Implementation and Evaluation of HPF/SX V2

Hitoshi Murai ,† Takuya Araki ,†† Yasuharu Hayashi ,† Kenji Suehiro † and Yoshiki Seo†

We are developing HPF/SX V2, an HPF compiler for vector parallel machines. It provides some unique extensions as well as the features of HPF 2.0 and HPF/JA. This paper describes in particular four of them: 1) the ON directive of HPF 2.0, 2) the REFLECT and LOCAL directives of HPF/JA, 3) vectorization directives, and 4) automatic parallelization. We evaluated these features through some benchmark programs on NEC SX-5. The results showed that each of them achieved a 5-8 times speedup in 8-CPU parallel execution and the four features are useful for vector parallel execution. We also evaluated the overall performance of HPF/SX V2 by using over 30 well-known benchmark programs from the HPFBench, APR Benchmarks, GENESIS Benchmarks, and NAS Parallel Benchmarks. About half of the programs showed good performance, while the other half suggest weakness of the compiler, especially on its runtimes. It is necessary to improve them to put the compiler to practical use.

1. Introduction

Distributed-memory multicomputers are superior to shared-memory multicomputers in cost and scalability, and therefore widely used in the areas of scientific and engineering computing. Message Passing Interface (MPI)[1),2)] is now frequently adopted as the means to program distributed-memory multicomputers. However, it forces users to manage all of the parallelization steps, such as data layout, communication generation and computation partitioning, which impose a heavy burden on users. High Performance Fortran (HPF)[3)] is designed to support efficient and portable parallel programming for scalable systems. Compared to MPI, HPF imposes a lighter burden on users. All that users need to do is to specify the data layout with some simple directives, and the remaining two tasks, communication generation and computation partitioning, are handled automatically by the compiler.

While such ease of writing parallel programs is a strong point of HPF, there is a problem in that the capability of compilers has a much greater impact on the performance of target programs than how one writes programs, because many of parallelization steps are entrusted entirely to the compilers. Although much research is done in an effort to improve the capability of compilers for HPF or HPF-like languages,[4),6)~8),10)] no mature compilers have been released and it can be said that writing HPF programs equal to MPI in the performance is difficult at present. The HPF2.0 approved extensions and the HPF/JA language specification[5)] were proposed to solve the above problem by supplementing the insufficient capability of current HPF compilers with language extensions.

We are developing HPF/SX V2, an HPF compiler for vector parallel machines. It provides some unique extensions as well as the features of HPF 2.0 and HPF/JA, which mainly aim to solve a problem performing stencil computations (also called a regular problem). This paper describes particularly four of them: 1) the ON directive of HPF 2.0, 2) the REFLECT and LOCAL directives of HPF/JA, 3) vectorization directives, and 4) automatic parallelization.

When a compiler cannot determine the optimal computation partitioning or generate the optimal communication, it parallelizes loops in an inefficient manner or generates unnecessary and expensive communications, with a resultant degradation of performance. In such a case, the ON directive of the HPF2.0 approved extensions or the REFLECT and LOCAL directives of the HPF/JA language specification can be used to instruct the compiler to perform efficient computation partitioning or communication generation.

The ON directive or something similar has been implemented on some compilers for use in research or commerce, such as ADAPTOR,[6)] the D System,[7)] VFC,[8)] pghpf,[9)] etc. VPP For-

† 1st Computers Software Division, NEC Solutions
†† HPC Technology Group, NEC Laboratories
☆ This is a preprint of an article published in "Concurrency and Computation – Practice & Experience – Special Issue : High Performance Fortran, Vol.14, No. 8-9, Wiley, July-10 August 2002".

tran,[10] which has a language specification similar to HPF, supports the `OVERLAPFIX` directive corresponding to the `REFLECT` directive. However, they have not been evaluated sufficiently. In this paper we evaluate and verify their effectiveness through some benchmark programs.

It is important to increase the vectorization ratio in order to achieve good performance on vector machines, but HPF has no specification for vectorizing programs. We define some vectorization directives for HPF and implement them into HPF/SX V2.

ADAPTOR provides the `SELECT` directive, which is used for specifying the vectorization of a dimension of an array, but is not usable in the case where the dimension to be vectorized varies from loop to loop. The vectorization directives of HPF/SX V2, in contrast, can be specified for each loop to attain more flexible vectorization.

HPF/SX V2 provides such an *automatic parallelization* feature, as described in Section 3.4. It tests data dependency in a loop and parallelizes it automatically if possible. This means that users are released from the tedious task of inserting `INDEPENDENT` directives as well as `NEW` clauses into their programs.

In addition to describing and evaluating each of the new features, this paper presents the status of the compiler in terms of the performance of its generating codes, evaluating over 30 well-known benchmark programs from HPFBench,[14] APR Benchmarks,[13] GENESIS Benchmarks,[17] and NAS Parallel Benchmarks.[18]

The remainder of this paper is organized as follows. Section 2 gives a brief overview of HPF/SX V2. Section 3 describes the four features of HPF/SX V2. The evaluation results are shown in Section 4. Finally, Section 5 provides the conclusions.

2. Overview of HPF/SX V2

The HPF/SX V2 compilation system consists of four components: a driver, a translator, a Fortran compiler, and runtime routines (**Fig. 1**).

This system works internally as follows: The driver interprets a command line and invokes the translator and the Fortran compiler with appropriate switches; the translator reads an HPF source program and translates it into an SPMD-style intermediate Fortran source program; the Fortran compiler generates an object program for the NEC SX series supercomputer[15] from the intermediate program; the object program is linked with the runtime routines to produce an executable target program.

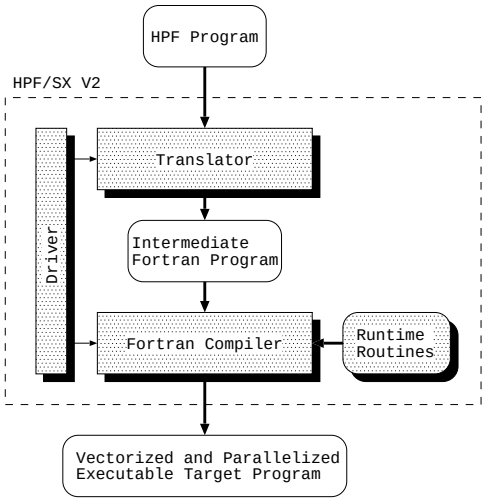

Fig. 1 Components of HPF/SX V2

Each of the four components are presented briefly in the following sections.

2.1 Driver

The driver works as a frontend of HPF/SX V2. First, the driver interprets a command line and translates user-specified compiler options into internal switches for the translator and the Fortran compiler. Second, it invokes in sequence the translator and the Fortran compiler with the translated switches to generate object programs, which are linked with the runtime routines to produce an executable target program.

2.2 Translator

The translator accepts HPF programs and generates SPMD-style intermediate Fortran source programs with calls to the runtime routines inserted. The main tasks of the translator are:

- computation partitioning,
- communication generation and its optimization, and
- inserting codes for parallel execution, such as runtime array descriptor management, temporary array allocation, etc.

It also performs some conventional optimizations such as redundant code elimination, loop-invariant code hoisting, induction variable replacement, etc.[11),12] Other optimizations are

left to the succeeding Fortran compiler.

2.3 Fortran compiler

The intermediate Fortran programs generated by the translator are passed to the backend Fortran compiler. It is an extension of the FORTRAN90/SX compiler developed to enable large-scale scientific computations through Fortran programs. The Fortran compiler vectorizes, parallelizes on SMPs and optimizes the intermediate programs and generates object programs for the NEC SX series supercomputer.

2.4 Runtime routines

The runtime routines are a collection of functions written in C containing MPI library calls. They are called from the intermediate programs to execute such runtime tasks as:

- array descriptor management,
- inter-processor communication, and
- other miscellaneous tasks, such as address conversion, execution control, memory management, I/O, runtime error check, etc.

The array descriptor is a runtime data that holds information about mapping, size, type, etc., of each array. Each of the runtime routines works on the basis of this information.

3. Features of HPF/SX V2

The features of HPF/SX V2 as well as HPF2.0 and HPF/JA are shown in **Fig. 2**.

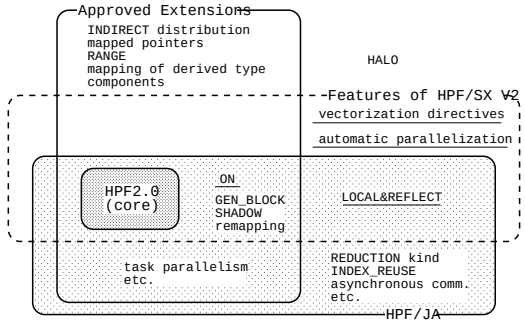

Fig. 2 Features of HPF/SX V2 (Underline means the features are described in this paper)

These features provided by HPF/SX V2 are aimed mainly at writing programs of regular problems effectively.

In parallelizing regular problems, computation partitioning and communication generation are significant for efficient execution. HPF/SX V2 can determine computation partitioning and communication generation not only automatically on the basis of the data layout and data-access pattern but according to the directives specified by users when it cannot determine the optimal by itself.

Considering the characteristics of our target computers, the SX series supercomputers, it is also important to increase the vectorization ratio of programs. HPF/SX V2 provides some unique vectorization directives for users to control its vectorization facility.

HPF/SX V2 has the capability of automatic parallelization, which tests data dependency in a loop and parallelizes it automatically if possible.

In the rest of this section, we describe the features one by one.

3.1 `ON` directive

HPF/SX V2 tries to determine the computation partitioning for loops or statements on the basis of the owner-computes rule. For some loops, however, the one determined by the compiler is not optimal and, as a result, performance is degraded because of loops parallelized inefficiently or expensive communications generated. In such a case, users can instruct the compiler to adopt a better computation partitioning by writing an `ON` directive defined in the HPF 2.0 approved extensions[3)].

Although, according to the HPF 2.0 approved extension, the `ON` directive is for declaring a *task*, which is a code block to be executed concurrently with the other tasks in a task parallel execution, and assigning a set of processors, called an *active processor set*, to the task, HPF/SX V2 now restricts the usage of the directive to specifying computation partitioning for loops or statements.

Consider the example shown in **Fig. 3**: Without the `ON` directive, the compiler would select `a(i)` as the home array for the loop and partition computations so that the *i*'th iteration of the loop is executed on the processor that owns `a(i)`. In this case, three SHIFT-type communications, for the array `b`, `c`, and `d`, respectively, are generated. By following the `ON` directive and selecting `b(i+1)` as the home array, the compiler generates only one SHIFT-type communication for the array `a` to reduce communication overhead by a great extent.

3.2 `REFLECT` and `LOCAL` directives

It is crucial for HPF compilers to detect and execute effectively SHIFT-type communications particularly in regular problems, or nearest-neighborhood codes.

```
!HPF$ DISTRIBUTE (BLOCK) :: a, b, c

!HPF$ INDEPENDENT
      do i=1, 99
!HPF$  ON HOME( b(i+1) )
       a(i) = b(i+1) + c(i+1) + d(i+1)
      end do
```

Fig. 3 Example of the ON directive

A standard technique used frequently for the purpose is to allocate storage on each processor for the local array section so as to include additional space for the elements that have to be moved in from neighboring processors. This additional space is called a SHADOW area. The SHADOW directive in the HPF 2.0 approved extensions is defined to declare a SHADOW area for an array and promote the compiler optimization.

Nevertheless, the specification of the directive does not mention the method of the optimization itself and, therefore, there are some cases occurred where a shadow area is declared but not utilized by compilers.

The REFLECT and LOCAL directives defined in the HPF/JA language specification[5] allow users to directly specify the access of a shadow area and reduce communication overheads. The REFLECT directive is an executable directive, used for updating the shadow area of the arrays each processor owns. The LOCAL directive is the assertion to the compiler that each processor must use the data in its shadow area for a remote data access.

The LOCAL directive can be said to be a stronger assertion than the RESIDENT directive of the HPF 2.0 approved extensions in that there should not be any communications needed for a *local* array, whereas there may be communications needed within the active processor set for a *resident* array.

The REFLECT and LOCAL directives when specified appropriately can improve execution performance for regular problems. Three examples of their usage are shown below.

The first is a case where a compiler cannot detect SHIFT-type communications. HPF/SX V2 analyzes the data layout and access pattern of each array to detect SHIFT-type communications. For the loop in **Fig. 4**, however, the compiler fails to detect a SHIFT-type communication because a variable k, which is not a compile-time constant, appears in the subscript of the array b. In such a case, users can instruct the compiler to generate a SHIFT-type communication with REFLECT and LOCAL, as shown in Fig. 4,[☆] if they know k has a value equal to or less than the shadow width of b.

```
      subtoutine sub(k)
      ...
!HPFJ REFLECT b
!HPF$ INDEPENDENT
      do i=1, 99
!HPF$  ON HOME( a(i) ), LOCAL(b)
       a(i) = a(i) + b(i+k)
      end do
```

Fig. 4 Example of the REFLECT and LOCAL directives (1)

The second is a case where a compiler cannot union two or more communications to eliminate redundant ones for some reason. While HPF/SX V2 has the capability of such an optimization, comparing generated communications with one another and eliminating redundant ones if possible, it sometimes cannot do the optimization sufficiently for the reason that accurate data-flow or control-flow information are not available at compiler-time.

For the program of **Fig. 5**, a naive implementation generates s SHIFT-type communication for the array b before each of the two loops. If it were not for any definitions to the array b along any path from the first loop to the second on the control-flow graph, the communication generated before the second loop was redundant and could be eliminated. Actually, the array b appears as an actual argument of a subroutine call between the two loops and may be updated, and, therefore, the compiler cannot do this optimization unless it performs an inter-procedure analysis. The REFLECT and LOCAL directives are also useful in this case.

The third is a case where a compiler cannot hoist loop-invariant communications out of a loop. A communication generated in a loop degrades execution performance significantly because the communication occurs at each iteration of the loop at runtime. HPF/SX V2 hoists such a communication out of the loop if the communication is *loop-invariant*. Here, a communication is loop-invariant at a loop if the array to be communicated is not modified in the

☆ "!HPFJ" is the prefix of the HPF/JA extensions.

```
!HPFJ REFLECT b
!HPF$ INDEPENDENT
      do i=1, 99
!HPF$  ON HOME( a(i) ), LOCAL(b)
       a(i) = a(i) + b(i+k)
      end do
      ...
      call sub(b)
      ...
!HPF$ INDEPENDENT
      do i=1, 99
!HPF$  ON HOME( a(i) ), LOCAL(b)
       a(i) = a(i) + b(i+k)
      end do
```

Fig. 5 Example of the `REFLECT` and `LOCAL` directives (2)

loop and the communication pattern (i.e., mapping of the array and the width to be shifted for SHIFT-type communications) does not vary during the whole iterations.

In the example of **Fig. 6**, a SHIFT-type communication for an array `b` is needed. If no definition to the array `b` is found in the outer `istep` loop, the best point at which the SHIFT-type communication can be generated is out of the `istep` loop. If your compiler is smart enough, it determines the point. Even if not, you can instruct it with the `REFLECT` and `LOCAL` directives, as shown in Fig. 6.

```
!HPFJ REFLECT b
      do istep=1, N
        ...
!HPF$   INDEPENDENT
        do i=1, 99
!HPF$     ON HOME( a(i) ), LOCAL(b)
          a(i) = a(i) + b(i+1)
        end do
        ...
        call sub(b)
        ...
      end do
```

Fig. 6 Example of the `REFLECT` and `LOCAL` directives (3)

3.3 Vectorization directives

It is important to increase the vectorization ratio in order to achieve good performance on vector machines such as the SX series supercomputer. Even though the backend Fortran compiler of HPF/SX V2 has the capability of advanced automatic vectorization, there are loops that it cannot automatically vectorize because of complicated data dependencies. There is another case where various transformations, such as array subscript conversion, loop restructuring, etc., done for parallelization by the translator may prevent vectorization.

The vectorization facility of HPF/SX V2 can be controlled through our unique vectorization directives. Such directives inserted into source programs are accepted by the compiler and used for generating more efficient vectorized codes.

Currently, HPF/SX V2 accepts a subset of the FORTRAN90/SX vectorization directives, which includes:

- `shortloop`, which asserts that the length of the loop does not exceed the system's vector register length, allowing an special optimization for such a short loop;
- `vector`, which instructs the compiler to vectorize the loop;
- `novector`, which instructs the compiler *not* to vectorize the loop;
- `select (vector|concur)`, which instructs the compiler whether the loop is to be vectorized or parallelized on SMPs; and
- `nodep`, which asserts that the loop does not have any dependency in it that prevents vectorization.

These directives must be preceded by the prefix "`!CDIR`".

For example, to parallelize the outer of such a doubly nested loop, as in **Fig. 7**, and vectorize the inner, users should specify a `NODEP` directive to the inner loop, because a vector subscript `ksamp` is found in the first dimension of the array `trout` which is to be updated in the loop and consequently the compiler cannot determine whether any dependence exists in the loop or not.

```
!HPF$ independent
      do iv=1, nvus
!CDIR  NODEP
       do j=1, lvec
        trout(ksamp(j,iv),iv) = trout(ksamp(j,iv),iv)
     & + wt(j + jbeg,iv)*(trin(isamp(j,iv),iv)
     & + del(j,iv)*(trin(isamp(j,iv) + 1,iv) -
     &              trin(isamp(j,iv),iv)))
       end do
      end do
```

Fig. 7 Example of the vectorization directives (from Benchmark gmo)

3.4 Automatic parallelization

The specification of HPF says that `INDEPENDENT` loops and `FORALL` loops

may be parallelized by an HPF compiler, but the others are not certain. Therefore, to achieve good scalability, users must insert an `INDEPENDENT` directive as well as `NEW` and `REDUCTION` clauses before each of the loops to be parallelized, or translate it into `FORALL` manually. These tasks are very tedious for users to perform.

HPF/SX V2 has the capability of automatic parallelization; it tests data dependency among arrays in each loop, detects `NEW` variables and parallelizes the loop if possible. Unfortunately, the function of detecting reductions automatically is not available now and is one of the targets of future work. The test is performed on the basis of an extended GCD test[12], which would make possible nearly strict analysis for linear subscripts.

This feature releases users from the burden of inserting `INDEPENDENT` and `NEW` into their source programs.

4. Evaluation

We evaluated the features of HPF/SX V2 described in this paper by using some benchmark programs on NEC SX-5. We also evaluated overall performance of HPF/SX V2 using over 30 well-known benchmark programs.

For the evaluation, we used the SX-5 single-node system with 16 processors and a 128-G byte main memory, running under the SUPER-UX (R10.1) operating system. Evaluated compilers were HPF/SX V2 (Rev. 1.1.1 – developing version) and FORTRAN90/SX (Rev. 205). Default values were used for all compiler options (with one exception). The timing routines used were *mpi_wtime* for HPF and *clock* for Fortran. Note that the former measures elapsed time while the latter measures CPU time, so the Fortran-compiled version has a slight advantage over the HPF-compiled version.

4.1 Evaluation of the new features of HPF/SX V2

Evaluation results for the features of HPF/SX V2 are shown one by one in the following sections.

4.1.1 tomcatv

tomcatv is a benchmark program of mesh generation from APR's public domain benchmark suite.[13] The size of each array is extended into 4,097×4,097. **Fig. 8** shows the main loop of this program.

This loop has some candidates for a home array in it. Without any `ON` directive, HPF/SX V2 would determine one of them, `x(:,i+1)`, as a home array, with the result of eight expensive communications. If `x(:,i)` is specified as a home array with an `ON` directive as in the program, only two SHIFT-type communications are generated.

```
!HPF$ independent
      DO 250 i = i1p, i2m
!HPF$ on home( aa(:,i) ) begin
        ...
        DO 310 j = j1p, j2m
          ...
          xx = x(j,i+1) - x(j,i-1)
          ...
          aa(m,i) = -b
          dd(m,i) = b + b + a * rel
          ...
  310   CONTINUE
!HPF$ end on
  250 CONTINUE
```

Fig. 8 Benchmark tomcatv

Table 1 and **Fig. 9** show the evaluation result. The row of "F90" in the table means the results of single-processor execution of the program compiled by FORTRAN90/SX. "original" and "on" represent the result of the HPF-compiled program without and with `ON` specified, respectively. The graph shows speedups relative to single-processor execution of the "original" version. The result shows that about a 1.3 times speedup can be achieved by specifying appropriate computation partitioning with an `ON` directive. The `ON` directive also improves the time of 1-CPU execution to the extent equaling that of Fortran90.

Because, in this program, the `i` loop to be parallelized is also to be vectorized by HPF/SX V2, the vector length, that is the loop length to be executed by a processor, decreases as the number of processors increases. The vector length goes down to 256, which equals the system's vector register length, in 16-CPU parallel execution and, as the result, performance is degraded because of the shorter vector length.

4.1.2 SOR

SOR is a benchmark program of the Red-Black SOR algorithm, which iterates 100 times four "five point stencil" computations, each of which updates (odd,odd), (even,even), (odd,even) and (even,odd) elements of an array, respectively. The size of the array is 4,097×4,097.

The evaluation result is shown in **Table 2**

Table 1 Evaluation result of tomcatv (execution time)

#PE	original (sec)	on (sec)
F90	23.10	–
1	31.99	23.90
2	16.21	11.91
4	8.50	6.29
8	4.65	3.36
16	3.94	3.11

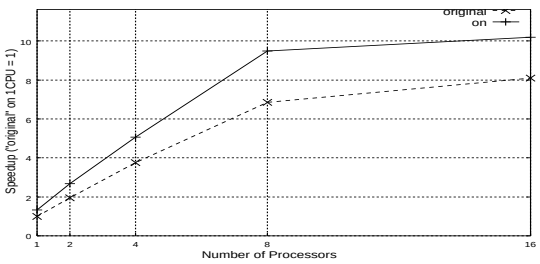

Fig. 9 Evaluation result of tomcatv (speedup)

```
      DO K=1, 100
!HPFJ  reflect a
!HPF$  independent
       DO J=1, (N-1)/2
!HPF$   on home( a(:,2*J+1) ), local
        DO I=1, (N-1)/2
         A(2*I+1, 2*J+1) = (W/4)*(A(2*I,   2*J+1)+
     &                            A(2*I+2, 2*J+1)+
     &                            A(2*I+1, 2*J  )+
     &                            A(2*I+1, 2*J+2))
     &                   + A(2*I+1, 2*J+1)*(1-W)
        END DO
       END DO
         ...
      END DO
```

Fig. 10 Benchmark SOR

and **Fig. 11**. The "original" is for the program with only mapping directives specified and neither `INDEPENDENT` nor `ON`. While the compiler can automatically parallelize all potentially parallel loops and determine the best computation partitioning for them, it cannot emit the optimal communications, with the result of performance degradation. The "reflect+local" represents the program into which `REFLECT` and `LOCAL` directives are inserted to optimize communications. The "reflect+local" is 1.5-2.0 times faster than "original". The time of 1-CPU execution is also improved to equaling that of "F90", as compared with "original".

The speedups of both "original" and "reflect+local" degrade slightly as the number of processors increases, because communication costs tend to dominate overall execution time as the number of processors increases.

4.1.3 gmo

gmo is a benchmark program of a generalized moveout seismic kernel from the HPF-Bench benchmark suite.[14] This program contains such loops as shown in Fig. 7. As described above, the inner loop cannot be vectorized unless `nodep` is inserted, and, consequently, the outer loop is both parallelized and vectorized.

The evaluation result is shown in **Table 3** and **Fig. 12**. Note that a compiler option which prohibit the backend Fortran compiler from performing loop interchange optimization is specified for correctness of the evaluation. The "nodep" representing the program with `nodep` is about 1.4 times faster than the "original" representing the one without `nodep`. The vector operation ratio of the "nodep" is about 93% in 16-CPU parallel execution, whereas that of the "original" is about 78%☆.

Table 2 Evaluation result of SOR (execution time)

#PE	original (sec)	reflect+local (sec)
F90	2.73	–
1	5.15	3.00
2	2.62	1.52
4	1.42	0.83
8	0.86	0.57
16	0.70	0.36

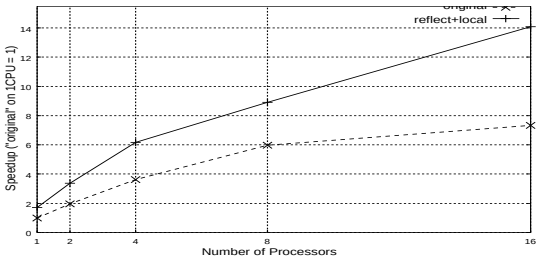

Fig. 11 Evaluation result of SOR (speedup)

Table 3 Evaluation result of gmo (execution time)

#PE	original (sec)	nodep (sec)
F90	33.13	19.89
1	36.19	25.63
2	17.47	12.88
4	8.93	6.43
8	4.47	3.23
16	2.42	1.63

4.1.4 shallow

shallow is a benchmark program of a shallow water model from APR's public domain benchmark suite. The size of arrays are extended into

☆ The values of vector operation ratio are obtained from the hardware monitor of the target machine.

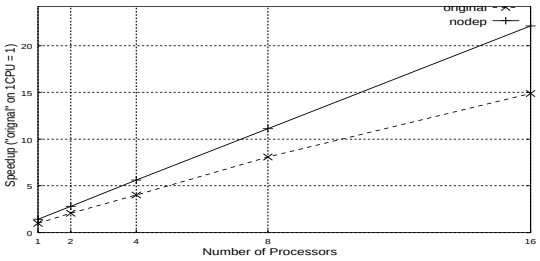

Fig. 12 Evaluation result of gmo (speedup)

4,097×4,097, and only mapping directives and no other directives are specified.

Table 4 and **Fig. 13** show the evaluation result. As against 346 lines of the whole programs consisting of 8 procedures, only 15 lines of directives, which is about 4.3% of the whole, are specified. Although neither `INDEPENDENT`, `ON`, `REFLECT` nor `LOCAL` directives are specified, HPF/SX V2 can automatically parallelize all potentially parallel loops, determine the best computation partitioning and generate optimal communications, with the result of an almost linear speedup of the performance. The result shows that only mapping directives, such as `DISTRIBUTE` , `ALIGN` , etc., are sufficient to achieve good performance of this benchmark program.

Table 4 Evaluation result of shallow (execution time)

#PE	(sec)
F90	10.34
1	14.01
2	7.04
4	3.56
8	1.80
16	0.94

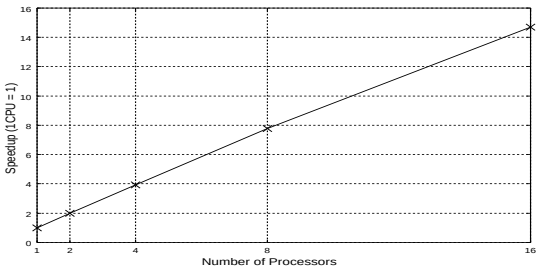

Fig. 13 Evaluation result of shallow (speedup)

4.1.5 Summary

The evaluation results show that the programs modified using the new features of HPF/SX V2 have been improved to about 1.3-2.0 times faster than the original. It can also be seen from the results that the features improve the performance in 1-CPU execution to nearly that of Fortran90. The feature of automatic parallelization can parallelize simple programs, such as **shallow** with neither `INDEPENDENT` or `NEW` inserted. Consequently, it can be said that, by using HPF/SX V2, good performance can be achieved easily with only mapping directives specified for relatively simple programs, and with the other HPF directives and the unique features of HPF/SX V2 for many of the other programs.

4.2 Evaluation on overall performance of HPF/SX V2

This section presents the evaluation results of well-known benchmark programs, from HPF-Bench, APR Benchmarks, GENESIS Benchmarks, and NAS Parallel Benchmarks, to evaluate overall performance of HPF/SX V2.

Each measurement is done in a multi-user environment and the results listed below are the best of (at least) five-time measurements.

Note that in this section we used some benchmarks also used in the evaluations of the preceding sections, and some of these results differ from that of the above sections because of different compiler option or array size.

4.2.1 HPFBench

The HPFBench[14] benchmark suite is a set of High Performance Fortran codes intended for evaluating HPF languages and compilers on scalable parallel architectures. The functionality of HPFBench covers linear algebra library functions and application kernels that reflect the computational structure and communication patterns in typical scientific applications.

Linear algebra library functions are:

(1) triangular solvers – conjugate gradient (**conj-grad**) and 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**)

and application kernels are:

(1) 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 (**mdcell** and **md**)
(8) generic direct N-body solvers with long-range forces (**n-body**)
(9) particle-in-cell in two dimensions (**pic-simple** and **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 nonsymmetrical 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**)

Table 5 and **Figs. 14, 15**, and **16** show the evaluation results for linear algebra library functions. The results were obtained using "as is" codes, i.e., codes without any modification such as "vector tuning". Each row in an entry denotes the execution time in seconds and speedups relative to single-processor execution of the HPF-compiled version. Column F90 presents the results of single-processor execution of programs compiled by the native Fortran 90 compiler (FORTRAN90/SX), and HPF/1, /2, /4, and /8 present the results of 1-, 2-, 4-, and 8-processor executions of programs compiled by HPF/SX V2, respectively.

As shown in the table and graphs, some benchmarks demonstrated poor performance. The major reasons are as follows:

fft: Runtime routine for *copy_scatter* of HPF_LIBRARY dominates the execution time.

gauss-jordan: Runtimes for the *maxloc* and *matmul* intrinsics dominate the execution time.

jacobi: Runtime for the *cshift* intrinsic dominates the execution time and most is consumed in constructing communication schedules. The schedules are constructed every time when *cshift* is called, but they are constant and, therefore, should be reused for the identical *cshift*.

matrix-vector: Runtime for the *sum* intrinsic dominates the execution time. Vector operation ratio was also downgraded from 99% (F90) to 90% (HPF/8).

Table 6 and **Figs. 17, 18**, and **19** show the evaluation results for application kernels. Again, the results were obtained using "as is" codes. Some of the programs had the following problems:

diff-3d: The result of HPF/1 costs over 55 times as much as that of F90. It is mainly due to the cost of constructing a communication schedule carried out on every time-marching iteration. Using a `REFLECT` directive of the HPF/JA extension, the schedule construction can be hoisted out of the loop, then the performance dramatically improves (labeled as "modified" in the table). The modified version is not scalable because of genuine communication costs.

ellip-2d, rp: Runtimes for the *cshift* intrinsic dominate the execution time.

Although we tried to evaluate all of the programs of HPFBench, some were not available for the following reasons:

fft-1d, fft-2d, fft-3d, boson, ellip-2d, ks-spectral, qmc, wave-1d: The F90 results are not available because these programs use `HPF_LIBRARY`[3] routines and FORTRAN90/SX cannot compile them.

lu: The F90 results are not available because of `HPF_LIBRARY` routines and the HPF/8 results are not available because this program terminates abnormally with a wrong result in 8-CPU parallel execution.

fem-3d, pic-simple, qptransport: Again, the F90 results are not available because of `HPF_LIBRARY` routines. The execution times of HPF versions are very long and could not be measured in this evaluation.

mdcell: Both F90 and HPF executions terminate with a user-specified error message. It is assumed that something was wrong with either the benchmark source codes or input data.

pic-gather-scatter: Since a CM Fortran[20] procedure `CMF_FE_ARRAY_TO_CM` is used in this program, FORTRAN90/SX and HPF/SX V2 cannot compile it.

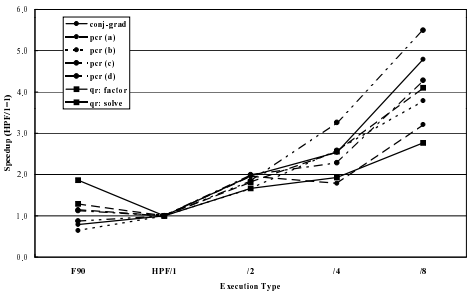

Fig. 14 Results of HPFBench: Linear Algebra Library Functions (1) – HPF's performance is near F90's.

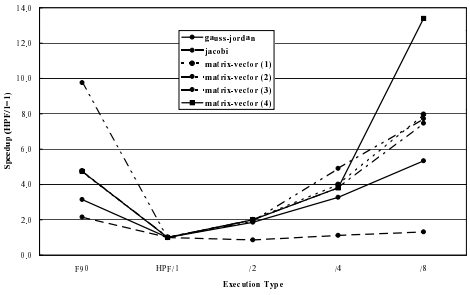

Fig. 15 Results of HPFBench: Linear Algebra Library Functions (2) – HPF's performance is inferior to F90's.

4.2.2 APR Benchmarks

The APR's public domain benchmark suite[13] is the HPF program collection made by Applied Parallel Research for the purpose of evaluating their own HPF compiler. The suite includes many famous benchmarks, such as **shallow**, **tomcatv**, and **grid**. **Table 7** and **Figs. 20** and **21** present the evaluation results for our original tuned versions of the benchmarks. Column Mod/Src displays the ratio of HPF directive lines and original source lines (without comments and blank lines). All benchmarks showed good performance except for rhs_90. In rhs_90,

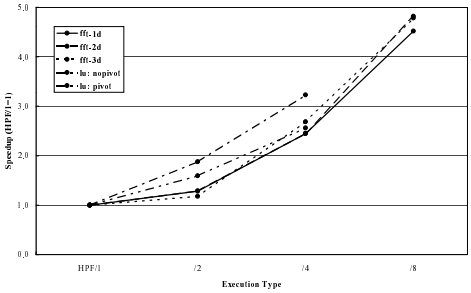

Fig. 16 Results of HPFBench: Linear Algebra Library Functions (3) – F90's performance is not available.

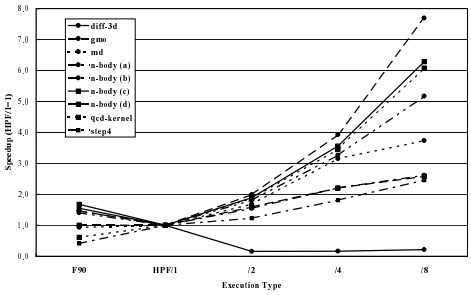

Fig. 17 Results of HPFBench: Application Kernels (1) – HPF's performance is near F90's.

the result of HPF/1 is much worse than that of the F90. Runtime routines for the *eoshift* intrinsic dominate the execution time, and that fact suggests the *eoshift*'s runtimes are inefficient. The performance of **x42** were improved by specifying HPF/JA's `REFLECT` directive on the main loop to control communication schedule construction (labeled as "with reflect" in the Table).

4.2.3 Other benchmarks

The results of some other benchmarks are shown in **Table 8** and **Fig. 22**. The program **trans1** from the GENESIS benchmarks[17] measures the performance of an array transposition operation. The result is bad mainly due to inefficient runtimes. The programs **EP** and **SP** are from the NAS Parallel Benchmarks (NPB-1)[18]. They are moderately scalable. These programs are relatively large and computation costs dominate the overall execution time, even on a vector multi-processor system.

Table 5 Results of HPFBench: Linear Algebra Library Functions

Program	Size/Layout	F90	HPF/1	/2	/4	/8
conj-grad	16777216 (block)	0.8144 (0.789)	0.6423 (1.000)	0.3266 (1.967)	0.2530 (2.539)	0.1341 (4.791)
fft-1d	1048576 (block)	n/a –	3.3857 (1.000)	2.6355 (1.285)	1.3841 (2.446)	0.7496 (4.517)
fft-2d	1024x1024 (b,b)	n/a –	7.4635 (1.000)	5.7974 (1.287)	3.0436 (2.452)	1.5490 (4.818)
fft-3d	128x128x64 (b,b,b)	n/a –	15.0218 (1.000)	12.7420 (1.179)	5.5921 (2.686)	3.1404 (4.783)
gauss-jordan	1024x1024 (b,b)	22.4771 (3.144)	70.6791 (1.000)	38.0925 (1.855)	21.6066 (3.271)	13.2468 (5.336)
jacobi	512x512 (b,b)	32.5512 (2.158)	70.2504 (1.000)	81.8526 (0.858)	62.7482 (1.120)	53.5185 (1.313)
lu: nopivot	1024x1024 (b,b)	n/a –	5.9489 (1.000)	3.1699 (1.877)	1.8431 (3.228)	n/a –
lu: pivot	1024x1024 (b,b)	n/a –	7.7357 (1.000)	4.8514 (1.595)	3.0193 (2.562)	n/a –
matrix-vector (1)	4096x4096 (b,b)	9.8527 (4.734)	46.6450 (1.000)	23.3257 (2.000)	11.6767 (3.995)	5.8529 (7.970)
matrix-vector (2)	4096x4096 (b,b)	9.7536 (4.783)	46.6465 (1.000)	23.3283 (2.000)	12.2170 (3.818)	6.2455 (7.469)
matrix-vector (3)	4096x4096 (b,b)	2.9940 (9.763)	29.2297 (1.000)	15.1836 (1.925)	5.9546 (4.909)	3.7840 (7.725)
matrix-vector (4)	4096x4096 (b,b)	9.8514 (4.739)	46.6856 (1.000)	23.3497 (1.999)	12.2623 (3.807)	3.4836 (13.401)
pcr (a) (Real, coefs, instances)	1024x1024 (b,b)	1.328 (1.148)	1.525 (1.000)	0.778 (1.960)	0.854 (1.786)	0.475 (3.211)
pcr (b) (Real, instances, coefs)	1024x1024 (b,b)	1.328 (0.642)	0.852 (1.000)	0.513 (1.661)	0.329 (2.590)	0.225 (3.787)
pcr (c) (Complex, coefs, instances)	1024x1024 (b,b)	2.718 (1.126)	3.061 (1.000)	1.535 (1.994)	1.339 (2.286)	0.715 (4.281)
pcr (d) (Complex, instances, coefs)	1024x1024 (b,b)	2.718 (0.873)	2.374 (1.000)	1.284 (1.849)	0.728 (3.261)	0.432 (5.495)
qr: factor	512x512 (b,b)	1.7940 (1.862)	3.3400 (1.000)	2.0070 (1.664)	1.7310 (1.930)	1.2070 (2.767)
qr: solve	512x512 (b,b)	4.9650 (1.288)	6.3970 (1.000)	3.5160 (1.819)	2.5020 (2.557)	1.5600 (4.101)

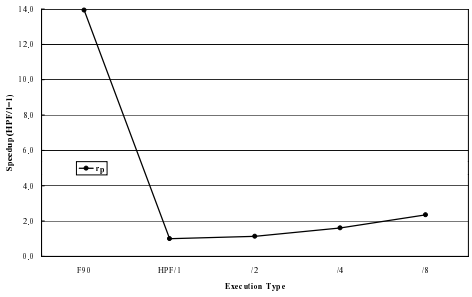

Fig. 18 Results of HPFBench: Application Kernels (2) – HPF's performance is inferior to F90's.

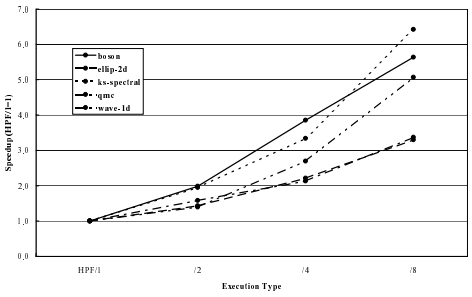

Fig. 19 Results of HPFBench: Application Kernels (3) – F90's performance is not available.

4.2.4 Summary

The results suggest several aspects of HPF/SX V2.

- There are few cases that the poorness of the compiler's generating codes degrades the performance. We meet some cases where useless reconstruction of reusable communication schedule affects the performance. In these cases, however, we can suppress the wasteful reconstruction by using a `REFLECT` directive defined in the HPF/JA extensions.
- Since a vector processor's performance is

Table 6 Results of HPFBench: Application Kernels

Program	Size/Layout	F90	HPF/1	/2	/4	/8
boson	8x128x128 (*,b,b)	n/a	66.1456 (1.000)	33.3061 (1.986)	17.1641 (3.854)	11.7325 (5.638)
diff-3d	128x128x128 (b,b,b)	0.5953 (55.973)	33.3210 (1.000)	31.0541 (1.073)	22.7221 (1.466)	14.9631 (2.227)
diff-3d (modified)	128x128x128 (b,b,b)	0.5952 (1.552)	0.9240 (1.000)	5.8784 (0.157)	5.6708 (0.163)	4.2895 (0.215)
ellip-2d	8192x8192 (b,b)	n/a	5.3082 (1.000)	3.7065 (1.432)	2.3977 (2.214)	1.6081 (3.301)
gmo	12000x2048 (*,b)	5.4244 (1.467)	7.9593 (1.000)	4.0025 (1.989)	2.0251 (3.930)	1.0342 (7.696)
ks-spectral	1024x512 (b,b)	n/a	50.2022 (1.000)	25.7369 (1.951)	15.0321 (3.340)	7.8151 (6.424)
md	4000x4000 (b,b)	3.3026 (0.935)	3.0873 (1.000)	1.8329 (1.684)	0.9796 (3.151)	0.8266 (3.735)
n-body (a) (broadcast)	32768 (block)	4.3457 (1.467)	6.3744 (1.000)	4.1220 (1.546)	2.8855 (2.209)	2.4338 (2.619)
n-body (b) (cshift)	32768 (block)	6.6655 (1.402)	9.3436 (1.000)	5.1636 (1.810)	2.8616 (3.265)	1.8069 (5.171)
n-body (c) (cshift-sym)	32768 (block)	5.0388 (1.676)	8.4466 (1.000)	4.4680 (1.890)	2.3708 (3.563)	1.3442 (6.284)
n-body (d) (spread)	32768 (block)	6.4219 (1.005)	6.4550 (1.000)	4.0500 (1.594)	2.9308 (2.203)	2.5005 (2.582)
qcd-kernel	5x2x8x8x8x8 (*,b,b,b,b,b)	27.0018 (0.612)	16.5372 (1.000)	8.8305 (1.873)	4.7797 (3.460)	2.7196 (6.081)
qmc	8192x128 (b,b)	n/a	27.0205 (1.000)	17.0990 (1.580)	12.6644 (2.134)	8.0175 (3.370)
rp	256x256x256 (b,b,b)	0.7027 (13.940)	9.7960 (1.000)	8.6301 (1.135)	6.0965 (1.607)	4.1586 (2.356)
step4	1024x512 (b,b)	5.252 (0.415)	2.177 (1.000)	1.774 (1.227)	1.201 (1.813)	0.886 (2.457)
wave-1d	16777216 (block)	n/a	70.2298 (1.000)	50.3420 (1.395)	26.0664 (2.694)	13.8512 (5.070)

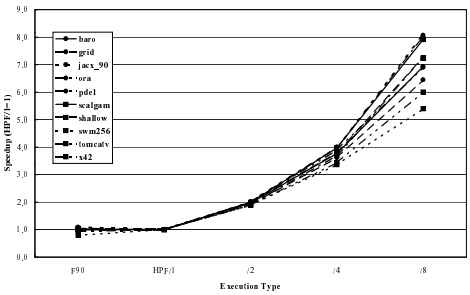

Fig. 20 Results of APR Benchmarks (1) – HPF's performance is near to F90's.

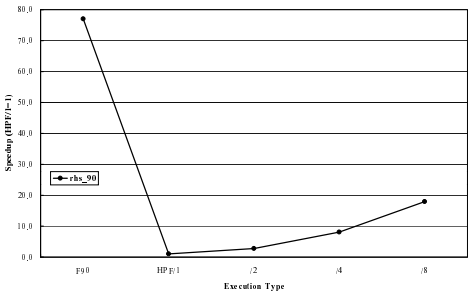

Fig. 21 Results of APR Benchmarks (2) – HPF's performance is inferior to F90's.

relatively high, communication costs tend to dominate overall costs in SX-5. To achieve high performance, problem sizes should generally be as large as possible.

- There are many cases where the poorness of the runtime routines degrades the performance. That is a major problem because users cannot control these runtimes by themselves. Some reasons for the performance degradation and possible improvements are as follows:

  vectorization There are unvectorized loops found in the runtime routines, such as reduction operations, *sum* intrinsic, etc. Some of these loops can be vectorized only by inserting vectorization pragma and the others need to be restructured.

  communication scheduling How to

Table 7 Results of APR Benchmarks

Program	Mod/Src	Size/Layout	F90	HPF/1	/2	/4	/8
baro	16/472 (3.39%)	3002x1502 (*,b)	4.3830 (0.988)	4.3310 (1.000)	2.2059 (1.963)	1.1485 (3.771)	0.6250 (6.929)
grid	14/108 (12.96%)	500x500x2 (*,b,*)	0.4568 (1.037)	0.4738 (1.000)	0.2467 (1.921)	0.1311 (3.614)	0.0735 (6.447)
jacx_90	10/80 (12.50%)	256x256x256 (*,*,b)	2.9711 (1.086)	3.2258 (1.000)	1.6066 (2.008)	0.8076 (3.994)	0.3998 (8.069)
ora	4/338 (1.18%)	456000 (block)	16.3943 (1.000)	16.3931 (1.000)	8.1965 (2.000)	4.0983 (4.000)	2.0492 (8.000)
pde1	17/236 (7.20%)	512x512x512 (*,*,b)	119.325 (0.929)	110.807 (1.000)	55.4094 (2.000)	28.4544 (3.894)	16.0597 (6.900)
rhs_90	5/208 (2.40%)	128x128x128 (*,*,b)	0.3943 (77.018)	30.3681 (1.000)	10.9597 (2.771)	3.7596 (8.077)	1.6943 (17.924)
scalgam	4/559 (0.72%)	256000 (block)	27.200 (1.024)	27.860 (1.000)	13.980 (1.993)	7.020 (3.969)	3.520 (7.915)
shallow	15/346 (4.34%)	2561x2561 (*,b)	5.6274 (1.048)	5.8992 (1.000)	2.9956 (1.969)	1.5487 (3.809)	0.8143 (7.245)
swm256	84/324 (25.93%)	2561x2561 (*,b)	20.8260 (0.791)	16.4650 (1.000)	8.7335 (1.885)	4.8961 (3.363)	3.0516 (5.396)
tomcatv	9/146 (6.16%)	5121x5121 (*,b)	36.5765 (0.966)	35.3407 (1.000)	18.5144 (1.909)	9.5544 (3.699)	4.8843 (7.236)
x42	7/246 (2.85%)	512x512 (*,b)	13.4101 (1.055)	14.1448 (1.000)	8.5145 (1.661)	5.4989 (2.572)	4.1524 (3.406)
x42 (with reflect)	10/246 (4.07%)	512x512 (*,b)	13.4101 (1.005)	13.4791 (1.000)	7.1256 (1.892)	3.9221 (3.437)	2.2434 (6.008)

Table 8 Results of other benchmarks

Program	Mod/Src	Size/Layout	F90	HPF/1	/2	/4	/8
trans1	2/86 (2.33%)	256x256 (*,block)	0.1050 (1.333)	0.1400 (1.000)	0.1780 (0.787)	0.1890 (0.741)	0.2070 (0.676)
trans1	2/86 (2.33%)	2560x2560 (*,block)	10.4060 (0.714)	7.4310 (1.000)	7.3210 (1.015)	5.9040 (1.259)	5.0530 (1.471)
EP	2/138 (1.45%)	Class A (block)	11.1473 (1.008)	11.2330 (1.000)	5.6176 (2.000)	2.8147 (3.991)	1.4173 (7.926)
SP	29/3464 (0.84%)	Class A (*,*,b)	881.10 (0.81)	712.10 (1.00)	349.50 (2.04)	188.10 (3.79)	107.70 (6.61)

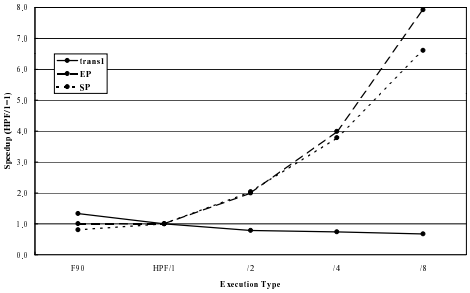

Fig. 22 Results of other benchmarks

schedule communications on each processor has a great impact on the performance. Especially in an array transposition operation that occurs in an intrinsic procedure *TRANSPOSE* or array remapping, communications are executed in sequence by each processor and its performance is not scalable. The array transposition operation must be sophisticated by a better scheduling algorithm such as that described in 19).

communication-schedule construction Our evaluation shows that constructing a communication schedule itself is a rather heavy task. The schedules of communication executed in such runtimes as *cshift* and *eoshift* are constructed every time when they are called, and they are very time-consuming. Besides the optimization of re-using a communication schedule at both compile-time and runtime, communication-schedule construction algorithm should be improved particularly for irregular communications.

5. Conclusion

HPF/SX V2 provides some unique extensions for vector parallel machines as well as the fea-

tures of HPF 2.0 and HPF/JA. In this paper, we described four of them: the `ON` directive, the `REFLECT` and `LOCAL` directives, vectorization directives, and automatic parallelization.

We compiled some benchmark programs using the features and evaluated their performance on an NEC SX-5. The results show that they achieve a 5-8 times speedup in 8-CPU parallel execution and it can be said that the four features are useful for vector parallel machines. We also evaluated over 30 well-known benchmarks, and about half of those show good performance. The other half suggest the compiler's runtime is not sufficiently tuned yet and should be more sophisticated.

We plan to strongly enhance HPF/SX V2 by studying these results in more detail. In addition, `INDIRECT` distribution,[3] HALO,[16] and `INDEX_REUSE` directives[5] are under development and will be supported in the next release. The HALO feature enables users to control communications of any pattern effectively to handle irregular problems. Cooperation with parallel execution on SMPs, task parallelism, parallel I/O, asynchronous communications, and tool support are also the targets of future work.

References

1) Message Passing Interface Forum, MPI: A Message Passing Interface Standard, June 1995.

2) Message Passing Interface Forum, MPI-2: Extensions to the Message-Passing Interface, July 1997.

3) High Performance Fortran Forum, High Performance Fortran Language Specification, January 1997.

4) S. Benkner and H. Zima, Compiling High Performance Fortran for distributed-memory architectures, Parallel Computing, vol. 25, pp. 1785-1825, 1999.

5) Japan Association of High Performance Fortran, HPF/JA Language Specification, `http://www.tokyo.rist.or.jp/jahpf/index-e.html`, 1999.

6) T. Brandes and F. Zimmermann, ADAPTOR - A Transformation Tool for HPF Programs, Programming Environments for Massively Parallel Distributed Systems, pp. 91-96, Birkhauser Verlag, 1994.

7) S. Hiranandani, K. Kennedy, and C. Tseng, Compiling Fortran D for MIMD distributed-memory machines, Communications of the ACM, vol. 35, no. 8, pp. 66-80, 1992.

8) S. Benkner, VFC: The Vienna Fortran Compiler, Scientific Programming, vol. 7, no. 1, pp. 67-81, 1999.

9) The Portland Group, PGHPF(R) Compiler for High Performance Computing (HPC) Systems, `http://www.pgroup.com/prodhpf.htm`.

10) H. Iwashita, S. Okada, M. Nakanishi, T. Shindo, and H. Nagakura, VPP Fortran and parallel programming on the VPP500 supercomputer, Proceedings of the 1994 International Symposium on Parallel Architectures, Algorithms and Networks (poster session papers), pp. 165-172, 1994.

11) H. Zima and B. Chapman, Supercompilers for parallel and vector computers, ACM Press, 1991.

12) Michael Wolfe, High Performance Compilers for Parallel Computing, Addison-Wesley, 1996.

13) Applied Parallel Research, Inc, APR's public domain benchmark suite, `ftp://ftp.infomall.org/tenants/apri/Benchmarks`, 1996.

14) Y. Charlie Hu, Guohua Jin, S. Lennart Johnsson, Dimitris Kehagias, and Nadia Shalaby, HPFBench: a high performance Fortran benchmark suite, ACM Transactions on Mathematical Software, vol. 26, no. 1, pp. 99-149, 2000.

15) K. Kinoshita, Hardware System of the SX Series, NEC RESEARCH & DEVELOPMENT, vol. 39, no. 4, pp. 362-368, 1998.

16) S. Benkner, HPF+ – High Performance Fortran for Advanced Scientific and Engineering Applications, Future Generation Computer Systems, vol. 15, pp. 381-391, 1999.

17) C. A. Addison, et al., The GENESIS Distributed-memory Benchmarks, Computer Benchmarks, J.J. Dongara and W. Gentzsch (Eds), Advances in Parallel Computing, Vol. 8, Elsevier Science Publications, BV (North Holland), Amsterdam, The Netherlands, pp. 257-271, 1991.

18) D. E. Bailey, et al., The NAS Parallel Benchmarks, Technical Report RNR-94-007, NASA Ames Research Center, 1994.

19) J. Choi, J. J. Dongarra, and D. W. Walker, Parallel Matrix Transpose Algorithms on Distributed Memory Concurrent Computers, Parallel Computing, vol. 21, no. 9, pp. 1387-1405, September 1995.

20) Thinking Machines Corporation, Cambridge, MA, CM Fortran Reference Manual, version 5.2-0.6 edition, September 1989.