matrix-vector(4)	$2^8 \times 2^8 \times 2^4$	1048.6	500
pcr(2): coef_inst	$2^9 \times 2^9 \times 2^3$	117.4	1
pcr(2): inst_coef	$2^9 \times 2^9 \times 2^3$	117.4	1
qr:factor	$2^9 \times 2^9 \times 16$	671.0	1
qr:solve	$2^9 \times 2^9 \times 16$	873.0	1
boson	$2^5 \times 2^6 \times 2^6$	272.9	4
diff-3d	$2^7 \times 2^7 \times 2^7$	1800.3	100
ellip-2d	$2^9 \times 2^9$	655.4	100
fem-3d	4913×4096	627.6	133
gmo	$256 \times 1500 \times 8$	1541.0	1000
ks-spectral	$2^9 \times 2^8$	228.6	4
md	500	1276.1	100
mdcell	$2^4 \times 2^4 \times 2^4 \times 4$	658.8	4
nbody:bcast	2^{14}	4563.1	1
nbody:cshift	2^{14}	4563.1	1
nbody:cshift-sym	2^{14}	3623.6	1
nbody:spread	2^{14}	4563.1	1
pic-gather-scatter	$2^{18}bodies$, $2^6 \times 2^6 \times 2^6 mesh$	118.0	1
pic-simple	$2^{18}bodies$, $2^6 \times 2^6 \times 2^7 mesh$	160.3	10
qcd-kernel	$4 \times 9 \times 8 \times 8 \times 16 \times 16 \times 2$	1270.9	64
qmc	$2 \times 1 \times 2^{11} \times 2^7$	6491.1	40
qptransport	2^{13}	178.1	40
rp	$2^6 \times 2^6 \times 2^5$	5765.1	1000
step4	$4 \times 2^8 \times 2^7$	819.1	10
wave-1d	2^{17}	104.3	4

heavy use of the FFT. A random-number generator is used for the simulation.

The only data structure in the code is a 1D array. Communication occurs within the computation of the FFT, and in an unbalanced four-point stencil.

3.3 HPFBench Problem Size

Two classes of problem sizes are defined for the HPFBench benchmark suite. Tables XIII and XIV give two problem sizes, Class A and Class B, and the total number of floating-point operations (in millions) performed for the two problem sizes for each of the HPFBench benchmarks. For all applica-

Table XIV. HPFBench Benchmarks Problem Size B

Benchmark	Problem Size B		
	Problem Size	Mflops	Iteration
conj-grad	2^{20}	26227.2	962
fft-1d	2^{20}	1048.6	10
fft-2d	$2^{10} \times 2^{10}$	1048.6	10
fft-3d	$2^7 \times 2^7 \times 2^6$	1048.6	10
gauss-jordan	$2^{10} \times 2^{10}$	2148.5	1
jacobi	$2^9 \times 2^9$	4872.7	6
lu: nopivot	$(2^{10}+1) \times (2^{10}+1)$	717.9	1
lu: pivot	$(2^{10}+1) \times (2^{10}+1)$	717.9	1
matrix-vector(1)	$2^{10} \times 2^{10}$	1048.6	500
matrix-vector(2)	$2^{10} \times 2^{10} \times 2^4$	8388.6	500
matrix-vector(3)	$2^{10} \times 2^{10} \times 2^4$	8388.6	500
matrix-vector(4)	$2^{10} \times 2^{10} \times 2^4$	8388.6	500
pcr(2): coef_inst	$2^{10} \times 2^{10} \times 2^3$	521.1	1
pcr(2): inst_coef	$2^{10} \times 2^{10} \times 2^3$	521.1	1
qr:factor	$2^{10} \times 2^{10} \times 16$	5369.0	1
qr:solve	$2^{10} \times 2^{10} \times 16$	6979.0	1
boson	$2^5 \times 2^7 \times 2^7$	2173.7	4
diff-3d	$2^8 \times 2^8 \times 2^8$	14402.4	100
ellip-2d	$2^{10} \times 2^{10}$	2621.4	100
fem-3d	4913×4096	627.57	133
gmo	$1024 \times 1500 \times 8$	6162.0	1000
ks-spectral	$2^{10} \times 2^9$	998.2	4
md	1372	9603.2	100
mdcell	$2^5 \times 2^5 \times 2^5 \times 4$	5270.4	4
nbody:bcast	2^{15}	18253.6	1
nbody:cshift	2^{15}	18253.6	1
nbody:cshift-sym	2^{15}	14495.1	1
nbody:spread	2^{15}	18253.6	1
pic-gather-scatter	$2^{20} bodies$, $2^7 \times 2^7 \times 2^7 mesh$	471.9	1
pic-simple	$2^{20} bodies$, $2^7 \times 2^7 \times 2^7 mesh$	718.2	10
qcd-kernel	$4 \times 9 \times 16 \times 16 \times 16 \times 16 \times 2$	1270.8	64
qmc	$2 \times 1 \times 2^{13} \times 2^7$	25963.0	40
qptransport	2^{16}	1426.1	40
rp	$2^7 \times 2^7 \times 2^7$	46140.0	1000
step4	$4 \times 2^{10} \times 2^9$	13107.0	10
wave-1d	2^{20}	960.5	4

tions, the problem sizes are chosen so that the total memory requirement would be around 50MB for Class A and 200MB for Class B. Therefore, the Class B problem sizes approach the capacity of the main memory in a single processor of most modern distributed-memory machines. Larger problem sizes will be added in the future when there is a significant increase of the main memory in the processors of parallel machines. For most mesh-based codes, the 200MB total memory requirement of Class B translates into about 2^{20} mesh points, i.e., $2^{10} \times 2^{10}$ in 2D or $2^7 \times 2^7 \times 2^6$ in 3D. For Class A, the 50MB requirement translates into about 2^{17} mesh points.

3.4 Core HPFBench Benchmarks

The large number of benchmarks included in the HPFBench suite strive to cover a wide variety of computational structures and communication patterns found in differnet disciplines. Nevertheless, it can be too strenuous for vendors to evaluate an HPF compiler using the full set of benchmarks. Therefore, we pick a subset of eight benchmarks to form the core HPFBench Suite. The criteria in picking the core benchmarks are to cover as many F90/HPF constructs, array distributions, computational structures, communication patterns, and different implementations of them as possible. The core suite consists of two from the linear algebra subset—`fft` and `lu`—and six from application kernels—`ellip-2d`, `fem-3d`, `mdcell`, `pic-simple`, `pic-gather-scatter`, and `rp`.

4. EVALUATION RESULTS OF PGHPF ON THE IBM SP2

We report on the results of evaluating an industry-leading HPF compiler, *pghpf* , from the Portland Groups, Inc., on the distributed-memory IBM SP2, using all 25 benchmarks in the HPFBench suite. Version 2.2-2 of *pghpf* with compiler switch `-O3 -W0,-O4` was used and linked with `-Mmpl`. To measure the overhead of HPF compilers on a single node, we also measure the performance of the HPF versions of the codes compiled under *pgf90* with the same compiler switchs as above plus `-Mf90` and linked with -rpm1, and the performance of the sequential code compiled using the native Fortran 77 compiler, *xlf*, with compiler flag `-O4`.

Our evaluation was performed on an IBM SP2 with 16 uniprocessor nodes. Each node has a RS6000 POWER2 Super Chip processor running AIX 4.3 at 120MHz and has 128MB of main memory. The nodes are communicating through IBM's MPL library on the IBM SP2 high-performance switch network with a peak node-to-node bandwidth of 150MB/second. All results were collected under dedicated use of the machines.

We use Class A problem sizes of the benchmarks as listed in Table XIII for the evaluation, due to the relative small main memory on each node of the IBM SP2.

4.1 Linear Algebra Functions

4.1.1 *Sequential Performance*. Table XV compares the the performance of the linear algebra benchmark codes compiled using *pgf90* and *pghpf*, respectively, versus that of the sequential versions of the same codes compiled using *xlf*, on a single node of the SP2. For `lu` using *pghpf*, block distribution is used, since using cyclic distribution gives much worse performance than using block distribution on one node and on multiple nodes, as reported in more detail in Section 4.3. Table XV shows that the HPF and the F90 compilers incur significant overhead to the generated code when running on a single processor. Specifically, codes compiled using *pgf90* and *pghpf* are 1.07 to 3.02 times slower than the sequential codes for all benchmarks except `fft-3d`, `lu:nopivot`, `pcr:coef_inst`, and `pcr:inst_coef`. Benchmarks `fft-3d` and `lu:nopivot` under *pgf90* are about

Table XV. Single-Node Performance of the Linear Algebra Kernels

	xlf		*pgf90*	*pghpf*
Code	Time (sec.)	FLOP Rate (Mflops/s)	Time (vs. *xlf*)	Time (vs. *xlf*)
conj-grad	176.7	18.56	1.08	1.17
fft-2d	52.5	2.12	1.07	1.08
fft-3d	68.1	1.65	0.95	1.00
gauss-jordan	28.3	9.49	1.24	1.46
jacobi	54.5	11.27	2.20	1.98
lu: nopivot	23.2	3.88	0.85	1.03
lu: pivot	19.6	4.60	1.14	1.16
matrix-vector(1)	3.62	18.2	3.02	2.92
matrix-vector(2)	57.2	18.3	2.86	2.93
matrix-vector(3)	56.3	18.6	2.90	2.94
matrix-vector(4)	56.4	18.6	2.93	2.93
pcr: coef_inst	25.3	0.57	1.23	0.59
pcr: inst_coef	25.3	0.57	1.22	0.57
qr: factor	83.7	8.01	1.30	1.34
qr: solve	75.3	11.58	1.53	1.47

5% and 15% faster than under *xlf*, respectively. Benchmarks `pcr:coef_inst` and `pcr:inst_coef` are 41–43% faster than their sequential counterparts. A close look at these benchmarks shows that the improvement is due to the default padding (`-Moverlap=size:4`) by *pghpf* along all parallel dimensions of an array which reduces cache conflict misses by from a moderate amount in `fft-3d` and `lu:nopivot` to a significant amount in `pcr:inst_coef` and `pcr:coef_inst`.

To understand the overhead incurred on the codes generated by the *pgf90* and *pghpf* compilers, we measured the time spent on segments of the code that would cause communication when running on parallel nodes, and compared them with those of the sequential codes. Table XVI lists the time breakdowns for the three versions of each benchmark. Table XVI shows that the overhead of the HPF and F90 compilers occurs mainly in code segments corresponding to *cshift*, *spread*, *sum*, and *scatter*. First, *cshift* is up to four times slower when compiled using *pgf90* or *pghpf*, compared with the sequential versions of the codes. This contributed to the longer total time for `conj-grad`, `jacobi`, and `pcr`. Second, *scatter* under *pgf90* and *pghpf* is about nine (for `lu:pivot`) to 44 times (for `fft-2d` and `fft-3d`) slower than under *xlf*. Third, *sum* under *pgf90* and *pghpf* is 2.3 to 14 times slower than under *xlf*, which contributed to the significant slowdown for `matrix-vector(4)` and `qr` under the *pgf90* and *pghpf* compilers. Lastly, *spread* is about 10% faster than *xlf* for all four of the `matrix-vector` benchmarks, but is 1.1 to 2.6 times slower than *xlf* for `gauss-jordan`, `jacobi`, `lu`, and `qr`.

4.1.2 *Parallel Performance*. Figure 1 shows the parallel speedups of the linear algebra benchmarks compiled using *pghpf* on up to 16 nodes of the SP2, using the performance of the sequential codes as the base. Overall,

Table XVI. Breakdown of the Single-Node Time of the Linear Algebra Benchmarks

Code	Breakdown	*xlf*	*pgf90*	*pghpf*
conj-grad	total	176.66	190.45	207.13
	cshift	44.81	61.77	65.57
fft-2d	total	52.5	56.0	56.7
	scatter	1.01	9.31	9.52
	cshift	18.7	15.3	17.7
fft-3d	total	67.6	64.8	68.8
	scatter	2.22	17.3	18.0
	cshift	27.1	11.0	13.5
gauss-jordan	total	28.31	35.18	41.36
	spread	4.72	4.91	5.06
	maxloc	9.65	17.73	18.27
	scatter	0.03	0.03	0.09
jacobi	total	54.5	119.9	107.77
	cshift	14.49	58.57	50.37
	spread	7.93	14.31	15.08
lu: nopivot	total	23.2	19.8	23.8
	spread	9.68	9.91	10.0
lu: pivot	total	19.6	22.3	22.7
	spread	9.49	9.80	10.0
	maxloc	0.05	0.08	0.20
	scatter	0.01	0.44	0.44
matrix-vector(1)	total	3.61	10.9	10.5
	sum	0.61	8.21	7.71
	spread	1.23	1.12	1.14
matrix-vector(2)	total	57.2	164.0	168.0
	sum	10.2	121.0	124.0
	spread	19.5	17.4	17.7
matrix-vector(3)	total	56.3	164.0	166.0
	sum	9.71	122.0	122.0
	spread	18.8	17.0	17.1
matrix-vector(4)	total	56.4	165.0	165.0
	sum	9.91	123.0	122.0
	spread	18.4	17.4	17.1
pcr: coef_inst	total	25.3	31.2	15.0
	cshift	2.38	3.25	3.07
pcr: inst_coef	total	25.3	30.77	14.50
	cshift	2.44	2.63	2.59
qr: factor	total	83.77	108.5	112.4
	sum	5.48	16.11	17.06
	spread	9.99	21.7	10.0
qr: solve	total	75.31	115.5	110.51
	sum	6.91	15.92	20.62
	spread	11.71	30.55	19.86

pghpf achieves a linear speedup for `pcr:inst_coef` and moderate speedups for `conj-grad`, `fft-2d`, `qr:solver`, and `lu:pivot`. But for `jacobi`, `gauss-jordan`, and `matrix-vector(4)`, there is little speedup beyond eight nodes.

To understand the contributing factors that limit the scalability of the benchmarks under *pghpf*, we further measure, for each benchmark, the

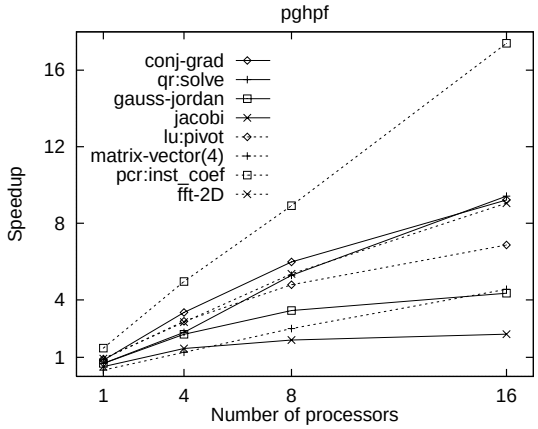


Fig. 1. Speedups of the linear algebra kernels under *pghpf* relative to the sequential performance under *xlf*.

amount of time spent in the different communications for runs on 1, 4, 8, and 16 nodes. Figure 2 shows the measured breakdowns of the total time.

Figure 2 shows that, in general, communications *cshift*, *scatter*, *spread*, and *sum* scale well under *pghpf* in most benchmarks, except for *cshift* in benchmark `pcr:inst_coef`. In this case, *cshift* is performed along the second axis of some two-dimensional arrays with both axes distributed across the nodes. For `jacobi`, *spreads* scales poorly beyond four nodes. This contributed to the poor speedup of the benchmark beyond eight nodes.

4.2 Application Kernels

4.2.1 *Sequential Performance*. Table XVII compares the performance of the application kernel benchmark codes compiled using *pgf90* and *pghpf*, respectively, versus that of the sequential versions of the same codes compiled using *xlf*, on a single node of the SP2. The table shows that, for application kernels, the performance difference of *pghpf* and *pgf90* versus *xlf* is quite mixed. Specifically, half of the benchmarks when compiled under *pgf90* run from 8% to 53% faster than when compiled with *xlf*. The other half of the benchmarks under *pgf90* run between 1.01 to 9.11 times slower than under *xlf*. In general, *pghpf* has higher overhead than *pgf90*. Specifically, eight of 20 benchmarks compiled with *pghpf* run between 3% to 85% faster than under *xlf*. The other 12 benchmarks under *pghpf* run between 1.04 to 4.04 times slower than under *xlf*.

To understand the overhead difference among the three compilers, we further measure the time spent on segments of the code that would cause communication when running on parallel nodes. The comparison (listed in Table XVIII) shows that *gather*/*scatter* and *sum* under *pgf90* and *pghpf* are 2.5 to 23 times slower than under *xlf* except in `mdcell` where *scatter* under *pghpf* and *pgf90* is 10% faster than *xlf* and in `ellip-2d` where *sum* under *pgf90* is 1.6 times faster than under *xlf*. *Sort* under *pgf90* and in *pghpf* is

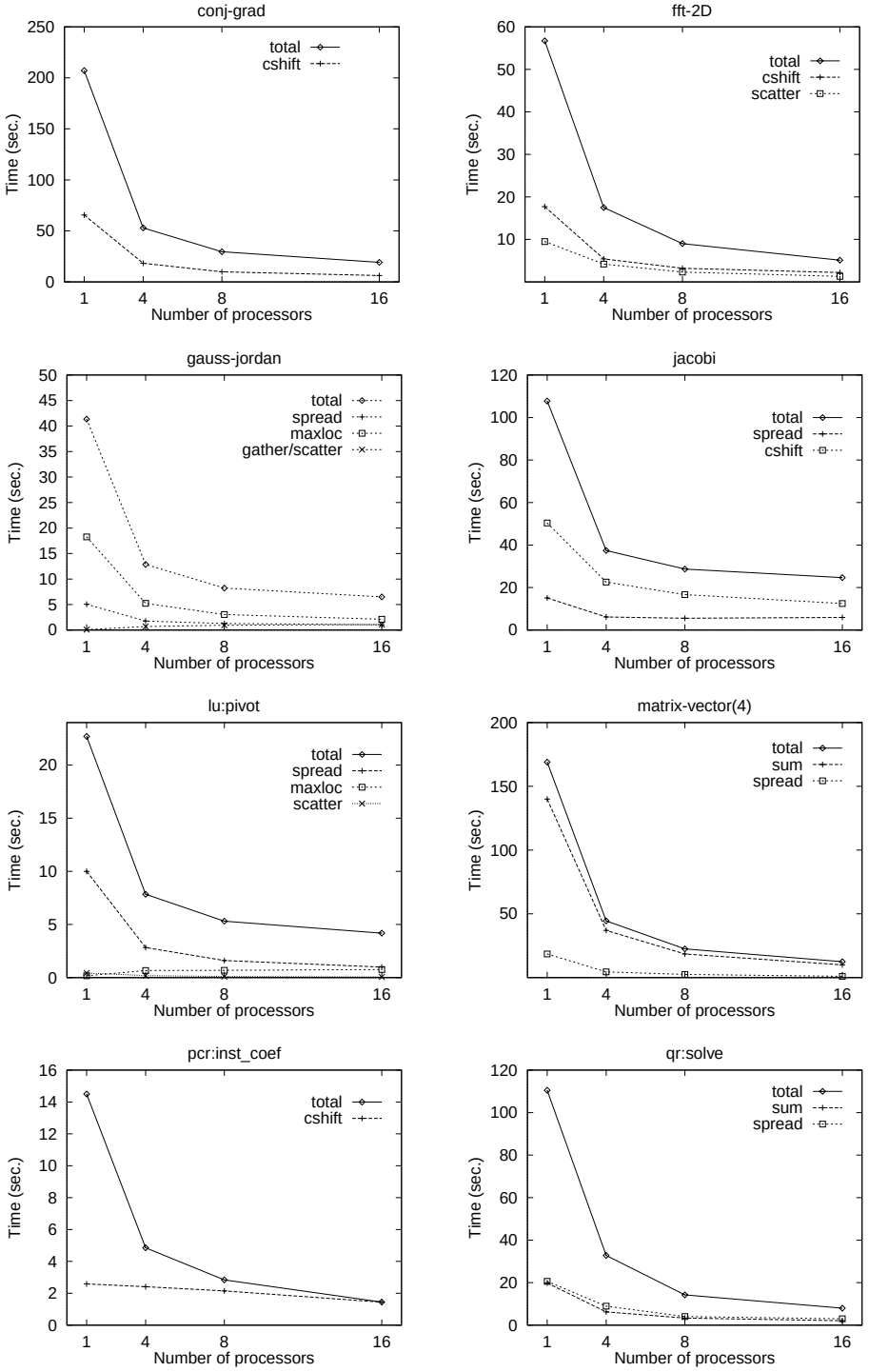


Fig. 2. Total running time and the communication breakdowns for linear algebra benchmarks. For benchmarks with multiple cases only one case is shown.

Table XVII. Single-Node Performance of the Application Kernels

	xlf		*pgf90*	*pghpf*
Code	Time (sec.)	FLOP Rate (Mflops/s)	Time (vs. *xlf*)	Time (vs. *xlf*)
boson	57.8	9.40	0.86	1.04
diff-3d	43.7	41.2	0.47	1.42
ellip-2d	96.5	6.79	0.88	0.15
fem-3d	159.0	3.95	1.01	1.48
gmo	107.0	14.4	0.94	0.99
ks-spectral	116.0	1.97	0.70	0.85
md	49.0	26.0	1.13	1.51
mdcell	86.1	4.62	1.04	1.13
n-body:bcast	100.0	45.5	0.62	0.97
n-body:cshift	98.9	46.1	1.05	0.98
n-body:cshift-sym	75.9	47.7	1.16	0.86
n-body:spread	100.0	45.4	0.78	1.12
pic-gather-scatter	24.6	4.80	2.22	4.03
pic-simple	83.9	1.91	0.92	1.22
qcd-kernel	37.9	33.5	1.11	1.37
qmc	107.0	60.7	1.32	2.10
qptransport	337.0	5.29	2.80	4.04
rp	80.5	71.6	9.11	3.49
step4	56.6	14.5	0.85	0.81
wave-1d	54.9	1.90	0.70	0.90

about 4 times slower than under *xlf* in `qptransport`, but is about the same in `pic-gather/scatter` under all three compilers.

4.2.2 *Parallel Performance*. Figure 3 shows the parallel speedups of the application benchmarks compiled using *pghpf* on up to 16 nodes of the SP2, using the performance of the sequential codes as the base. The applications can be divided into four groups according to their speedups. The first group consists of `ellip-2d`, which achieves better than linear speedup as a consequence of 6.7 times better performance under *pghpf* than under *xlf* with both running on one node. The much better performance with *pghpf* is due much less cache conflicts from padding. The second group consists of `gmo`, which achieves a linear speedup because of no communication. The third group consists of `boson`, `ks-spectral`, `mdcell`, `md`, `pic-simple`, `fem-3d`, `wave-1d`, `n-body:cshift-sym`, `n-body:cshift`, `step4`, `qcd-kernel`, and `qmc`. These benchmarks achieve from moderate to almost linear speedups on up to 16 nodes. The last group consists of `diff-3d`, `pic-gather-scatter`, `n-body:bcast`, `n-body:spread`, `qptransport`, and `rp`. These benchmarks achieve fairly poor speedups. Among these five applications, `qptransport`, `pic-gather-scatter`, and `rp` achieve poor speedups as a result of their poor sequential performance when compared with *xlf*. The poor sequential performance is a result of the poor performance of *gather*/*scatter*, *cshift*, *sort*, and *sum* under *pghpf* than under *xlf*, as shown in Table XVIII.

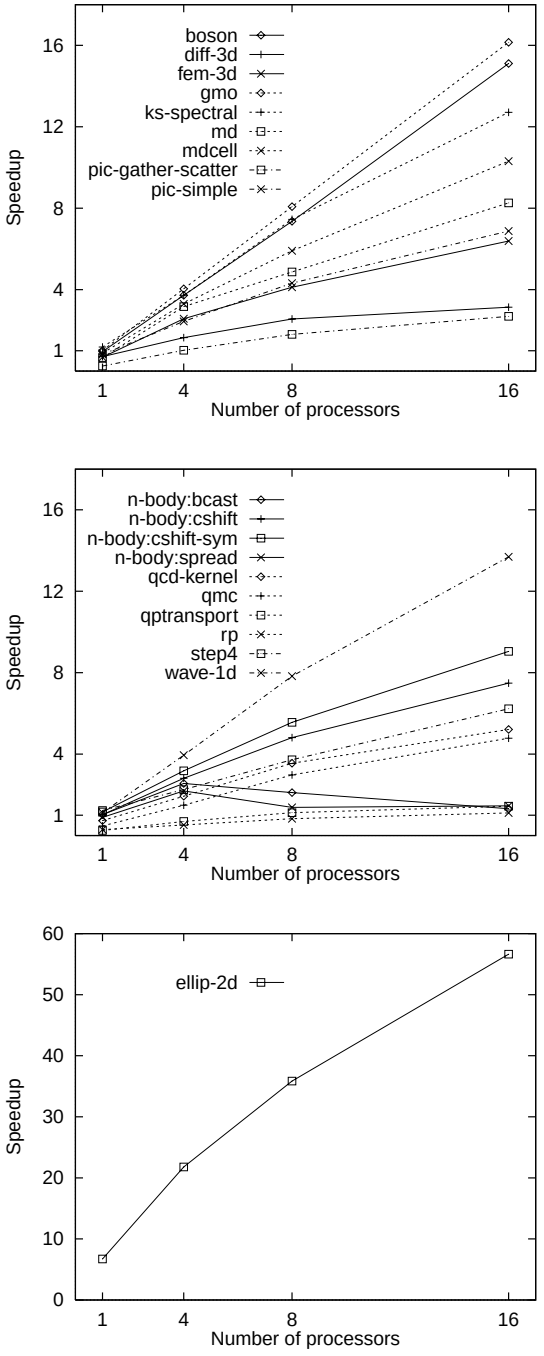


Fig. 3. Speedups of the application kernels under *pghpf* relative to the sequential performance under *xlf*.

Similar to linear algebra benchmarks, we again measure, for each benchmark, the amount of time spent in the different communications for

Table XVIII. Breakdown of the Single-Node Time of the Application Kernel Benchmarks

Code	Breakdown	*xlf*	*pgf90*	*pghpf*
boson	total	57.8	50.1	60.2
	cshift	2.90	9.44	3.57
diff-3d	total	43.7	20.4	61.8
ellip-2d	total	96.5	84.6	14.4
	cshift	87.7	49.5	5.86
	sum	2.46	1.54	2.55
fem-3d	total	159.0	160.0	235.0
	gather	0.77	0.52	41.6
	scatter	3.02	32.7	39.3
gmo	total	107.0	101.0	106.0
ks-spectral	total	116.2	81.6	98.5
	scatter	0.41	6.15	7.46
	cshift	54.9	21.7	32.9
md	total	49.0	55.1	74.1
	spread	5.91	5.53	6.03
	sum	3.11	24.1	34.7
mdcell	total	86.1	89.2	97.7
	cshift	9.03	13.6	15.0
	scatter	2.27	1.99	2.06
n-body:bcast	total	100.0	62.2	96.9
	broadcast	15.2	15.4	15.7
n-body:cshift	total	98.9	103.0	97.2
	cshift	15.5	57.3	15.8
n-body:cshift-sym	total	75.9	88.4	65.2
	cshift	13.0	43.0	13.4
n-body:spread	total	100.0	77.6	112.4
	spread	15.6	31.0	31.3
pic-gather-scatter	total	24.6	54.5	99.0
	sort	0.91	0.68	0.95
	scan	0.93	2.64	3.80
	gather	2.77	7.43	14.32
	scatter	1.06	12.5	18.5

runs on 1, 4, 8, and 16 nodes. Figures 4–6 show the measured breakdowns of the total time. For most applications, communications *cshift*, *gather/scatter*, *sort*, and *sum* scale well under *pghpf* till eight nodes. On the other hand, communications *broadcast* and *spread* achieve slowdown beyond four nodes, as shown in `n-body:bcast` and `n-body:spread`.

4.3 Effect of Data Layouts

We also evaluate the effects of different data layouts on the benchmark performance under the HPF compiler *pghpf*. Specifically, the LU decomposition used in the `lu` benchmark and and QR factorization used in the `qr` benchmark is well known to achieve better load balance with the cyclic array layout than with block array layout. We measure the overall running time as well as the communication time breakdowns for `lu:pivot` and `qr:solve` with three distributions: block, cyclic, and block-cyclic with a block size 4, respectively. The measurements are shown in Figures 7 and 8.

Table XVIII. *Continued*

Code	Breakdown	*xlf*	*pgf90*	*pghpf*
pic-simple	total	83.9	77.3	102.0
	sum_scatter	0.80	4.22	7.41
	copy_scatter	0.31	7.11	7.14
	cshift	32.1	21.2	24.1
	gather	1.11	1.11	19.3
qcd-kernel	total	37.9	41.9	52.0
	cshift	6.00	8.82	9.13
qmc	total	107.0	141.0	225.0
	scan	16.1	11.5	13.4
	scatter	3.71	33.8	87.9
	reduction	0.72	0.80	0.91
qptransport	total	33.7	94.5	136.0
	sort	13.8	55.4	61.5
	scan	4.11	10.9	14.4
	scatter	7.42	18.8	40.4
rp	total	80.5	735.0	282.0
	sum	4.01	67.1	6.01
	cshift	60.5	645.0	262.0
step4	total	56.6	48.0	46.0
	cshift	27.7	30.6	28.4
wave-1d	total	54.9	38.6	49.2
	cshift	16.6	8.66	12.3
	scatter	0.72	1.93	2.84

Overall, The total running time using block-cyclic(4) distribution is about 2 to 5.5 times longer than using block distribution on 1, 4, 8, and 16 nodes for both benchmarks. Using pure cyclic distribution, the total running time is almost identical to that using block distribution for `qr:solve` and is about 2 to 2.5 times for `lu:pivot`. The fact that the performance gap on parallel nodes is consistent with that on one node suggests that the cause of the gap is again due to poor nodal compilation for cyclic distributions.

A close look at the communication time breakdowns shows that block-cyclic(4) distribution loses to block and cyclic distributions mainly in the *spread* communication on one node as well as on parallel nodes. The block-cyclic(4) distribution actually beats the pure cyclic distribution for *maxloc* and *scatter* communications as shown by `lu:pivot`. For `qr:solve`, the pure cyclic distribution only loses to the block distribution on *spread* by a factor of two on parallel nodes.

5. SUMMARY

The HPFBench benchmark suite is a set of High Performance Fortran codes intended for evaluating HPF compilers on scalable parallel architectures. The codes contain new constructs of Fortran 90 and HPF. The benchmarks cover several aspects of the array syntax of Fortran 90 and HPF, scientific software library functions, and application kernels that reflect the computational structure and communication patterns in typical scientific applications, particularly fluid dynamic simulations, fundamental

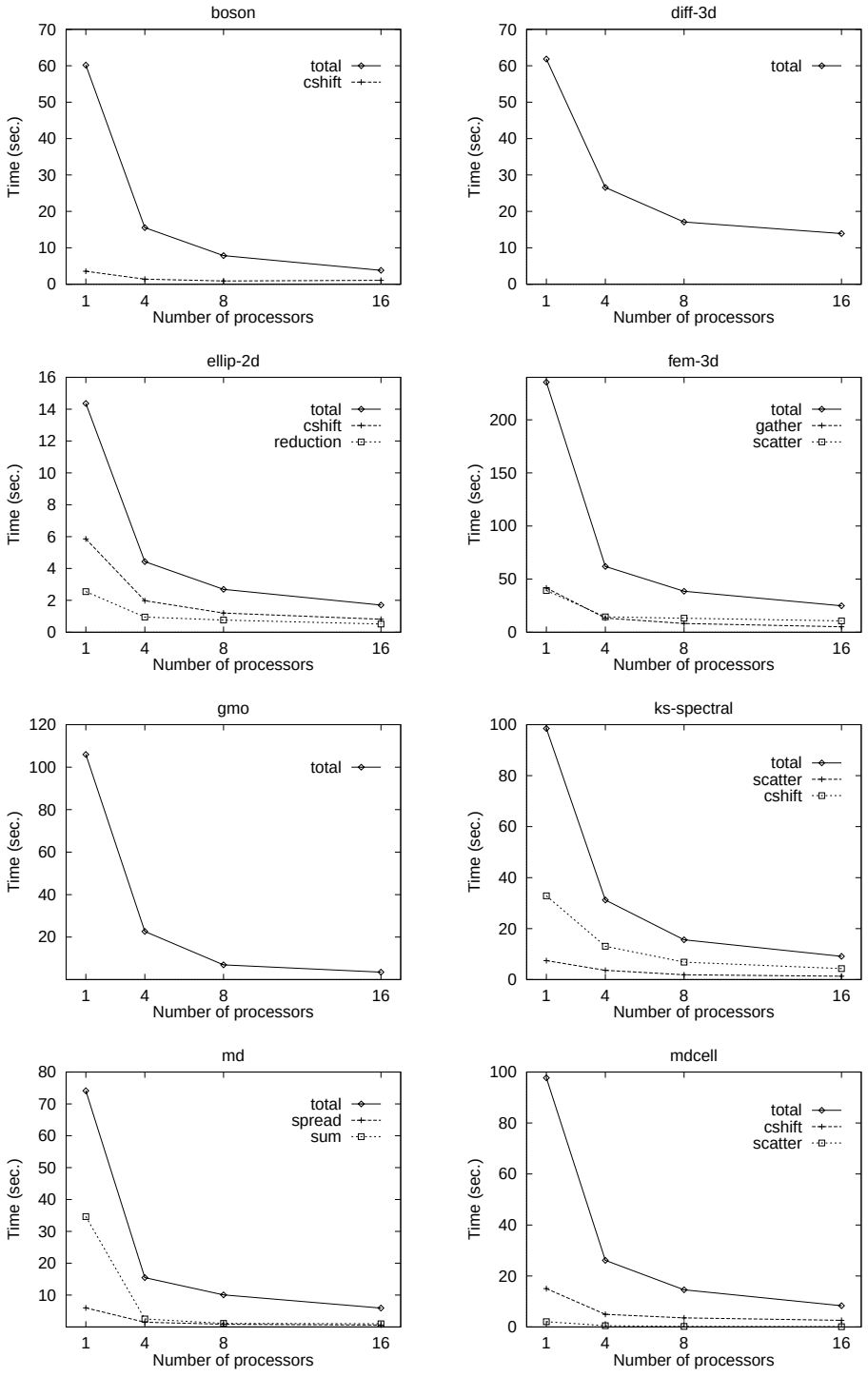


Fig. 4. Total running time and the communication breakdowns for application kernels `boson`, `diff-3d`, `ellip-2d`, `fem-3d`, `gmo`, `ks-spectral`, `md`, and `mdcell`.

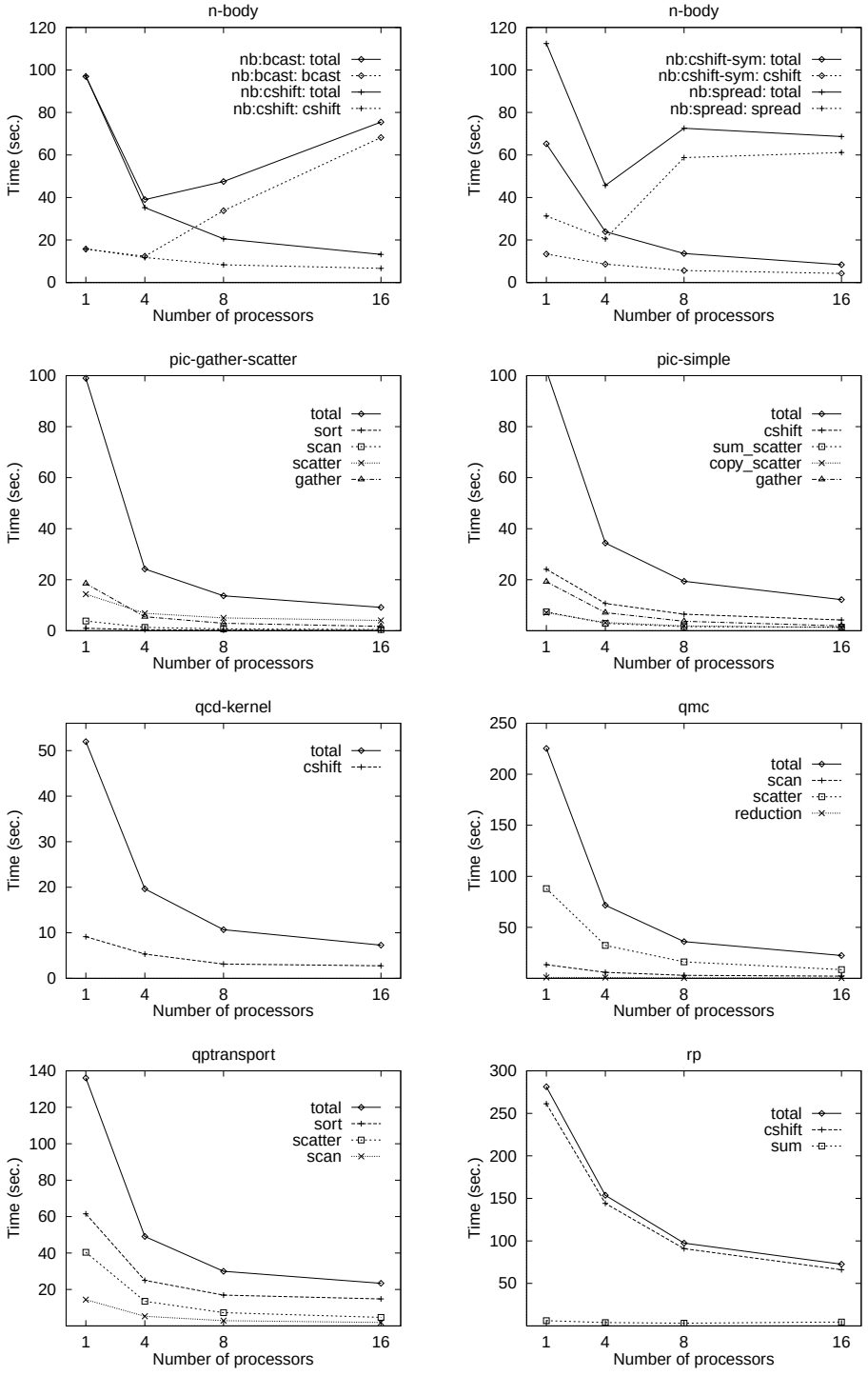


Fig. 5. Total running time and the communication breakdowns for application kernels `n-body`, `pic-gather-scatter`, `pic-simple`, `qcd-kernel`, `qmc`, `qptransport`, and `rp`.

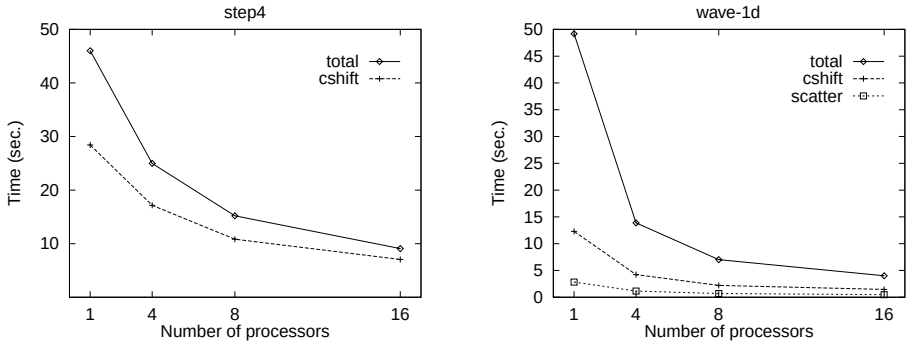


Fig. 6. Total running time and the communication breakdowns for application kernels `step4` and `wave-1d`.

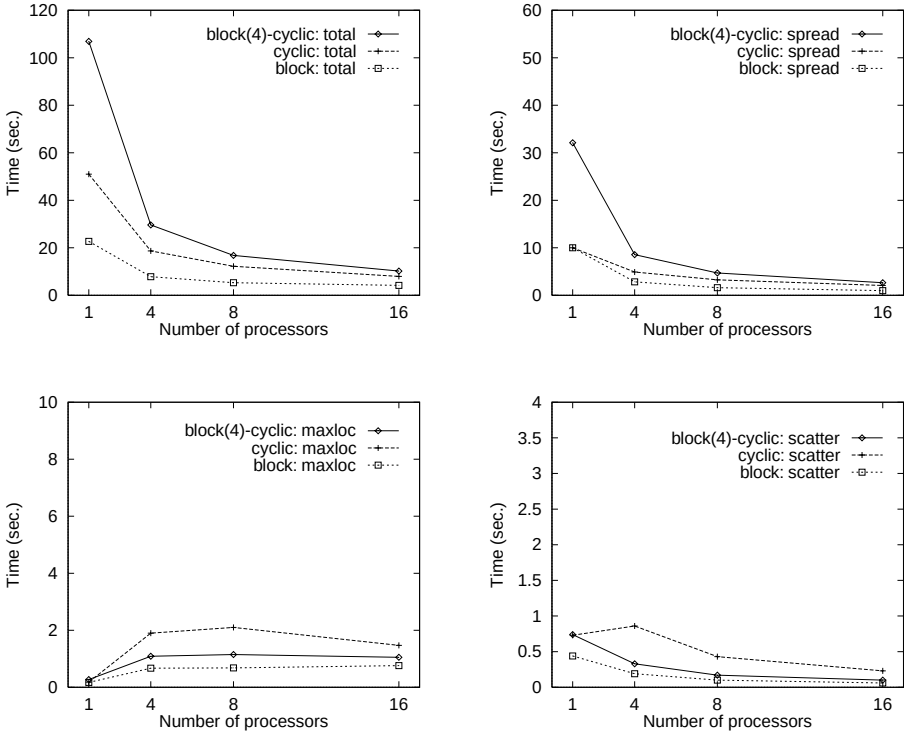


Fig. 7. Total running time and communication time breakdowns of `lu:pivot` under different array layouts. Note the different scale of the y-axis for the breakdowns.

physics, and molecular studies in chemistry or biology. We provide performance evaluation metrics in the form of elapsed times, FLOP rates, and communication time breakdowns, and quantify performance according to the FLOP count, memory usage, communication pattern, local memory access, as well as operation and communication counts per iteration. We also provide a benchmark guide to aid the choice of subsets of the benchmarks for evaluating particular aspects of an HPF compiler. We expect the

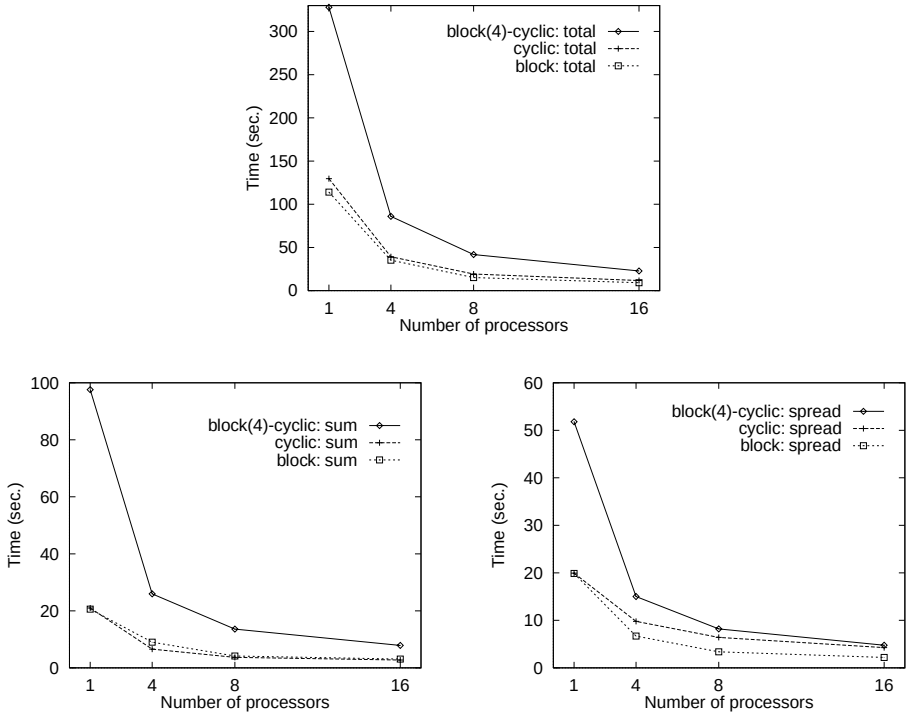


Fig. 8. Total running time and communication time breakdowns of `qr:solve` under different array layouts. Note the different scale of the y-axis for the breakdowns.

HPFBench benchmark suite to be a valuable asset in the development and evaluation of Fortran 90 and HPF compilers.

We have reported a performance evaluation of an industry-leading HPF compiler, *pghpf*, from the Portland Group Inc. using the HPFBench benchmarks on the distributed-memory IBM SP2. While with respect to running the HPF benchmarks on one node, most of the benchmarks achieved good speedups on up to 16 nodes of the SP2, over half of the 25 benchmarks compiled using *pghpf* run over 10% slower than their Fortran 77 counterparts compiled using the native F77 compiler *xlf*. Among these benchmarks, the gap is over a factor of two for five benchmarks. The measurement of communication time breakdowns shows that the high overhead with *pghpf* when running on a single node mainly comes from the poor nodal performance of these communications segments of the codes; though on one node, the data movement is all within the node.

ACKNOWLEDGMENTS

We would like to thank the anonymous referees and the Editor-in-Chief Ronald Boisvert for careful reading of the manuscript and providing us with detailed comments which have greatly improved both the content and the presentation of the article.

REFERENCES

ANDERSON, J., TRAYNOR, C., AND BOGHOSIAN, B. 1991. Quantum chemistry by random walk: Exact treatment of many-electon systems. *J. Chem. Phys. 95*, 10 (Nov.), 7418–7425.

AOKI, S., SHROCK, R., BERG, B., OGILVIE, M., PETCHER, D., BHANOT, G., ROSSI, P., BITAR, K., EDWARDS, R., HELLER, U. M., KENNEDY, A., SANIELEVICI, S., BROWER, R., POTVIN, J., REBBI, C., BROWN, F. R., CHRIST, N., MAWHINNEY, R., DETAR, C., DRAPER, T., LIU, K., GOTTLIEB, S., HAMBER, H., KILCUP, G., SHIGEMITSU, J., KOGUT, J., KRONFELD, A., LEE, I. H., NEGELE, J., OHTA, S., SEXTON, J. C., SHURYAK, E., SINCLAIR, D. K., SONI, A., AND WILCOX, W. 1991. Physics goals of the QCD teraflop project. *Int. J. Mod. Phys. C 2*, 4 (Dec.), 829–947.

BAILEY, D. H. AND BARTON, J. 1985. The NAS kernel benchmark program. Tech. Memo. 86711. RIACS, NASA Ames Research Center, Moffett Field, CA.

BAILEY, D., BARSZCZ, E., BARTON, E., BROWNING, D., CARTER, R., DAGUM, L., FATOOHI, R., FINEBERG, S., FREDERICKSON, P., LASINSKI, T., SCHREIBER, T., SIMON, R., VENKATAKRISHNAN, V., AND WEERATUNGA, S. 1994. The NAS parallel benchmarks. Tech. Rep. RNR-94-007. RIACS, NASA Ames Research Center, Moffett Field, CA.

BAILEY, D., HARRIS, T., SAPHIR, W., WIJNGAARTAND, R., WOO, A., AND YARROW, M. 1995. The NAS parallel benchmarks 2.0. Tech. Rep. NAS-95-020. RIACS, NASA Ames Research Center, Moffett Field, CA.

BATROUNI, G. AND SCALETTAR, R. 1992. World-line quantum Monte Carlo algorithm for a one-dimensional Bose model. *Phys. Rev. B46*, 14 (Oct.), 9051–9062.

BERRY, M., CHEN, D., KOSS, P., KUCK, D., LO, S., PANG, Y., POINTER, L., ROLOFF, R., SAMEH, A., CLEMENTI, E., CHIN, S., SCHNEIDER, D., FOX, G., MESSINA, P., WALKER, D., HSIUNG, C., SCHWARZMEIER, J., LUE, K., ORSZAG, S., SEIDL, F., JOHNSON, O., GOODRUM, R., AND MARTIN, J. 1989. The PERFECT Club benchmarks: Effective performance evaluation of supercomputers. *Int. J. Supercomput. Appl. High Perform. Eng. 3*, 1, 5–30.

BLACKFORD, L. S., CHOI, J., CLEARY, A., D'AZEVEDO, E., DEMMEL, J., DHILLON, I., DONGARRA, J., HAMMARLING, S., HENRY, G., PETITET, A., STANLEY, K., WALKER, D., AND WHALEY, R. C. 1997. Scalapack: A linear algebra library for message-passing computers. In *Proceedings of the SIAM Conference on Parallel Processing* (Mar.), SIAM, Philadelphia, PA.

BRAMLEY, R. AND SAMEH, A. 1992. Row projection methods for large nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput. 13*, 1 (Jan. 1992), 168–193.

BRICKNER, R. G., GEORGE, W., JOHNSSON, S. L., AND RUTTENBERG, A. 1993. A stencil compiler for the Connection Machine models CM-2/200. In *Proceedings of the 4th International Workshop on Compilers for Parallel Computers*, H. Sips, Ed. Delft University of Technology, Delft, The Netherlands, 68–78.

CIMMINO, G. 1939. Calcolo approssimato per le soluzioni dei sistemi di equazioni lineari. *Ric. Sci. Progr. Tecn. Econom. Naz. 9*, 326–333.

COOLEY, J. C. AND TUCKEY, J. 1965. An algorithm for the machine computation of complex Fourier series. *Math. Comput. 19*, 291–301.

CYBENKO, G., KIPP, L., POINTER, L., AND KUCK, D. 1990. Supercomputer performance evaluation and the Perfect Benchmarks. In *Proceedings of the 1990 ACM International Conference on Supercomputing* (ICS '90, Amsterdam, The Netherlands, June 11–15), A. Sameh and H. van der Vorst, Eds. ACM Press, New York, NY, 254–266.

DAHLQUIST, G., BJÖRCK, A., AND ANDERSON, N. 1974. *Numerical Methods*. Prentice-Hall Series in Automatic Computation. Prentice-Hall, Englewood Cliffs, NJ.

DEKKER, T. AND HOFFMAN, W. 1989. Rehabilitation of the Gauss-Jordan algorithm. *Numer. Math. 54*, 4, 591–599.

DONGARRA, J. J. 1989. Performance of various computers using standard linear equations software. Tech. Rep. CS-89-85. Department of Computer Science, University of Tennessee, Knoxville, TN.

DONGARRA, J., MARTIN, J., AND VORLTON, J. 1987. Computer benchmarking: Paths and pitfalls. *IEEE Spectrum 24*, 7 (July 1987), 38–43.

GEORGE, W., BRICKNER, R. G., AND JOHNSSON, S. L. 1994. POLYSHIFT communications software for the connection machine system CM-200. *Sci. Program. 3*, 1 (Spring), 83–99.

GOLUB, G. AND VAN LOAN, C. F. 1989. *Matrix Computations*. 2nd ed. Johns Hopkins University Press, Baltimore, MD.

GREENBERG, A., MESIROV, J., AND SETHIAN, J. 1992. Programming direct N-body solvers on Connection Machines. Tech. Rep. 245. Thinking Machines Corp., Bedford, MA.

HENNESSY, J. L. AND PATTERSON, D. A. 1990. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA.

HIGH PERFORMANCE FORTRAN FORUM. 1993. High Performance Fortran; language specification, version 1.0. *Sci. Program. 2*, 1-2, 1–170.

HIGH PERFORMANCE FORTRAN FORUM. 1997. High Performance Fortran language specification, version 2.0. Rice University, Houston, TX. http://dacnet.rice.edu/Depts/CRPC/HPFF/versions/hpf2/hpf2-v20/index.html.

HIRSCH, J., SUGAR, R., SCALAPINO, D., AND BLANKENBECLER, R. 1982. Monte Carlo simulations of one-dimensional Fermion systems. *Phys. Rev. B26*, 9 (Nov.), 5033–5055.

HO, C.-T. AND JOHNSSON, S. L. 1990. Optimizing tridiagonal solvers for alternating direction methods on Boolean cube multiprocessors. *SIAM J. Sci. Stat. Comput. 11*, 3 (May 1990), 563–592.

HOCKNEY, R. W. 1965. A fast direct solution of Poisson's equation using Fourier analysis. *J. ACM 12*, 1, 95–113.

HOCKNEY, R. AND BERRY, M. 1994. Public international benchmarks for parallel computers: Parkbench committee report-1. Tech. Rep. Oak Ridge National Laboratory, Oak Ridge, TN.

HOCKNEY, R. W. AND EASTWOOD, J. W. 1988. *Computer Simulation Using Particles*. Taylor and Francis, Inc., Bristol, PA.

HOCKNEY, R. W. AND JESSHOPE, C. 1988. *Parallel Computers 2*. Adam Hilger.

HU, Y. AND JOHNSSON, S. L. 1996. Implementing $O(N)$ N-body algorithms efficiently in data-parallel languages. *Sci. Program. 5*, 4, 337–364.

HU, Y. C. AND JOHNSSON, S. L. 1999. Data parallel performance optimizations using array aliasing. In *Algorithms for Parallel Processing*, M. Heath, A. Ranade, and R. Schreiber, Eds. IMA Volumes in Mathematics and Its Applications, vol. 105. Springer-Verlag, Vienna, Austria, 213–245.

HU, Y. C., JOHNSSON, S. L., AND TENG, S.-H. 1997. High Performance Fortran for highly irregular problems. In *Proceedings of the 6th ACM Symposium on Principles and Practice of Parallel Programming* (SIGPLAN '97, Las Vegas, NV, June 18–21), M. A. Berman, Ed. ACM Press, New York, NY.

IBM. 1996. *IBM Parallel Engineering and Scientific Subroutine Library Release 2, Guide and Reference*. IBM Corp., Riverton, NJ.

JOHNSON, C. 1987. *Numerical Solutions of Partial Differential Equations by Finite Element Method*. Cambridge University Press, New York, NY.

JOHNSON, C. AND SZEPESSY, A. 1987. On the convergence of a finite element method for a nonlinear hyperbolic conservation law. *Math. Comput. 49*, 180, 427–444.

JOHNSSON, S. L. 1985. Solving narrow banded systems on ensemble architectures. *ACM Trans. Math. Softw. 11*, 3 (Sept. 1985), 271–288.

JOHNSSON, S. L. AND HO, C.-T. 1989. Optimum broadcasting and personalized communication in hypercubes. *IEEE Trans. Comput. 38*, 9 (Sept.), 1249–1268.

JOHNSSON, S. L., HARRIS, T., AND MATHUR, K. K. 1989. Matrix multiplication on the connection machine. In *Proceedings of the 1989 Conference on Supercomputing* (Reno, NV, Nov. 13–17, 1989), F. R. Bailey, Ed. ACM Press, New York, NY, 326–332.

JOHNSSON, S. L., JACQUEMIN, M., AND KRAWITZ, R. L. 1992. Communication efficient multi-processor FFT. *J. Comput. Phys. 102*, 2 (Oct.), 381–387.

LOMDAHL, P. S., TAMAYO, P., GRØNBECH-JENSEN, N., AND BEAZLEY, D. M. 1993. 50 GFlops molecular dynamics on the Connection Machine 5. In *Proceedings of the Conference on Supercomputing* (Supercomputing '93, Portland, OR, Nov. 15–19), B. Borchers and D. Crawford, Eds. IEEE Computer Society Press, Los Alamitos, CA, 520–527.

LUBECK, O., MOORE, J., AND MENDEZ, R. 1985. A benchmark comparison of three supercomputers: Fujitsu vp-200, hitachi s810/20, and cray x-mp/2. *IEEE Computer 18*, 12, 10–24.

MCMAHON, F. 1988. The Livermore Fortran kernels: A test of numerical performance range. In *Performance Evaluation of Supercomputers*, J. L. Martin, Ed. Elsevier North-Holland, Inc., New York, NY, 143–186.

OLSSON, P. 1994. The numerical behavior of high-order finite difference methods. *J. Sci. Comput. 9*, 4 (Dec. 1994), 445–466.

OLSSON, P. 1995a. Summation by parts, projections, and stability. I. *Math. Comput. 64*, 211 (July 1995), 1035–1065.

OLSSON, P. 1995b. Summation by parts, projections, and stability. II. *Math. Comput. 64*, 212 (Oct. 1995), 1473–1493.

PRESS, W. H., TEUKOLSKY, S. A., VETTERLING, W. T., AND FLANNERY, B. P. 1992. *Numerical Recipes in C: The Art of Scientific Computing*. 2nd ed. Cambridge University Press, New York, NY.

SCHROFF, G. AND SCHREIBER, R. 1988. On the convergence of the cyclic Jacobi method for parallel block orderings. Tech. Rep. 88-11. Computer Science Deptartment, Rensselaer Polytechnic Institute, Troy, NY.

SINVHAL-SHARMA, P., RAUCHWERGER, L., AND LARSON, J. 1991. Perfect benchmarks: Instrumented version. Tech. Rep. CSRD-TR-1152.

TOBOCHNIK, J., BATROUNI, G. G., AND GOULD, H. 1992. Quantum Monte Carlo on a lattice. *Comput. Phys. 6*, 6 (Nov.-Dec.), 673–680.

TRAYNOR, C., ANDERSON, J., AND BOGHOSIAN, B. 1991. A quantum Monte Carlo calculation of the ground state energy of the hydrogen molecule. *J. Chem. Phys. 94*, 5 (Mar.), 3567–3664.

WILKINSON, J. 1961. Error analysis of direct methods of matrix inversion. *J. ACM 8*, 3 (July), 281–330.

WUELLER-WICHARDS, D. AND GENTZSCH, W. 1982. Performance comparisons among several parallel and vector computers on a set of fluid flow problems. Tech. Rep. IB 262-82 R01.

Received: November 1996; revised: July 1998, October 1999, and November 1999; accepted: November 1999