Implementation of NAS Parallel Benchmarks in High Performance Fortran

Michael Frumkin, Haoqiang Jin, Jerry Yan[1]

NAS Technical Report NAS-98-009 September 98

{frumkin,hjin,yan}@nas.nasa.gov
NAS Parallel Tools Group
NASA Ames Research Center
Mail Stop T27A-2
Moffett Field, CA 94035-1000

Abstract

We present an HPF implementation of BT, SP, LU, FT, CG and MG of the NPB2.3-serial benchmark set. The implementation is based on HPF performance model of the benchmark specific primitive operations with distributed arrays. We present profiling and performance data on SGI Origin 2000 and compare the results with NPB2.3. We discuss advantages and limitations of HPF and pghpf compiler.

[1] MRJ Technology Solutions, Inc. M/S T27A-2, NASA Ames Research Center, Moffett Field, CA 94035-1000

1. Introduction

The goal of this study is an evaluation of High Performance Fortran (HPF) as a choice for machine independent parallelization of aerophysics applications. These applications can be characterized as numerically intensive computations on a set of three-dimensional (3D) grids with local access patterns to each grid and global synchronization of boundary conditions over the grid set. In this paper we limited our study to six NAS benchmarks: simulated applications BT, SP, LU and kernel benchmarks FT, CG and MG [2].

HPF provides a data parallel model of computations [8]. In this model calculations are performed concurrently with data distributed across processors. Each processor processes the segment of data which it owns (*owner computes rule*). The sections of distributed data can be processed in parallel if there are no dependencies between them. The sections with dependencies may or may not be processed in parallel depending on the HPF compiler's ability to pipeline computations.

The data parallel model of HPF appears to be a good paradigm for aerophysics applications working with data defined on structured 3D grids. A simple decomposition of grids into independent sections of closely located points followed by a distribution of these sections across processors would fit into the HPF model. In order to be processed efficiently these sections should be independent, well balanced multidimensional blocks. In our implementation of the benchmarks we addressed these issues and suggested data distributions satisfying these requirements.

HPF has a limitation in expressing pipelined computations essential for parallel processing of distributed data with dependencies between sections. To make the code immune to the compiler's ability to pipeline the computations it is necessary to redistribute data in directions orthogonal to the dependencies. It also requires scratch arrays to be kept with an alternate distribution (see sections on BT, SP and FT).

A practical evaluation of the HPF versions of benchmarks was done with the Portland Group `pghpf 2.4` compiler [12] on an SGI Origin 2000 (the only HPF compiler available to us at the time of writing). In the course of the implementation we had to address several technical problems: overhead introduced by the compiler, unknown performance of operations with distributed arrays, and additional memory required for storing arrays with an alternative distribution. To address these problems we built an empirical HPF performance model. In this respect our experience confirms two known problems with HPF compilers [11,4]: lack of a theoretical performance model and the difficulty of tracking down the poor performing pieces of the code. A significant advantage of using HPF is that the conversion from F77 to HPF results in a well structured easily maintained portable program. An HPF code can be developed on one machine and run on another (more than 50% of our development was done on a “Pentium cluster”).

In section 2 we consider a spectrum of choices HPF provides for parallelization and build an empirical HPF performance model in section 3. In section 4 we characterize the algorithmic nature of BT, SP, LU, FT, CG and MG benchmarks and describe an HPF implementation for each of them. In section 5 we compare our performance results with the performance of NPB2.3 Class A benchmarks. Related work and conclusions are discussed in sections 6 and 7 respectively.

2. HPF Programming Paradigm

In the data parallel model of HPF, calculations are performed concurrently over data distributed across processors[1]. Each processor processes the segment of data it owns. In many cases HPF compiler can detect concurrent calculations with distributed data. HPF advises a two-level strategy for data distribution. First, arrays should be coaligned with ALIGN directive. Then each group of coaligned arrays should be distributed onto abstract processors with the DISTRIBUTE directive.

There are several ways to express parallelism in HPF: F90 style of array expressions, FORALL and WHERE constructs, the INDEPENDENT directive and HPF library intrinsics [9]. In array expressions, operations are performed concurrently on segments of data owned by a processor. The compiler takes care of communicating data between processors if necessary. The FORALL statement performs computations for all values of the index (indices) of the statement without guaranteeing any particular ordering of the indices. It can be considered as a generalization of F90 array assignment statement.

The INDEPENDENT directive states that there are no dependencies between different iterations of a loop and the iterations can be performed concurrently. In particular it asserts that Bernstein’s conditions are satisfied: sets of read and written memory locations on different loop iterations don’t overlap and no memory location is written twice on different loop iterations [8, p. 193]. All loop variables which do not satisfy the condition should be declared as new and are replicated by the compiler in order for the loop to be executed in parallel.

[1] The expression “data distributed across processors” commonly used in papers on HPF is not very precise since data resides in memory. This expression can be confusing for shared memory machines. The use of this expression assumes that there is a mapping of memory to processors.

Many HPF intrinsic library routines work with arrays and are executed in parallel. For example, the `random_number` subroutine initializes an array of random numbers in parallel with the same result as a sequential subroutine `compute_initial_conditions` of FT. Other examples are intrinsic reduction and prefix functions.

3. Empirical HPF Performance Model

The concurrency provided by HPF does not come for free. The compiler introduces overhead related to processing of distributed arrays. There are several types of the overhead: creating communication calls, implementing independent loops, creating temporaries, and accessing distributed arrays' elements. The communication overhead is associated with requests of elements residing on different processors when they are necessary for evaluation of an expression with distributed arrays or executing an iteration of an independent loop. Some communications can be determined at compile time while others can be determined only at run time causing extra copying and scheduling of communications [12, Section 6]. As an extreme case, the calculations can be scalarized resulting in a significant slowdown.

The implementation of independent loops in `pghpf` extends the "owner computes" rule. It assigns a *home* array to each independent loop and uses the home array for the localization of loop iterations. The compiler selects a home array from array references within the loop or creates a new template for the home array. If there are arrays which are not aligned with the home array they are copied into a temporary array. This involves allocating/deallocating of the temporaries on each execution of the loop. An additional overhead is associated with the transformations on the loop which the compiler has to perform to ensure its correct parallel execution.

Temporaries can be created for passing a distributed array to a subroutine. All temporarily created arrays must be properly distributed to reduce the amount of copying. Inappropriate balance of the computation/copy operations can cause noticeable slowdown of the program.

The immanent reason of the overhead is that HPF hides the internal representation of distributed arrays. It eliminates the programming effort necessary for coordinating processors and keeping distributed data in a coherent state. The cost of this simplification is that the user does not have a consistent performance model of concurrent HPF constructs. The `pghpf` compiler from Portland Group has a number of ways to convey the information about expected and actual performance to the user. It has flags `-Minfo` for the former, `-Mprof` for the later and `-Mkeepftn` for keeping the intermediate FORTRAN code for the user examination. The `pghpf`

USER's guide partially addresses the performance problem by disclosing the implementation of the INDEPENDENT directive and of distributed array operations [12, Section 7].

To compensate for the lack of a theoretical HPF performance model and to quantify compiler overhead we have built an empirical performance model. We have analyzed the NPB, compiled a list of array operations used in the benchmarks and then extracted a set of primitive operations upon which they can be implemented. We measured performance of the primitive operations with distributed arrays and used the results as a guide in HPF implementations of the NPB.

We distinguish 5 types of primitive actions with distributed arrays, as summarized in Table 1:

- loading/storing a distributed array and copying it to another distributed array with the same or a different distribution (includes shift, transpose and redistribution operations);
- filtering a distributed array with a local kernel (the kernel can be a first or second order star-shaped stencil as in BT, SP and LU, or compact 3x3x3 stencil as in the smoothing operator in MG);
- matrix vector multiplication of a set of 5x5 matrices organized as 3D array by a set of five-dimensional vectors organized in the same way (manipulation with 3D arrays of five-dimensional vectors is a common CFD operation);
- passing a distributed array section as an argument to a subroutine;
- performing a reduction sum.

We used 5 operations of the first group including: (1) assignment of values to a non-distributed array, (2) assignment of values to a distributed array, (3) assignment of values to a distributed array within a loop having a non-distributed dimension declared as independent, (4) shift of a distributed array along a distributed dimension and (5) copy of a distributed array to an array distributed in another dimension (redistribution). In the second group we used filtering with the second order (7 point) finite difference stencil and the fourth order (13 point) finite difference stencil. We used both the loop syntax and the array syntax for implementation. In the third group we used 2 variants of matrix vector multiplication: (10) the standard and (11) with the internal loop unrolled. In the fourth group we have passed 2D section of 5D array to a subroutine. (This group appeared to be very slow and we did not include it into the table). The last group includes: (12) reduction sum of a 5D array to a 3D array and (13) reduction sum.

All arrays in our implementations of these primitive operations are 101x101x101 arrays (odd block sizes were chosen to reduce the cache related effects) of scalars, 5x1 vectors and of 5x5 matrices. We used `BLOCK` distribution only (see section 4.1). The profiling results of these operations, compiled with `pghpf` and run on a SGI Origin 2000, are given in Table 1. The execution time of the first operation compiled explicitly for a single processor was chosen as a base time in each group.

Operation Name\nprocs	**Single Proc.**	**1**	**2**	**4**	**8**	**16**	**32**
1. Serial assignment	1.00	1.27	1.37	1.52	1.56	2.99	3.06
2. Distributed assignment	1.23	1.27	0.68	0.37	0.18	0.10	0.04
3. Distributed assignment + INDEPENDENT	0.89	8.82	5.44	2.89	1.24	0.90	0.58
4. Distributed shift	1.34	2.59	1.93	1.10	0.69	0.65	0.62
5. Redistribution	1.34	1.00	1.21	0.93	0.41	0.30	0.24
6. First order stencil sum	1.00	1.55	0.77	0.24	0.10	0.05	0.03
7. First order stencil sum (array syntax)	0.72	1.55	0.80	0.24	0.09	0.05	0.03
8. Second order stencil sum	1.09	1.94	0.97	0.34	0.14	0.07	0.03
9. Second order stencil sum (array syntax)	0.85	2.18	1.05	0.38	0.16	0.07	0.03
10. Matrix vector multiplication	1.00	1.45	0.67	0.43	0.13	0.11	0.05
11. Matrix vector mult. with internal loop unrolled	1.23	1.49	0.69	0.44	0.14	0.11	0.05
12. 5D to 3D reduction sum	1.00	1.44	0.77	0.42	0.22	0.15	0.06
13. Reduction sum	9.83	2.19	1.16	0.60	0.39	0.19	0.10

TABLE 1. Relative time of basic operations on SGI Origin 2000. The column labeled as “Single Proc.” lists the results of the program compiled with a flag -Mf90 and run on a single processor. This removes overhead of handling of distributed arrays. All other columns list results of the program compiled for a symbolic number of processors and run on the specified number of processors.

We can suggest some conclusions from the profiling data.

- Execution of a sequential, non-distributed code slows down as the number of processors grows (line 1).
- Distribution of an array can have a significant performance penalty even when running on single processor (Single Proc. vs. column 1).

- A placement of the independent directive before a loop over a nondistributed dimension confuses the compiler and slows down the program (line 3).
- Efficiency of some parallel operations is close to 1 (lines 6 and 11) while others have efficiency less then 0.5 (lines 4 and 5).
- Replacing loops with array assignment speeds up the sequential program (line 7 and 9 vs. line 6 and 8) but has no effect on parallel performance.
- Loop unrolling does not affect performance (line 11 vs. line 10) as much in `pghpf 2.4`. In `pghpf 2.2` the difference was larger than a factor of 3 for more than 8 processors.
- The smaller the number of dimensions that are reduced, the better the operation scales (line 12 vs. line 13).
- Passing array sections as arguments is an order of magnitude slower then passing the whole array (not included in the table).

We have used the model to choose the particular way to implement operations with distributed arrays. For example, we have used an array syntax instead of loops in the cases where communications were required (such as calculating differences along the distributed direction). Also we have inlined subroutines called inside of loops with sections of distributed arrays as arguments. We have parallelized a loop even if it looked like the loop performs a small amount of computations and should not affect the total computation time (see conclusion 1).

We used `pgprof` and internal NPB timer for profiling the code. The `pgprof` allows to display time spent in subroutines or lines of the code per each processor. To get the profiling data the code should be compiled with `-Mprof=lines` or `-Mprof=func` flag. The profiler also allows to display the number and the total size of messages. The profiling involves a significant overhead and can not be used for profiling of large programs. For profiling of the benchmarks we used an internal timer supplied with NPB. The timer is serial and can be accessed at synchronization points only, which makes it unsuitable for a fine grain profiling such as processor load variation.

4. HPF Implementation of NAS Benchmarks

NAS Parallel Benchmarks consist of eight benchmark problems (five kernels and three simulated CFD applications) derived from important classes of aerophysics applications [2]. The NPB2.3 suite contains MPI implementations of the benchmarks which have good performance on multiple platforms and are considered as a reference implementation. The

NPB2.3-serial suite is intended to be starting points for the development of both shared memory and distributed memory versions, for testing parallelization tools, and also as single processor benchmarks. We have not included HPF version of EP since we don't expect to get any useful data on HPF performance from EP. We have not included HPF version of C benchmark IS either.

We took NPB2.3-serial as a basis for HPF version. We used our empirical HPF performance model as a guide for achieving performance of HPF code. Also we relied on the compiler generated messages regarding the information on loop parallelization and warnings about expensive communications. We used standard HPF directives (actually a very limited basic subset of the directives) as specified in [7].

We limited ourselves to moderate modifications of the serial versions such as inserting HPF directives, writing interfaces, interchanging loops and depth-1 loop unrolling. The resulting program is F77 code, modernized with F90 syntax and HPF directives rather then pure HPF program written from scratch. We avoided significant changes such as inlining, removing arrays from common blocks and passing them as subroutine arguments. We avoided usage of optimized low level linear algebra and FFT library subroutines. We used flag `-Mmpi` to `pghpf` compiler to generate a parallel code and `mpirun` to run it.

The source code of NPB can be found in the NAS parallel benchmarks home page[1]. The page also contains links to HPF implementations of NPB by Portland Group and by Advanced Parallel Research. Extensive data on NPB performance can be found in T. Faulkner's home page[2]. A comparison of different approaches to semi-automatic parallelization of NPB is given in [5].

Benchmarks BT, SP and LU solve a 3D discretization of Navier-Stokes equation

$$Ku = r \tag{1}$$

where u and r are 5x1 vectors defined at the points of a 3D rectangular grid and K is a 7 diagonal block matrix of 5x5 blocks. The three benchmarks differ in the factoring of K. The FT performs FFT of a 3D array, CG solves a sparse system of linear equations by the conjugate gradient method, and MG solves a discrete Poisson problem on a 3D grid by the V-cycle multigrid algorithm.

[1] http://science.nas.nasa.gov/Software/NPB

[2] http://science.nas.nasa.gov/~faulkner

4.1 BT Benchmark

BT uses Alternating Direction Implicit (ADI) approximate factorization of the operator of equation (1):

$$K \cong BT_x \cdot BT_y \cdot BT_z$$

where BT_x, BT_y and BT_z are block tridiagonal matrices of 5x5 blocks if grid points are enumerated in an appropriate direction. The resulting system is then solved by solving the block tridiagonal systems in x-, y- and z-directions successively. The main iteration loop of BT starts from the computation of r (`compute_rhs`) followed by successive inversion of BT_x, BT_y and BT_z (`x_solve`, `y_solve` and `z_solve`) and is concluded with updating of the main variable `u` (`add`).

Each subroutine `x_solve, y_solve` and `z_solve` solves a second order recurrence relations in the appropriate directions. These computations can be done concurrently for all grid lines parallel to an appropriate axis while the computation along each line is sequential. A concurrency in `x_solve` and `y_solve` can be achieved by distributing the grid along z-direction. This distribution would formally preclude concurrency in `z_solve` since HPF contains no expression mechanism to organize processors to work in a pipelined mode. In order for `z_solve` to work in parallel the grid has to be redistributed along x- or y-direction or both.

In our HPF implementation of BT the subroutines `compute_rhs, x_solve`, `y_solve` and `add` work with `u`, `rhs` and `lhs` distributed blockwise along z-direction. The subroutine `z_solve` works with `rhsy` and `lhsy` distributed blockwise along y-direction and uses `uy` (`uy`, `rhsy` and `lhsy` are copies of `u`, `rhs` and `lhs` distributed in y-direction). The redistribution of `rhs` to `rhsy` is performed at the entrance to `z_solve` and back redistribution is performed upon exit from `z_solve`. The redistribution `uy=u` is performed just before calculation of `lhsy`.

The main loop in `x_solve` (symmetrically `y_solve` and `z_solve`) for each grid point calls 5x5 matrix multiplication, 5x5 matrix inversion and 5x5 matrix by 5x1 vector multiplication. Using `pgprof` we found that the calls had generated a significant overhead probably related to passing a section of a distributed array to a subroutine. These subroutines were inlined and the external loop was unrolled. This reduced the execution time by a factor of 2.9 on up to 8 processors. For a larger number of nodes scaling limitations come into effect and reduction is less.

The inlining and loop unrolling made the internal loop of x_solve too complicated and the compiler message indicated that it had not been able to parallelize the loop. The INDEPENDENT directive was sufficient for parallelization of the loop. It, however, introduced an overhead which caused the program to run 1.85 times slower on single processor relative to the program compiled with `-Mf90` flag.

Note that use of two-dimensional distributions would not give any reduction in the computation to communication ratio. In fact, it would require the redistribution of data three times per iteration and would result in a slower program.

The profile of main BT subroutines is shown on Figure 1. The subroutines which do not involve redistribution and/or communications scale nicely. The communication during the computations of fluxes and dissipation in the *z*-direction affects scaling of the `rhs`. The redistribution time essentially stays constant with the number of processors and is responsible for the reduction of the efficiency on more than 8 processors.

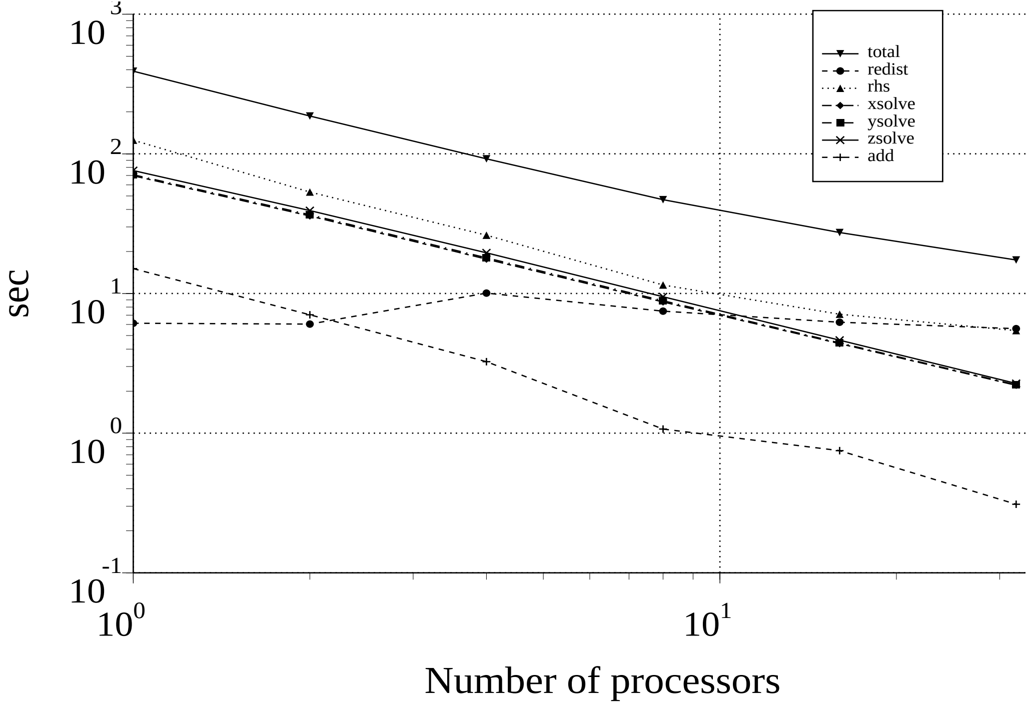

FIGURE 1. BT profile on Origin 2000. Note that `rhs` does not scale well since it involves communications when computing the flux and the dissipation in *z*-direction. The redistribution diminishes the efficiency of the processor utilization as the number of processors grows.

4.2 SP Benchmark

SP uses the Beam-Warming approximate factorization and Pulliam-Chaussee diagonalization of the operator of equation (1) and adds fourth-order artificial dissipation:

$$K \cong T_x \cdot P_x \cdot T_x^{-1} \cdot T_y \cdot P_y \cdot T_y^{-1} \cdot T_z \cdot P_z \cdot T_z^{-1}$$

where T_x, T_y and T_z are block diagonal matrices of 5x5 blocks, P_x, P_y and P_z are scalar pentadiagonal matrices. The resulting system then solved by inverting block diagonal matrices T_x, $T_x^{-1} \cdot T_y$, $T_y^{-1} \cdot T_z$ and T_z^{-1} and solving the scalar pentadiagonal systems.

The main iteration loop of SP is similar to the one in BT. It starts with the computation of `rhs` which is almost identical to `compute_rhs` in BT followed by an interleaved inversion of block diagonal and scalar pentadiagonal matrices and is concluded with updating of the flux `u` (`add`), see Figure 2.

```
do step = 1,niter
   call compute_rhs
   call txinvr
   call x_solve
   call ninvr
   call y_solve
   call pinvr
   call z_solve
   call tzetar
   call add
end do
```

FIGURE 2. The main iteration loop of SP.

Parallelization of SP is similar to the parallelization of BT: all subroutines except `z_solve` operate with data distributed blockwise in the *z*-direction. The subroutine `z_solve` works with data distributed blockwise in *y*-direction. The redistribution of `rhs` and of a few auxiliary arrays is performed at the entrance to `z_solve` and back redistribution of `rhs` is performed on the exit from `z_solve`. As in BT a 2D distribution would require more redistributions and would slow down the benchmark.

Profile of SP (see Figure 3) suggests a few conclusions. The dominant factor of the execution time is the computation of `rhs` and the redistribution. The redistribution time varies slightly with the number of processors and is the major factor affecting scaling of the benchmark. The communications involved in the computing `rhs` in *z*-direction also affect the scaling.

The solver itself takes much less time than these two operations and scales well.

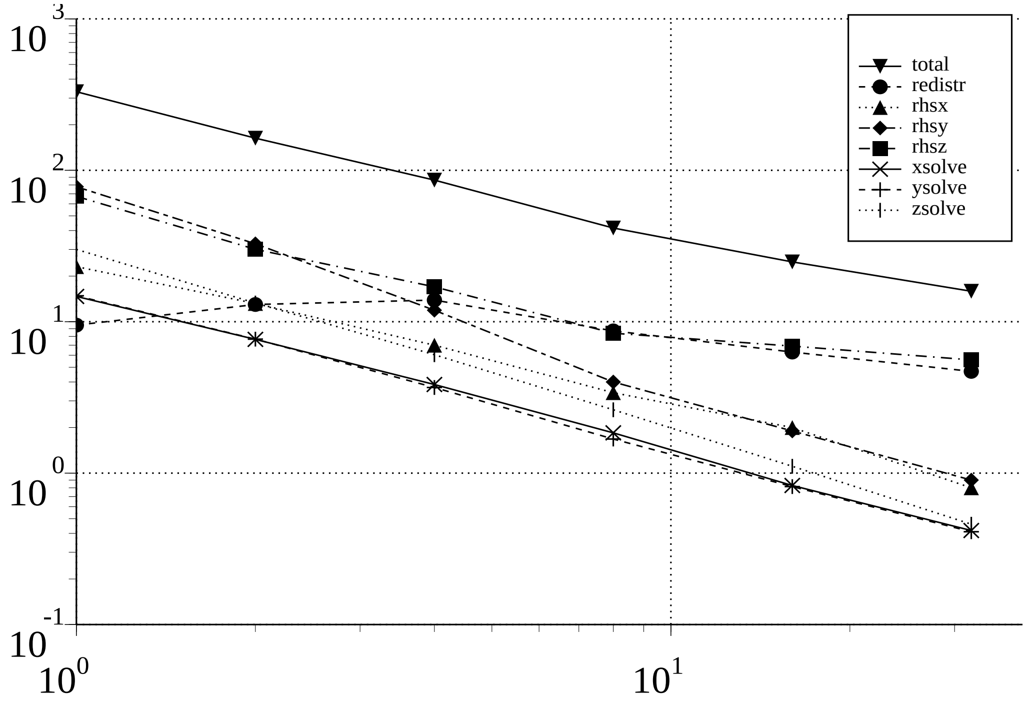

FIGURE 3. SP profile on SGI Origin 2000. The redistribution and communications in `rhsz` effect scaling of SP.

4.3 LU Benchmark

LU implements a version of SSOR algorithm by splitting of the operator of equation (1) into a product of lower triangular matrix and upper triangular matrix:

$$K \cong \omega(2-\omega)(D+\omega Y)(I+\omega D^{-1}Z)$$

where ω is a relaxation parameter, *D* is the main block diagonal of *K*, *Y* consists of three sub block diagonals and *Z* consists of three super block diagonals. The problem is solved by computing elements of the triangular matrices (subroutines `jacld` and `jacu`) and solving the lower and the upper triangular system (subroutines `blts` and `buts`).

The `ssor` is implemented as a sequence of sweeping of the horizontal planes of the grid, see Figure 4.

```
DO k = 2, nz -1
   call jacld(k)
   call blts(k)
END DO
```

```
DO k = nz - 1, 2, -1
   call jacu(k)
   call buts(k)
END DO
call add
call rhs
```

FIGURE 4. LU implementation of `ssor` subroutine.

The subroutines `jacld`, `jacu`, `add` and `rhs` are completely data parallel meaning that operations can be performed concurrently in all grid points. Both `blts` and `buts` have a limited parallelism because processing of an (i,j,k) grid point depends on the values in the points $(i+e,j,k)$, $(i,j+e,k)$ and $(i,j,k+e)$, where $e = -1$ for `blts` and $e = 1$ for `buts`. The small amount of work on each parallel step would cause too many messages to be sent. A method of increasing parallelism and of reduction of the number of messages called Hyperplane Algorithm was proposed by Lamport and used in [3] for the LU implementation and we decided to choose this algorithm for HPF implementation.

In the Hyperplane Algorithm, computations are performed along the planes $i+j+k=m$, where m is a hyperplane number, $m = 6,...,nx+ny+nz-3$. For calculation of the values on each plane, values from the previous plane (lower triangular system) or from the next plane (upper triangular system) are used.

In the Hyperplane Algorithm the external loop on k was replaced by the loop on the plane number m, and j-loop bounds became functions of m and i-loop bounds became functions of m, and j and k is computed as $k = m-i-j$. These loop bounds were taken from precalculated arrays.

Parallelization of LU was done by distribution of arrays blockwise in the j-direction. An advantage of LU relative to BT and SP is that no redistributions are necessary. A disadvantage is uneven distribution of plane grid points causing load imbalance. A 2D distribution could not be handled by the compiler efficiently. (The problem was in assigning of an appropriate home array to a nest of two independent loops with variable loop bounds.)

Profile of LU is shown in Figure 5. Low efficiency of LU resulted from two sources: a large number of relatively small messages (caused by processing of odd shaped arrays) have to be sent after each iteration of *m* loop, and a poor load balancing.

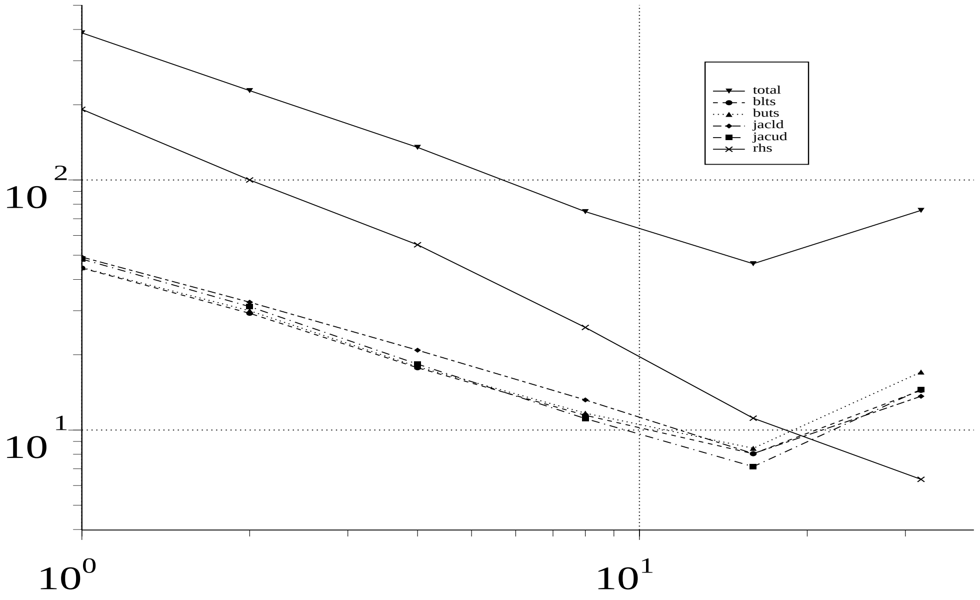

FIGURE 5. LU profile on SGI Origin 2000.

4.4 FT Benchmark

FT implements Fast Fourier Transformation (FFT) of a 3D array. The transformation can be formulated as a matrix vector multiplication:

$$v = (F_m \otimes F_n \otimes F_k)u$$

where *u* and *v* are 3D arrays of dimensions *(m,n,k)* represented as vectors of dimensions *m*x*n*x*k* and F_l, *l=m,n,k* is an FFT matrix of the order l^1. The algorithm is based on factorization of the FFT matrix:where I_l , *l=m,n,k* is

$$F_m \otimes F_n \otimes F_k = (I_m \otimes I_n \otimes F_k)(I_m \otimes F_n \otimes I_k)(F_m \otimes I_n \otimes I_k)$$

the identity matrix of the order *l*. Multiplication of each factor by a vector is equivalent to FFT of the array in one direction, henceforth FT performs FFTs in *x*-, *y*- and *z*- directions successively. The core FFT is implemented as a Swarztrauber's vectorization of Stockham autosorting algorithm performing independent FFTs over sets of vectors. The number of vectors in the sets are chosen to fit the sets into the primary cache.

For the HPF implementation we distributed *u* blockwise in *z*-direction, perform FFTs in *x*-direction, transpose the array, perform FFT in *y*-direction, redistribute the array along *y*-direction and perform FFT in *z*-direc-

[1] Here $A \otimes B$ is a block matrix with blocks $a_{ij}B$ and is called tensor product of *A* and *B*

tion. The loops with FFTs in one direction calling pure `Swarztrauber` subroutine were declared as INDEPENDENT. The transposition and redistribution operations were converted by `pghpf` compiler to FORALL statements automatically given `-Mautopar` flag so that INDEPENDENT directives were unnecessary for these loops.

Note the significant difference between transposition and redistribution. The transposition operation involves reading an array columnwise and writing it rowwise and assumes that these dimensions are not distributed. The transposition does not involve communications. The redistribution copies between two arrays with different distributions and usually requires all-to-all communications. The difference between transposition and redistribution is not as significant on shared memory machines as on distributed memory machines.

Note also that iterations of FT are independent since the result of one iteration is not used for the next one. Neither our HPF version of FT nor NPB2.3 version take advantage of this level of parallelism.

The profile of FT (see Figure 6) shows that the core FFT computations consume about 50% of total time and scale well with the number of processors. The redistribution and transposition don't scale as consistently as the core calculations, reducing the efficiency of the benchmark on a large number of processors

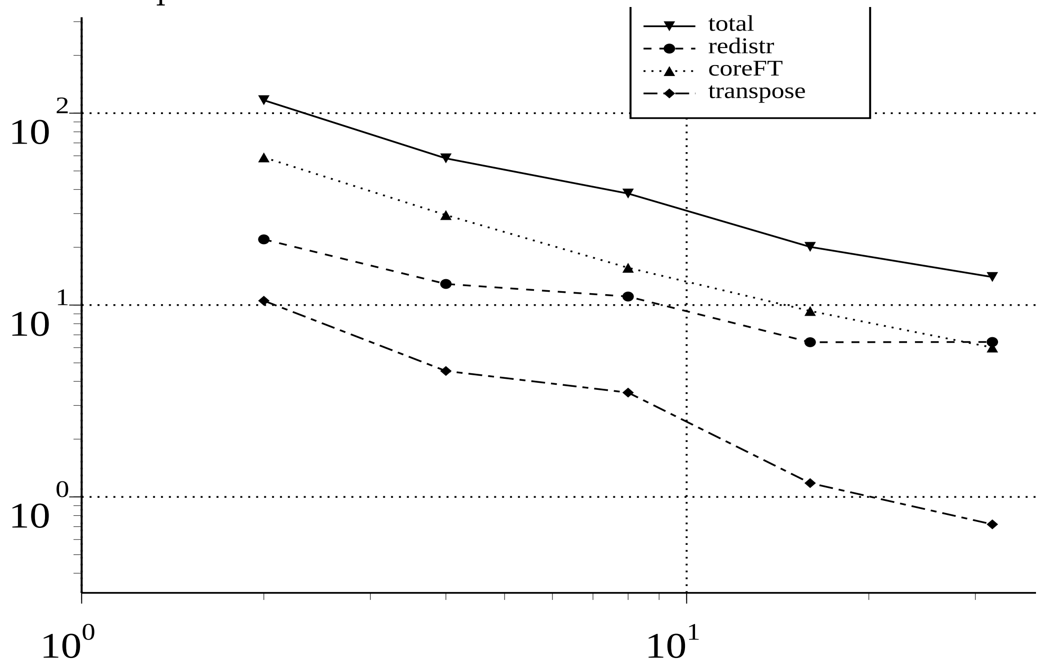

FIGURE 6. FT profile on SGI Origin 2000.

4.5 CG Benchmark

CG is different from the other benchmarks since it works with a large sparse unstructured matrix. CG estimates the largest eigenvalue of a symmetric positive definite sparse matrix by the inverse power method. The core of CG is a solution of a sparse system of linear equations by iterations of the conjugate gradient method. One iteration can be written as follows:

$$q = Ap,\ d = p^T q,$$

$$\alpha = \frac{\rho}{d},\ z = z + \alpha p,\ \ r = r - \alpha q$$

$$\rho_0 = \rho,\ \rho = r^T r,\ \beta = \frac{\rho}{\rho_0},\ p = r + \beta p$$

The main iteration loop contains one sparse matrix vector multiplication, two reduction sums, three `daxpy` operations and a few scalar operations. The most computationally expensive operation is the sparse matrix vector multiplication $q = Ap$. Nonzero elements of *A* are stored by row in a compressed format. The column indices of matrix elements are stored in a separate array `colidx`.

The matrix vector multiplication and `daxpy` operations are parallel, meaning that the computation of each component of the result is independent. In our HPF implementation we distributed *z,q,r* and *x* and replicated *A*, *p* and `colidx`. This allowed the matrix vector operation to be performed in parallel, however `daxpy` operations were performed with vectors having different distributions.

The replication of *A* will cause problems if *A* will not fit into the memory of one processor. On each processor only a small number of rows of *A* are used to calculate the section of *q* distributed onto the processor. The sparsity of *A* makes the sizes of the rows vary and in order to distribute it we created a matrix *B* with number of columns equal to the maximum number of nonzero elements in rows of *A*. We aligned rows of *B* with *q* and copied *A* to *B* row by row. This eliminated replication of *A* but resulted in 20% slower code.

The profile of CG is shown in Figure 7. The matrix vector multiplication scales well. The `daxpy` operations of a replicated *p* with distributed vectors *r* and *z* scale negatively, ruining performance on 32 processors. An explicit replication of *p* slows the program down. An algorithm for matrix vector multiplication which does not require all-to-all communication is given in [10].

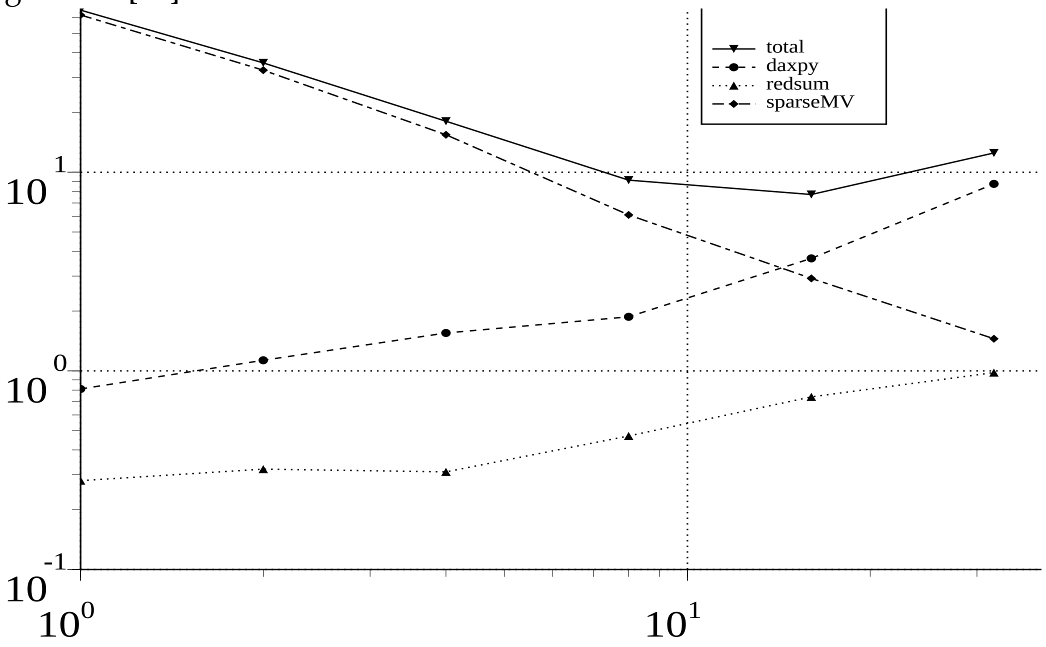

FIGURE 7. CG profile on SGI Origin 2000

4.6 MG benchmark

MG benchmark performs iterations of *V*-cycle multigrid algorithm for solving a discrete Poisson problem $\nabla u = v$ on a 3D grid with periodic boundary conditions [2]. Each iteration consists of evaluation of the residual:

$$r = v - Au$$

and of the application of the correction:

$$u = u + Mr$$

where *M* is the *V*-cycle multigrid operator.

The *V*-cycle starts from an approximate solution on the finest grid, computes the residual and projects it onto progressively coarse grids (down substep). On the coarsest grid it computes an approximate solution by smoothing the residual (`psinv` subroutine), interpolates the solution onto a finer grid, computes the residual and applies the smoothing on the finer grid (up substep). In a few `interp-resid-psinv` substeps the *V*-cycle finishes with an updated solution on the finest grid.

To implement MG in HPF we introduced a 4 dimensional array and mapped grids of different coarseness into 3D sections of this array with a fixed value of the last dimension. We used 1D BLOCK distribution of the array in the *z*-direction. The projection, interpolation, smoothing and computation of the residual are performed at each grid point independently. 2D or 3D partitions would reduce the surface to volume ratio of the array sections and would reduce the number of messages. In practice, however 2D partition resulted in a slightly slower code and 3D partition in a significantly slower code.

The HPF implementation of MG stretches the limits of `pghpf` in a few respects. First, the number of grids and their sizes vary depending on the benchmark class. In order to be able to implement a loop over the grids we need an array of pointers to arrays. This feature is not implemented in the version of `pghpf` compiler which we used. As a work around we introduced the 4D array and used its last dimension as a grid pointer. The overhead of this is allocation of significantly larger memory than actually is used and large strides in accessing points of coarse grids. (In the original F77 version 3D arrays are packed into a 1D array with and are referred to by the address of the first elements.)

Second, the residual and the smoother work on the same grid performing convolutions with 3x3x3 kernels. This operation requires access to non local sections of data and results in a poor scalability of these two subroutines (see MG profile on Figure 9). An implementation of these convolutions with an array syntax did not speed up the benchmark.

The projection and the interpolation subroutines work with a pair of grids, one of which is a refinement of another. Using the same block distribution for all grids collapses the coarsest grids onto a smaller number of processors. It inhibits access to the appropriate portions of the coarser grid. The projection and the interpolation subroutines involve the shuffling operations with grids:

```
  u(2*i1-1,2*i2-1,2*i3-1) =
    u(2*i1-1,2*i2-1,2*i3-1)+z(i1,i2,i3)
  u(2*i1,2*i2-1,2*i3-1)   =
    u(2*i1,2*i2-1,2*i3-1)+z(i1+1,i2,i3)+z(i1,i2,i3)
```

The compiler was not able to parallelize the loop with the shuffling operation in the body because of complex index expressions (according to the compiler's message). We have used the array syntax and `ONHOME` clause for parallelization (see Figure 8).

```
 !hpf$ align w011(i1,i2,i3) with
      .      u(2*i1,2*i2-1,2*i3-1)
 !hpf$ align w111(i1,i2,i3) with
      .      u(2*i1-1,2*i2-1,2*i3-1)
       w111(1:m1-1,1:m2-1,1:m3-1) =
                      z(1:m1-1,1:m2-1,1:m3-1)
       w011(1:m1-1,1:m2-1,1:m3-1) =
      .               z(1:m1-1,1:m2-1,1:m3-1) +
     .                z(2:m1,1:m2-1,1:m3-1)
!hpf$ independent, on home(w011(i1,i2,i3))
      do i3=1,m3-1
        do i2=1,m2-1
          do i1=1,m1-1
          u(2*i1,2*i2-1,2*i3-1) =
     .       u(2*i1,2*i2-1,2*i3-1) + w011(i1,i2,i3)
          end do
        end do
      end do
!hpf$ independent, on home(w111(i1,i2,i3))
      do i3=1,m3-1
        do i2=1,m2-1
          do i1=1,m1-1
          u(2*i1-1,2*i2-1,2*i3-1) =
     .     u(2*i1-1,2*i2-1,2*i3-1) + w111(i1,i2,i3)
           end do
         end do
       end do
```

FIGURE 8. Implementation of the shuffling with ONHOME clause.

The profile of MG (see Figure 9) shows that the smoothing and the residual operators do not scale well. These operators are not factored and require communications to access grid points distributed on different processors.

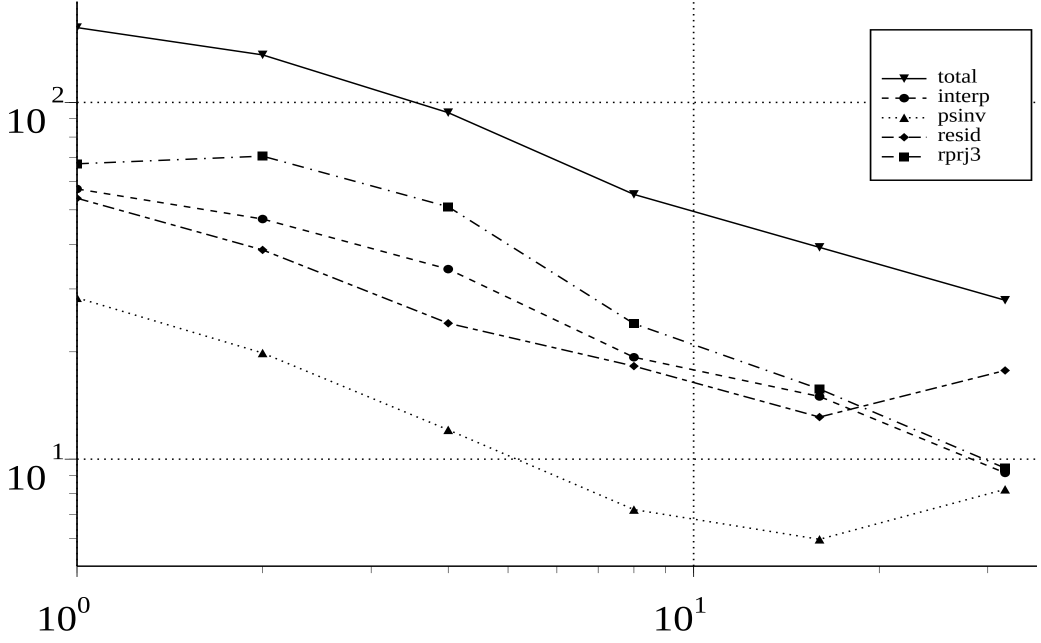

FIGURE 9. MG profile on SGI Origin 2000.

5. Comparison with MPI version of NPB2.3

The timing results of the benchmarks are summarized in Table 2 and the plot is shown in Figure 10. As a reference we use the time on SGI Origin 2000 of the MPI version reported on the NPB home page.

TABLE 2. Benchmarks time on SGI Origin 2000(sec)

Nprocs	**1**	**2**	**4**	**8**	**9**	**16**	**25**	**32**
BT.A pghpf 2.4	3911.3	1865.4	921.1	469.7		273.6		174.0
BT.A NPB2.3	2611.0		731.5		314.0	161.4	91.9	
SP.A pghpf 2.4	3302.9	1629.4	861.2	416.1	371.6	248.4	175.7	158.9
SP.A NPB2.3	1638.4		352.6		142.0	79.1	46.2	
LU.A pghpf 2.4	3285.2	2277.8	1350.4	752.7		462.4		755.6
LU.A NPB2.3	1741.5	795.0	308.2	144.3		67.4		33.8
FT.A pghpf 2.4		116.8	58.1	38.1		20.1		14.0
FT.A NPB2.3	132.8	85.8	44.4	23.1		11.8		6.3
CG.A pghpf 2.4	64.4	34.6	17.32	9.0		7.72		12.5
CG.A NPB2.3	36.4	20.7	9.6	4.4		2.6		1.6
MG.A pghpf 2.4	162.1	136.0	93.65	59.3		39.31		29.5
MG.A NPB2.3	52.7	30.0	15.0	7.6		4.0		2.1

The HPF version are consistently slower than the MPI versions. The lower performance of HPF versions results from two main sources: a single node HPF code runs slower and it does not scale as well as MPI code.

A comparison of single process performance of `pghpf` compiled code versus `f77` code shows that former generates about 2 times slower code than the latter. Since we did not do any code modifications which would change the total operations count or would distort any array layout in the memory (MG is an exception), we would account for this slowdown to the compiler introduced overhead and cost of the redistribution. (The redistribution on a single processor consumes less then 10% of the computational time.)

Processor utilization in HPF code is not as efficient as in the MPI versions (NPB 2.3) for two reasons. HPF versions require an extra redistribution of big arrays and the redistribution does not scale well. In the version of the compiler which we used, the REDISTRIBUTE statement had not been implemented. Implementation of this directive would allow one to organize computations in BT and SP in the following sequence

x_solve, y_solve, z -> y redistribution, z_solve,

x_solve, y -> x redistribution, y_solve, z_solve, x -> z redistribution, ...

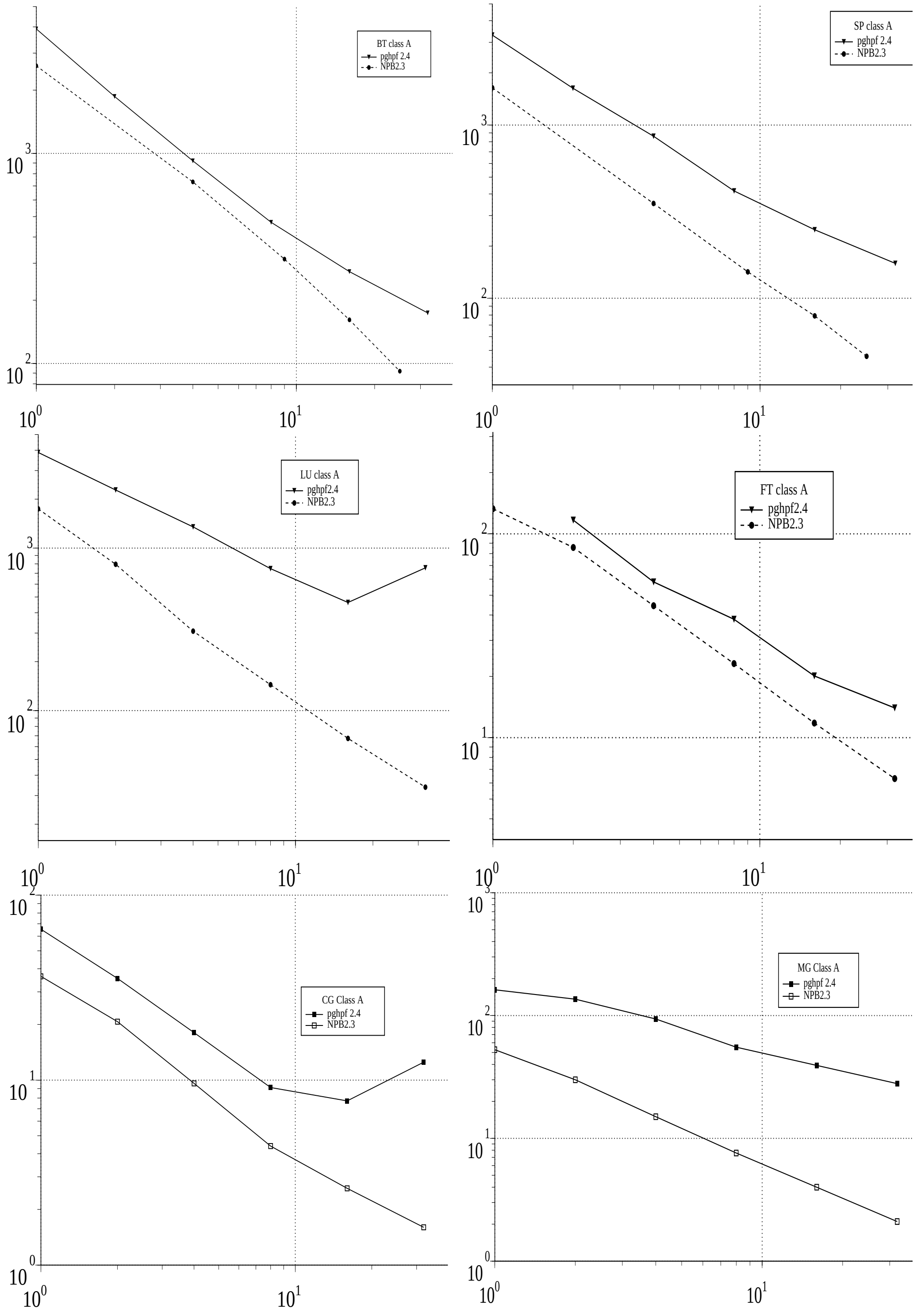

FIGURE 10. HPF versus MPI time for class A on SGI Origin 2000. The horizontal axis is number of processors and the vertical axis is execution time in seconds

This would require 3 redistributions per 2 iterations instead of the current 4 and would reduce the redistribution overhead by a factor of 3/4. Doing this by hand would involve unrolling the main loop in the solver and would require significant code rewriting. The redistribution was the main reason of flattening performance between 16 and 32 processors in BT, SP and FT. An efficient implementation of redistribution would improve scalability of these benchmarks. In the HPF 2.0 language specification, however, the status of REDISTRIBUTE was changed from a language statement to an approved extension (see 7) probably because of difficulties with the implementation. Also HPF does not accommodate advanced domain decompositions like multi-partitioning. However, multi-partitioning itself does require some communication during the matrix inversion.

5. Related Work

The NPB are well recognized benchmarks for testing parallelizing compilers, parallel hardware and parallelization tools [1,11,13]. These benchmarks contain important kernels of aerophysics applications and may be used for early validation of various approaches to the development of high performance CFD codes.

Performance results of HPF implementation of "pencil and paper" NPB specifications submitted by APR[1] and Portland Group[2] are reported in [13]. The compiler vendors know the implementation of operations with distributed arrays and may be implicitly they have an HPF performance model. It allows them to choose the most efficient option for implementation choice. In some cases they use intrinsic customized HPF functions. It allows some `pghpf` compiled benchmarks to outperform handwritten MPI versions of NPB on CRAY T3D and CRAY T3E. Neither implementation has a version of the LU benchmark. APR's implementation of MG uses proprietary HPF directives. The Portland Group FT implementation uses some HPF intrinsic functions customized for the benchmark.

The portability and scalability of HPF programs are studied in [11]. EP, FT and MG are used for comparison of a number of compilers, MPI and ZPL (a data parallel language developed at the University of Washington) implementations. One of the conclusions is that a consistent HPF performance model is important for scalability and portability of HPF programs. The authors of the paper regret: "Unfortunately, a portable HPF version of these (NPB) benchmarks is not available ...". The current report provides a solution to the problem.

[1] http://www.apri.com/apr_nasbench.html

[2] http://www.pgroup.com/npb_results.html

Problems of analysis and code generation for data parallel programs were discussed in [1]. As a solution the authors developed an integer set framework and implemented it in the dHPF environment. The framework was tested and profiled with BT, SP and LU.

A development of a large parallel application in an HPF programming environment called *Fx* is reported in [14]. The authors showed that an air pollution model Airshed fits into the HPF programming paradigm, however it requires a number of redistributions to keep parallelism on the acceptable level. The code demonstrated good performance on up to 64 processors of the Cray T3D and Cray T3E.

An HPF implementation of a reservoir simulation involving a Gaussian elimination algorithm for dense matrices is reported in [6]. Two compilers were compared and good scalability results were achieved on a number of platforms. A comparison of three HPF compilers for IBM SP2 machine is reported in [1].

6. Conclusions

HPF gives the programmer high-level programming language constructs for expressing parallelism existing in a sequential code. It allows the porting of certain classes of sequential codes to a parallel environment with a moderate effort and results in a well structured parallel program. The machine architecture can be accounted for by using an appropriate lower level message passing library as specified by `-Mmpi`, `-Msmp` or `-Mrmp` flags to the `pghpf` compiler and requires a minimal effort from the user.

The hiding of distributed array handling results in uncertainty of the overhead of primitive operations with distributed arrays. Currently there are no HPF language constructs which can convey this overhead to the user. For example, data movement between processors can not be expressed in terms of the HPF language. The problem is addressed in `pghpf` compiler directives `-Minfo` and `-Mkeepftn` as well as in `pgprof` ability to show message size and number. A clear performance model for handling distributed arrays would allow the user to steer the code to a better performance.

The HPF model of parallelism appears to be adequate for expressing the parallelism that existed in BT, SP and FT with one exception. Due to the inability of HPF to express pipelined computations or express multipartitioning, an extra 3D array redistribution was required in each of these benchmarks. The concurrency regions of the LU benchmark are planes

[1] http://www.crpc.rice.edu/NHSFreview/HPF

normal to the grid diagonal and nontrivial code modifications were required to express the parallelism. The efficiency of MG was affected by inability of the compiler to handle arrays of pointers.

At the current level of HPF compiler maturity it generates code which runs about 2 times slower on a single processor than the original serial code. On multiple processors the code speeds up almost linearly until the point (in the 16-32 processor range) where the redistribution creates a significant overhead. We have plans to implement the ARC3D code in HPF and evaluate performance and portability of the benchmarks compiled with other HPF compilers.

Acknowledgments: The authors wish to acknowledge NAS scientists involved in the effort of parallelization of NAS benchmarks: Maurice Yarrow, Rob F. Van der Wijngaart, Michelle Hribar, Abdul Waheed and Cathy Schulbach. Insight to `pghpf` implementation of distributed array operations was provided by Douglas Miles and Mark Young from Portland Group. The work presented in the paper is supported under NASA High Performance Computing and Communication Program.

7. References

[1] V. Adve, J. Mellor-Crummey. *Using Integer Sets for Data-Parallel Program Analysis and Optimization.* To Appear in Proceedings of the SIGPLAN'98 Conference On Programming Language Design and Implementation, June 98.

[2] D. Bailey, T. Harris, W. Sahpir, R. van der Wijngaart, A. Woo, M. Yarrow. *The NAS Parallel Benchmarks 2.0.* Report NAS-95-020, Dec. 1995. http://science.nas.nasa.gov/Software/NPB.

[3] E. Barszcz, R. Fatoohi, V. Venkatakrishnan, S. Weeratunga. *Solution of Regular, Sparse Triangular Linear Systems on Vector an Distributed-Memory Multiprocessors*. NAS report RNR-93-007, April 1993.

[4] J-Y. Berhou, L. Colomert. *Which approach to parallelizing scientific codes - That is a question.* Parallel Computing 23(1997) 165-179.

[5] M. Frumkin, M. Hribar, H. Jin, A. Waheed, J. Yan. *A Comparison of Automatic Parallelization Tools/Compilers on the SGI Origin 2000.*Will be presented as a technical paper at Supercomputing98.

[6] K.G. Li, N. M. Zamel. *An Evaluation of HPF Compilers and the Implementation of a Parallel Linear equation Solver Using HPF and MPI.*Technical paper presented at Supercomputing 97, November 97, San Jose, CA.

[7] *High Performance Fortran Language Specification*. High Performance Fortran Forum, Version 2.0, CRPC-TR92225, January 1997, http://www.crpc.rice.edu/CRPC/softlib/TRs_online.html

[8] C.H. Koelbel, D.B. Loverman, R. Shreiber, GL. Steele Jr., M.E. Zosel. *The High Performance Fortran Handbook*. MIT Press, 1994.

[9] C.H. Koelbel. *An Introduction to HPF 2.0. High Performance Fortran - Practice and Experience*. Tutorial Notes of Supercomputing 97. November 97, San Jose, CA.

[10]J.G. Lewis, R.A. van de Geijn. *Distributed Memory Matrix-Vector Multiplication and Conjugate Gradient Algorithms.* Supercomputing'93, Proc. Portland, OR, Nov. 15-19, 1993, pp. 484-492.

[11]T. Ngo, L. Snyder, B. Chamberlain. *Portable Performance of Data Parallel Languages.* Technical paper presented at Supercomputing 97, November 97, San Jose, CA.

[12]The Portland Group. *pghpf Reference Manual*. February 1997, 142 pp. http://www.pgroup.com/ref_manual/ hpfref.htm.

[13] S. Saini, D. Bailey. *NAS Parallel Benchmark (Version 1.0) Results 11-96.* Report NAS-96-18, November 1996.

[14]J. Subhlok, P. Steenkiste, J. Stichnoth, P. Lieu. *Airshed Pollution Modeling: A Case Study in Application Development in an HPF Environment.* IPPS/SPDP 98. Proceedings. Orlando, March 30- April 3, 1998, pp. 701-710.

NAS	NAS TECHNICAL REPORT
	Title: Implementation of NAS Parallel Benchmarks in High Performance Fortran
	Author(s): Michael Frumkin, Haoqiang Jin, Jerry Yan
	Reviewers: “I have carefully and thoroughly reviewed this technical report. I have worked with the author(s) to ensure clarity of presentation and technical accuracy. I take personal responsibility for the quality of this document.”
Two reviewers must sign.	Signed: ______________________________ Name: Rob F. Van der Wijngaart Signed: ______________________________ Name: Maurice Yarrow___________
After approval, assign NAS Report number.	Branch Chief: Approved: ______________________________
Date:	NAS ReportNumber: NAS-98-009