An Evaluation of High Performance Fortran Compilers Using the HPFBench Benchmark Suite

Guohua Jin and Y. Charlie Hu

Department of Computer Science
Rice University
6100 Main Street, MS 132
Houston, TX 77005
{jin, ychu}@cs.rice.edu

Abstract. The High Performance Fortran (HPF) benchmark suite HPFBench was designed for evaluating the HPF language and compilers on scalable architectures. The functionality of the benchmarks covers scientific software library functions and application kernels. In this paper, we report on an evaluation of two commercial HPF compilers, namely, *xlhpf* from IBM and *pghpf* from PGI, on an IBM SP2 using the linear algebra subset of the HPFBench benchmarks.
Our evaluation shows that, on a single processor, there is a significant overhead for the codes compiled under the two HPF compilers and their Fortran 90 companions, compared with the sequential versions of the codes compiled using *xlf*. The difference mainly comes from the difference in code segments corresponding to the communications when running in parallel. When running in parallel, codes compiled under *pghpf* achieve from slightly to significantly better speedups than when compiled under *xlhpf*. The difference is mainly from better performance of communications such as *cshift, spread, sum* and *gather/scatter* under *pghpf*.

1 Introduction

High Performance Fortran (HPF) [8] is the first widely supported, efficient, and portable parallel programming language for shared and distributed memory systems. Since the first release of the HPF specification in 1993, 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 feed back their experience often with some particular class of applications on a particular type of architecture (see, for example [12]). In [11], we have developed the first comprehensive suite of HPF benchmark codes for evaluating HPF compilers.

The functionality of the HPFBench benchmarks covers linear algebra library functions and application kernels. The motivation for including linear algebra li-

P. Amestoy et al. (Eds.): Euro-Par'99, LNCS 1685, pp. 248–257, 1999.

brary functions is for measuring the capability of compilers in compiling the frequently time-dominating computations in many science and engineering applications. 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. Detailed descriptions of the HPFBench suite can be found in [11]. The source code and the details required for the use of the HPFBench suite are covered online at `http://www.crpc.rice.edu/HPFF/benchmarks`.

In this paper, we use the linear algebra subset of the HPFBench benchmarks to evaluate two commercial HPF compilers, namely, *xlhpf* from IBM and *pghpf* from PGI, on an IBM SP2. We evaluate the functionality, the single–processor performance, parallel performance, and performance of communications of the codes generated by the two compilers. Our evaluation shows that, when running on a single processor, there is a significant overhead for the codes compiled under the two HPF compilers and their Fortran 90 companions, compared with the sequential versions of the codes compiled under *xlf*. The difference mainly comes from the difference in code segments corresponding to the communications when running in parallel. When running in parallel, codes compiled under *pghpf* scale from slightly to significantly better than when compiled under *xlhpf*. The difference is mainly from better performance of communications such as *cshift, spread, sum* and *gather/scatter* under *pghpf*.

While numerous benchmarking packages [15, 6, 1, 2, 10] have been developed for measuring supercomputer performance, we are aware of only two that evaluate HPF compilers. The NAS parallel benchmarks [1] were first developed as "paper and pencil" benchmarks that specify the task to be performed and allow the implementor to choose algorithms as well as programming model, though an HPF implementation has been developed recently. The NAS parallel benchmarks consist of only a few benchmarks which are mostly representative of fluid dynamics applications.

The PARKBENCH [10] suite intends to target both message passing and HPF programming models, and to collect actual benchmarks from existing benchmarks suites or from users' submissions. It includes 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 HPF kernels comprises several simple, synthetic applications which test implementations of parallel statements, such as FORALL statements, intrinsic functions with different data distributions, and passing distributed arrays between subprograms.

2 The Linear Algebra Subset of the HPFBench

Linear algebra functions frequently appear as the time-dominant computation kernels of large applications, and are often hand-optimized as mathematical library functions by the supercomputer vendors (e.g. ESSL and PESSL [13] from IBM) or by research institutions (e.g. ScaLAPACK [3] from University of Ten-

Communication Pattern	Arrays		
	1–D(BLOCK)	2–D(BLOCK,BLOCK)	3–D(*,BLOCK,BLOCK)
CSHIFT	conj-grad jacobi, FFT-1D	fft-2D, pcr:coef_inst jacobi, pcr:inst_coef	fft-3D
SPREAD		gauss-jordan, jacobi lu:nopivot, lu:pivot qr:factor, qr:solve	matrix-vector
SUM		qr:factor, qr:solve	matrix-vector
MAXLOC		gauss-jordan, lu:pivot	
Scatter		gauss-jordan, fft-2D lu:pivot	fft-3D
Gather		gauss-jordan	

Table 1. Communication pattern of linear algebra kernels (the array dimensions for reduction and broadcast are of source and destination respectively).

nessee, et. al.). 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 subset of the HPFBench benchmark suite is provided to enable testing the performance of compiler generated code against that of any highly optimized library, such as the PESSL and ScaLAPACK, and exploiting potential automatic optimizations in compilers.

The linear algebra library function subset included in the HPFBench suite is comprised of eight routines. Table 1 shows an overview of the data layout for the dominating computations and the communication operations used along with their associated array ranks.

`conj-grad` uses the Conjugate Gradient method for the solution of a single instance of a tridiagonal system. The tridiagonal system is stored in three 1–D arrays. The tridiagonal matrix–vector multiplication in this method corresponds to a three–point stencil in one dimension. The three–point stencil is implemented using *CSHIFT*s.

`fft` computes the complex–to–complex Cooley–Tukey FFT [4]. 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.

`gauss-jordan` computes the inversion of a matrix via the Gauss–Jordan elimination algorithm with partial pivoting [7]. Pivoting is required if the system is not symmetric positive definite. 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 matrices are brought to zero. Rather than replacing the original matrix with the identity matrix, this space is used to accumulate the inverse solution.

`jacobi` uses the Jacobi method [7] to compute the *eigenvalues* of a matrix. *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. As the sweeps continue, the matrix approaches a diagonal matrix, and the diagonal elements approach the eigenvalues.

`lu` solves dense system of equations by means of matrix factorization. It includes Gaussian elimination with and without partial pivoting. Load–balance is a well–known issue for LU factorization, and the desired array layout is cyclic.

`matrix-vector` computes matrix–vector products. Given arrays **x**, **y** and **A** containing *multiple instances* [14] of vectors x and y and matrix A, respectively, it performs the operation $y \leftarrow y + Ax$ for each instance. The version tested in this paper has multiple instances, with the row axis (the axis crossing different rows) of **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 requires communication during the reduction.

`pcr` solves irreducible tridiagonal system of linear equations $AX = B$ using the *parallel cyclic reduction* method [7, 9]. The code handles multiple instances of the system $AX = B$. The three diagonals representing A have the same shape and are 2–D 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 3–D. In this case, its first axis represents the right–hand–sides, and 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.

`qr` solves dense linear systems of equations $AX = B$ using Householder transformations [5, 7], commonly referred to as the *QR routines*. With the QR routine, the matrix A is factored into a trapezoidal matrix Q and a 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 $AX = B$. In our experiments, these two phases are studied separately and only the solver is reported. The HPFBench version of the QR routines only supports single–instance computation and performs the Householder transformations *without* column pivoting.

3 Methodology

This paper evaluates two commercial HPF compilers on an IBM SP2. The evaluation therefore focuses on comparing the functionality of the HPF languages supported by the compilers and the performance of the generated codes.

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 multiply are available from IBM's PESSL library, and in other libraries such as ScaLAPACK. 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 require 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 *xlf*. Such a comparison will expose the overhead of the HPF compiler-generated codes. We then measure the speedups of the HPF codes on parallel processors relative to the sequential code. This will provide a notion on how well the codes generated by different HPF compilers scale with the parallelism available in a scalable architecture.

4 Evaluation Results

The evaluation reported here are of two commercial HPF compilers, *pghpf* from the Portland Groups, Inc., and *xlhpf* from IBM. Version 2.2-2 of *pghpf* with compiler switch `-O3 -W0,-O4` was used and linked with `-Mmpl`. *xlhpf* is invoked with compiler switch `-O4`. To measure the overhead of HPF compilers on a single processor, we also measured code compiled using Fortran 90 compilers, specifically, using *xlf90* `-O4` and *pghpf* `-Mf90 -O3 -W0,-O4`. The later was linked with `-Mrpm1` and will be denoted as *pgf90* in the rest of this section.

Our evaluation was performed on an IBM SP2. Each node has a RS6000 POWER2 Super Chip processor running AIX 4.3 at 120Mhz and has 128 Mbytes 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 150 Mbytes/second. All results were collected under dedicated use of the machines.

Due to the space limitation, for each benchmark with several subprograms, we only report on one subprogram. For example, we only report on `fft-2d`, `lu:pivot`, `pcr:inst_coef` and `qr:solve`. Table 2 gives the problem size and the total MFLOPs performed for the problem sizes. The problem sizes are chosen so that the total memory requirement will be around 50 Mbytes.

4.1 Functionality

In comparing the two compilers, we found that *pghpf* supports the full set of library procedures which include gather/scatter operations used to express the gather/scatter communication in `fft` and `gauss-jordan`. On the other hand, *xlhpf* does not support any of the library procedures, and we had to rewrite the gather/scatter communication as `forall` statements. Furthermore, *xlhpf* restricts distributed arrays in a subprogram to have the same number of distributed axes.

Benchmark	Problem Size	Iteration	Mflop
conj-grad	2^{17}	962	3278.37
fft-2d	$2^9 \times 2^8$	1	11.1
gauss-jordan	$2^9 \times 2^9$	1	268.7
jacobi	$2^8 \times 2^8$	6	614.2
lu: pivot	$2^9 \times 2^9$	1	89.5
matrix_vector	$2^8 \times 2^8 \times 2^4$	10	21.0
pcr: inst_coef	$2^9 \times 2^9$, 8 RHS	1	117.4
qr: solve	$2^9 \times 2^9$, 16 RHS	1	872.4

Table 2. Problem size and FLOP count.

Code	*xlf*		*xlf90*	*xlhpf*	*pgf90*	*pghpf*
	Time (sec.)	FLOPrate (Mflops/s)	Time vs. *xlf*	Time vs. *xlf*	Time vs. *xlf*	Time vs. *xlf*
conj-grad	176.7	18.56	1.07	1.36	1.08	1.17
fft-2D	5.3	2.12	1.44	2.28	1.07	1.08
gauss-jordan	28.3	9.49	1.17	5.48	1.24	1.46
jacobi	54.5	11.27	4.28	6.04	2.20	1.98
lu: pivot	19.8	4.51	1.07	1.08	1.16	1.13
matrix_vector	1.1	18.66	1.00	7.64	2.64	3.09
pcr: inst_coef	25.3	4.64	1.35	3.36	1.22	0.57
qr: solve	75.3	11.58	1.39	2.07	1.53	1.47

Table 3. Single-processor performance of the linear algebra kernels.

4.2 Sequential Performance

Table 3 compares the the performance of the HPF benchmark codes compiled using *xlf90, xlhpf, 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 all four HPF or F90 compilers incur significant overhead to the generated code when running on a single processor. More specifically, codes compiled using *xlf90* are up to 4.28 times slower than the sequential codes compiled using *xlf*. Futhermore, codes compiled using *xlhpf* are up to 7.64 times slower than the sequential codes, showing that *xlhpf* generates code with additional overhead. In contrast, codes compiled using *pgf90* and *pghpf* are only up to 3.09 times slower than the sequential codes for all benchmarks except `pcr:inst_coef`, which is 43% faster than its sequential counterpart. A close look at this benchmark shows that the improvement is due to the default padding (`-Moverlap=size:4`) by *pghpf* along all parallel dimensions of an array which significantly reduces cache conflict misses in `pcr:inst_coef`.

To understand the overhead incurred on the codes generated by the four compilers, we measured the time spent on segments of the code that would cause communication when running on parallel processors, and compared them with those of the sequential codes. Table 4 lists the time breakdown for the five versions of each benchmark.

Code	Breakdown	*xlf*	*xlf90*	*xlhpf*	*pgf90*	*pghpf*
`conj-grad`	total	176.66	189.82	240.44	190.45	207.13
	cshift	44.81	47.37	75.15	61.77	65.57
`fft-2d`	total	5.25	7.54	11.96	5.60	5.67
	scatter	0.10	0.13	2.28	0.93	0.95
	cshift	1.87	3.43	6.08	1.53	1.77
`jacobi`	total	54.5	233.03	328.98	119.9	107.77
	cshift	14.49	180.60	196.94	58.57	50.37
	spread	7.93	12.62	15.63	14.31	15.08
`pcr: inst_coef`	total	25.3	34.21	84.59	30.77	14.50
	cshift	2.44	15.44	71.01	2.63	2.59
`gauss-jordan`	total	28.31	33.18	155.05	35.18	41.36
	spread	4.72	4.80	5.26	4.91	5.06
	maxloc	9.65	14.22	13.58	17.73	18.27
	scatter	0.03	0.03	119.62	0.03	0.09
`matrix_vector`	total	1.12	1.12	8.37	2.93	3.38
	sum	0.20	0.20	7.45	2.42	2.80
	spread	0.39	0.39	0.41	0.34	0.37
`qr: solve`	total	75.31	104.56	155.57	115.5	110.51
	sum	6.91	8.44	9.42	15.92	20.62
	spread	11.71	24.70	86.94	30.55	19.86
`lu: pivot`	total	19.83	21.14	21.34	22.90	22.50
	spread	8.43	9.58	10.04	9.89	9.93
	maxloc	0.03	0.04	0.09	0.07	0.18
	scatter	0.01	0.02	0.15	0.52	0.44

Table 4. Breakdown of single-processor time of the linear algebra kernels.

Table 4 shows that the overhead of the four HPF or F90 compilers occurs in code segments corresponding to *cshift, spread, sum* and *scatter*. First, *cshift* is up to 29 times slower when compiled using *xlhpf* or *xlf90*, and up to 4 times slower when compiled using *pgf90* or *pghpf*, compared with the sequential versions of the codes. This contributed to the longer total time for `fft-2D, jacobi, pcr:inst_coef` and `conj-grad`. Second, *scatter* under *xlhpf* is extremely slow and this contributed to the slowdown of `gauss-jordan` and `fft-2D` using *xlhpf* when compared to other compilers. Third, *sum* is extremely efficient under *xlf90* and makes `matrix-vector` under *xlf90* as fast as the sequential code. Lastly, *spread* costs almost the same in all five versions of three benchmarks (`lu:pivot, matrix_vector, gauss-jordan`), and is up to 7.4 times slower under *xlhpf* and up to 2.6 times slower under *xlf90, pgf90*, or *pghpf*, for `jacobi` and `qr:solve`, compared with the sequential codes.

4.3 Parallel Performance

Figure 1 shows the parallel speedups of the benchmarks compiled using *xlhpf* and *pghpf* on up to 16 processors of the SP2, using the performance of the sequential codes as the base. Overall, *pghpf* compiled codes scale from about the same to

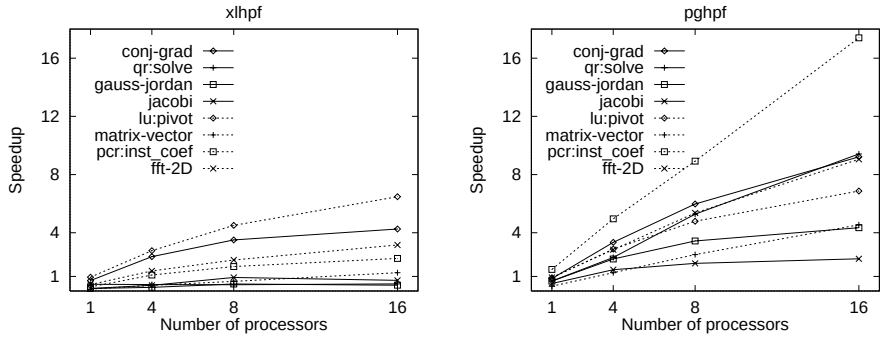

Fig. 1. Speedups of the linear algebra kernels.

significantly better than those compiled with *xlhpf*. When compiled with *xlhpf*, `jacobi`, `gauss-jordan`, `matrix-vector` and `qr:solve` achieve no speedups on 4, 8, and 16 nodes.

To understand the contributing factors to the performance difference between codes compiled using the two compilers, we further measure, for each benchmark, the time spent in communication for runs on 1, 4, 8, and 16 nodes. Figure 2 shows the measured breakdowns of the total time. The performance difference between the two compilers divides the benchmark codes into two groups. In the first group, `lu:pivot` shows very close running time under the two compilers. The rest kernels belong to the second group. These codes when compiled using *pghpf* run from slightly to much faster than when compiled using *xlhpf*. The difference is mainly from the difference in *cshift* under the two compilers, as shown by the kernels `conj-grad, fft-2D`, `jacobi`, and `pcr:inst_coef`. However, for `gauss-jordan`, the *xlhpf* version is much slower than the *pghpf* version because of *gather/scatter*. `matrix-vector` is slower with the *xlhpf* version than the *pghpf* version because of the difference in *sum*. The reason for the poor performance of `qr:solve` with the *xlhpf* is from the slowdown in performing *spread*.

5 Conclusion

We have reported a performance evaluation of two commercial HPF compilers, *pghpf* from Portland Groups Inc. and *xlhpf* from IBM, using a subset of the HPFBench benchmarks on the distributed–memory IBM SP2.

We first compare the single–processor performance of the codes when compiled using the two compilers, their Fortran 90 companions versus that of sequential versions of the benchmarks. Our experiments show that the HPF or F90 incur significant overhead to the generated code. Compared to the sequrntial codes, the codes compiled using *xlhpf* are up to 7.64 times slower, and those compiled using *pghpf* up to 3.09 times slower. We further measure the time breakdowns of selected benchmarks and find that the total running time difference largely comes from difference in code segments which would generate communication when running in parallel. Other than *sum* which is highly efficient under *xlf90*,

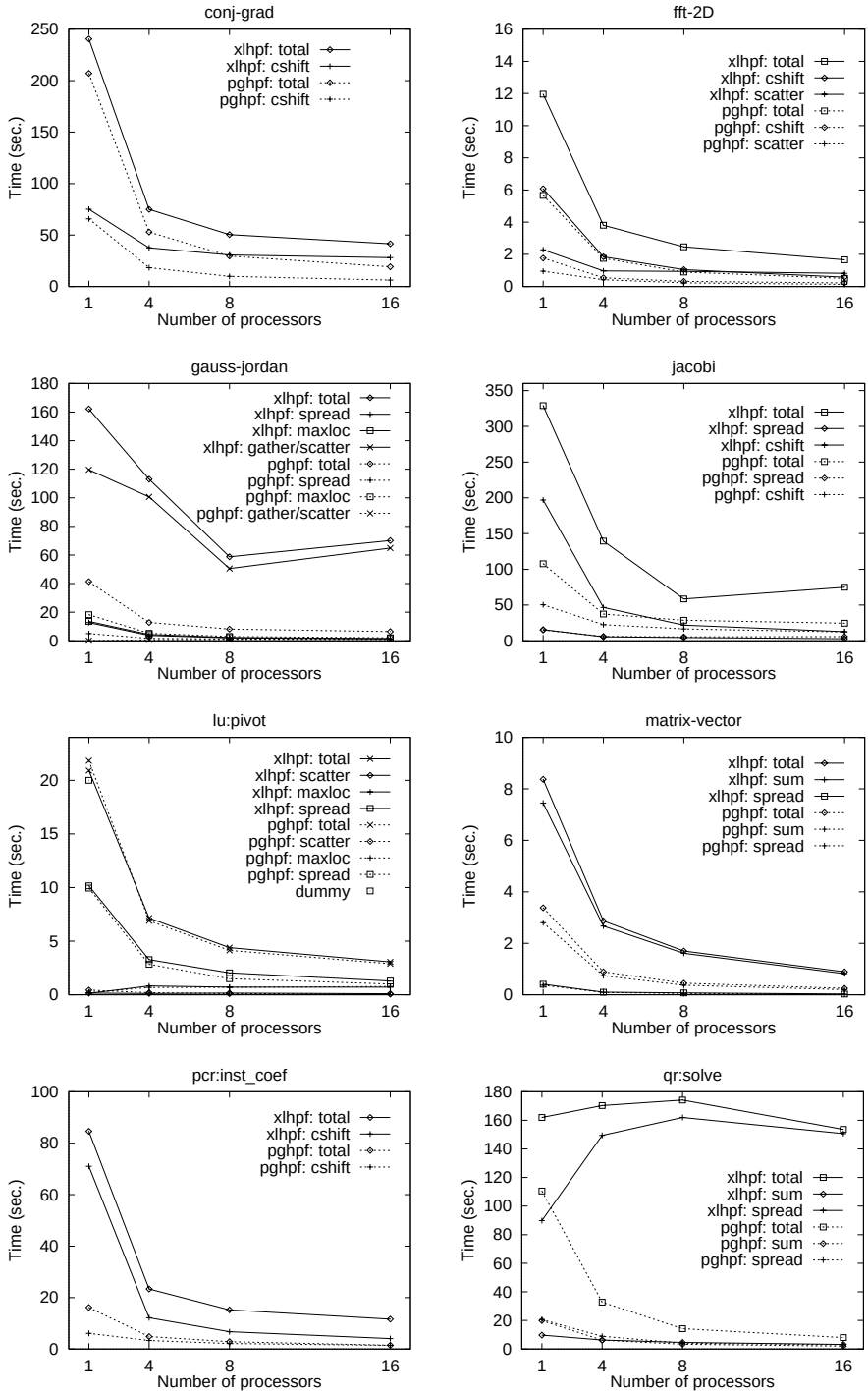

Fig. 2. Total running time and the communication breakdown .

other data movements such as *cshift, spread, gather/scatter* are mostly slower under *xlf90* and *xlhpf* than under *pgf90* and *pghpf*. When running in parallel, codes compiled under *pghpf* show better scalability than when compiled under *xlhpf*. The difference is mainly from better performance of communications such as *cshift, spread, sum* and *gather/scatter* under *pghpf*.

References

[1] D. Bailey, et. al. The NAS parallel benchmarks. Technical Report RNR-94-007, NASA Ames Research Center, Moffett Field, California, March 1994.

[2] M. Berry et. al. The PERFECT Club benchmarks: Effective performance evaluation of supercomputers. *International Journal of Supercomputer Applications*, 3:5 – 40, 1989.

[3] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. ScaLAPACK: A Linear Algebra Library for Message-Passing Computers. In *SIAM Conference on Parallel Processing*, March 1997.

[4] J. C. Cooley and J. Tukey. An algorithm for the machine computation of complex fourier series. *Math. Comp*, 19:291–301, 1965.

[5] G. Dahlquist, A. Björck, and N. Anderson. *Numerical Methods*. Series in Automatic Computation. Prentice Hall, Inc., Englewood Cliffs, NJ, 1974.

[6] J. J. Dongarra. Performance of various computers using standard linear equations software. Technical Report CS-89-85, University of Tennessee, Department of Computer Science, 1989.

[7] G. Golub and C. vanLoan. *Matrix Computations*. The Johns Hopkins University Press, second edition, 1989.

[8] High Performance Fortran Forum. High Performance Fortran; language specification, version 1.0. *Scientific Programming*, 2(1 - 2):1–170, 1993.

[9] R. W. Hockney and C. Jesshope. *Parallel Computers 2*. Adam Hilger, 1988.

[10] R. Hocney and M. Berry. Public international benchmarks for parallel computers: Parkbench committe report-1. Technical report, Netlib, Oak Ridge National Laboratory, February 1994.

[11] Y. C. Hu, G. Jin, S. L. Johnsson, D. Kehagias, and N. Shalaby. HPFBench: A High Performance Fortran benchmark. Tech. Rep. TR98-322, Computer Science Dept., Rice Univ., 1998. URL: `http://www.crpc.rice.edu/HPFF/benchmarks`.

[12] Y. C. Hu, S. L. Johnsson, and S.-H. Teng. High Performance Fortran for highly irregular problems. In *Proc. of the 6th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Las Vegas, Nevada, June 1997.

[13] IBM. *IBM Parallel Engineering and Scientific Subroutine Library Release 2, Guide and Reference*, 1996.

[14] S. L. Johnsson, T. Harris, and K. K. Mathur. Matrix multiplication on the Connection Machine. In *Supercomputing 89*, pages 326–332. ACM, November 1989.

[15] F. McMahon. The Livermore Fortran kernels: A test of numerical performance range. In *Performance Evaluation of Supercomputers*, pages 143 – 186. North Holland, 1988.