Towards a Lightweight HPF Compiler

Hidetoshi Iwashita[1], Kohichiro Hotta[1], Sachio Kamiya[1],
and Matthijs van Waveren[2]

[1] Strategy and Technology Division, Software Group
Fujitsu Ltd.
140 Miyamoto, Numazu-shi, Shizuoka 410-0396, Japan
{iwashita.hideto, hotta, kamiya.sachio}@jp.fujitsu.com
[2] Fujitsu European Centre for Information Technology Ltd.
Hayes Park Central, Hayes End Road
Hayes UB4 8FE, UK
waveren@fecit.co.uk

Abstract. The UXP/V HPF compiler, that has been developed for the VPP series vector-parallel supercomputers, extracts the highest performance from the hardware. However, it is getting difficult for developers to concentrate on a specific hardware. This paper describes a method of developing an HPF compiler for multiple platforms without losing performance. Advantage is taken of existing technology. The code generator and runtime system of VPP Fortran are reused for high-end computers; MPI is employed for general distributed environments, such as a PC cluster. Following a performance estimation on different systems, we discuss effectiveness of the method and open issues.

Keywords: HPF, compiler, distributed parallel computing, MPI, VPP Fortran

1 Introduction

The progress of the most recent computer hardware is remarkable. Only several years ago, vector computers provided the high performance needed to tackle HPC problems. But now, multiprocessor systems with scalar CPUs, which are becoming cheaper and more rapid each year, are assuming this position. Even the latest communication equipment, such as InfiniBand, Gigabit Ether, IEEE1394, and USB2.0 is starting to catch up in speed with the special hardware networks which support distributed parallel processing. Moreover, it is not only the speed of such change but diversity that is the latest tendency. There is a variety of SMP, SMP cluster, and cc-NUMA architectures with various cash construction and memory hierarchies on the market with the objective of using multiple CPUs simultaneously and effectively.

We have developed *UXP/V HPF*, the HPF compiler for VPP800 and VPP5000 [1] series vector-parallel computers [2]. This compiler offers valuable results, which include the world record of performance in HPF applications [3]. This is due to the runtime system and to the compiler being expert enough in the characteristic of VPP hardware that it can pull out the maximum performance. However the hardware life-cycles are decreasing, and there are greater variations in the hardware, making it difficult to develop software specialized only in a specific hardware. Therefore, we need to rethink the development method, and reconstruct the compiler into layered modules in order to support multiple platforms. We also need technical breakthroughs in order to avoid performance falls due to simplistic generalizations.

As CPUs become cheaper and networks become quicker, the importance of parallel computing in a distributed environment is expected to increase. The most popular application interface to describe distributed parallel computing seems to be MPI [4] [5], which can be called a de-facto standard and includes all we need to do for parallel computing. However, even if MPI is useful for computer scientists and professional programmers, most HPC users seem to find it hard to use it to write real world applications. We don't think naked MPI will be the best answer to write parallel programs. We expect that MPI will be important not for programmers but for systems such as HPF compilers and parallelization support tools.

The Grid [6] is a recent remarkable technology as a platform of distributed environment. Many people expect it to increase in importance in the future.

OpenMP [7] is a language designed for a pure SMP environment and it does not have features to handle data locality. Therefore, it is not suitable for application in a distributed environment if no extensions are introduced. SGI and Compaq [8] have developed vendor-specific OpenMP language extensions in order to support data locality on their cc-NUMA architectures. The OpenMP Architecture Review Board (ARB) is discussing whether language extensions for distributed memory model are needed in OpenMP Version 3.0.

The SCore [9] technology contains a software distributed shared memory (SDSM) layer called SCASH, which works on distributed memory and which offers application software a view of shared memory. However, the OpenMP compiler for the SCore environment requires some language extensions to OpenMP to specify data locality in order to get a high performance [10].

VPP Fortran is an original data parallel language of Fujitsu and it is supported on all VPP series computers. Similar to the current HPF, it uses put/get communication that is supported on the VPP series, by their strong data transfer and hardware barrier facilities. Because the VPP Fortran language specification requires one-sided communication, it is difficult to adopt send/recv communication instead of put/get communication. We have the experience of implementing VPP Fortran on AP1000 and AP3000 distributed memory computers with the put/get communication method with little hardware support [11].

We wish to show in this paper that our HPF compiler can support multiple platforms without losing the performance. The structure of the HPF language processor is introduced in Section 2 and it is applied to VPP series vector-parallel computers in Section 3. In Section 4, development issues of the language processor on multiple platforms that are not restricted to the VPP series are discussed. Section 5 estimates the validity, and section 6 is the conclusion.

2 UXP/V HPF Compiler

UXP/V HPF system contains an HPF translator, which converts an HPF program into Fortran code, and a Fortran vector compiler. This section introduces the FLOPS compiler platform and some important passes in it, which constitute the HPF translator.

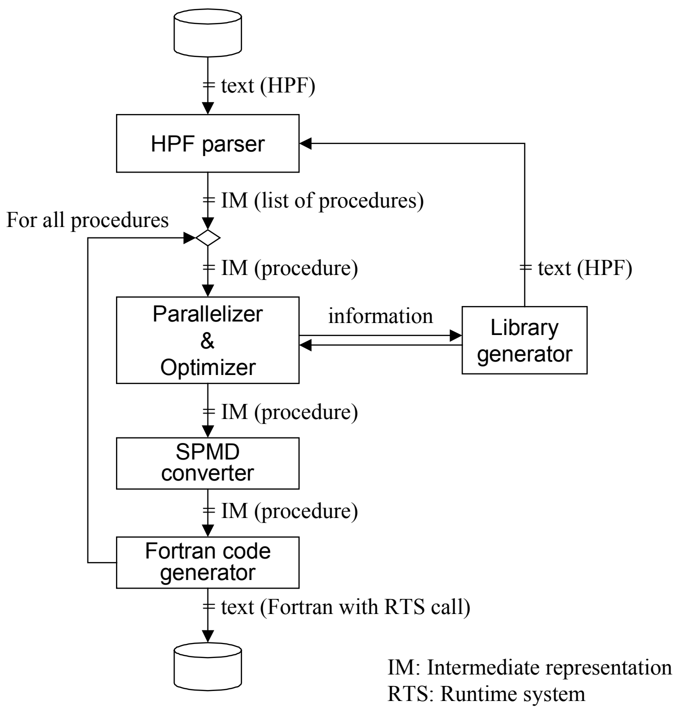


Fig. 1. Configuration of the FLOPS HPF compiler

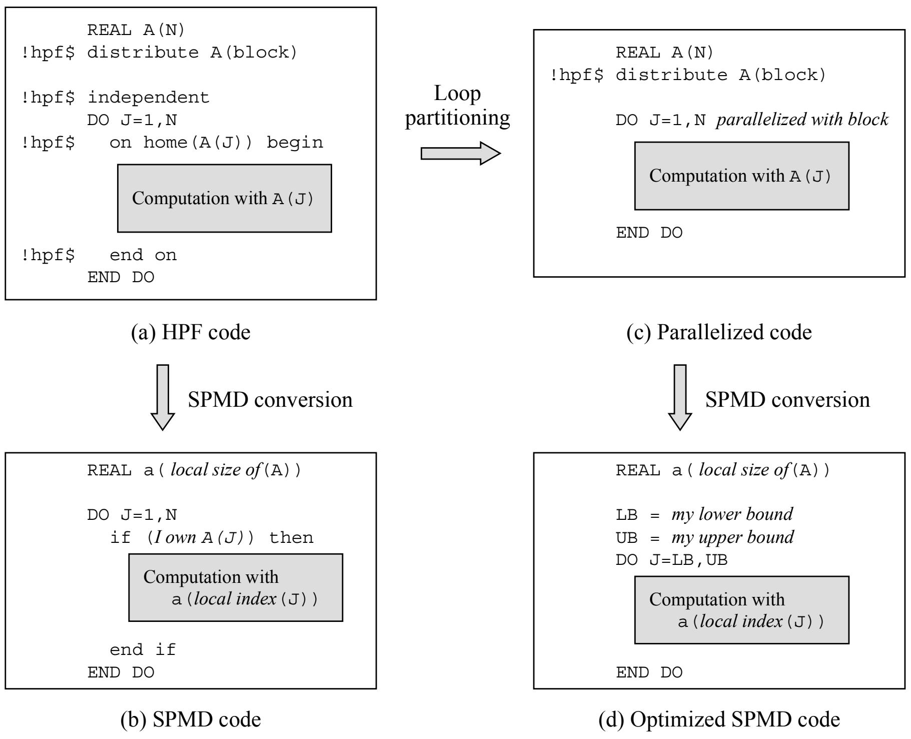
2.1 FLOPS Compiler

FLOPS (Fujitsu Labs' Optimizing and Parallelizing System) is the framework of source-to-source compilers whose main targets are distributed memory machines [12]. It is written in C and yacc (GNU Bison V1.27 for HPF). It has been used as a basis for parallel compiler products on AP3000 and VPP series computers since its research prototype was developed on the AP1000 scalar parallel computer [13]. The FLOPS framework defines the *intermediate representation (IM)* used in FLOPS compilers and provides access and utility functions onto the IM.

The configuration of the HPF version FLOPS compiler is shown in **Fig. 1**. IM is formed as C structures in product versions and can be input and output as a text of S-expression style in the research system. For all procedures (subroutines and functions) in the HPF source file, IM code is generated, parallelized, optimized, and finally output as a Fortran code.

2.2 SPMD converter

While HPF program code represents single thread execution and a global name space, *SPMD (Single Program/Multiple Data) code* represents execution and data for each processor. **Fig. 2** shows an example of the function of the SPMD converter.



```
      REAL A(N)
!hpf$ distribute A(block)

!hpf$ independent
      DO J=1,N
!hpf$   on home(A(J)) begin
            Computation with A(J)
!hpf$   end on
      END DO
```

(a) HPF code

```
      REAL A(N)
!hpf$ distribute A(block)

      DO J=1,N  parallelized with block
            Computation with A(J)
      END DO
```

(c) Parallelized code

```
      REAL a( local size of(A))

      DO J=1,N
        if (I own A(J)) then
            Computation with
              a(local index(J))
        end if
      END DO
```

(b) SPMD code

```
      REAL a( local size of(A))

      LB = my lower bound
      UB = my upper bound
      DO J=LB,UB
            Computation with
              a(local index(J))
      END DO
```

(d) Optimized SPMD code

Fig. 2. Parallelization and SPMD conversion

2.3 Parallelizer and Optimizer

The conversion of (a) to (b) in **Fig. 2** implements simplistically the ON HOME directive. Good performance cannot be expected for the following reasons:

- Each iteration of the loop will have an extra cost of condition and branch. Because the conditional expression depends on the loop variable, it is hard for the Fortran compiler to optimize it.
- The loop will not scale to the number of processors because the loop bounds are still global.

Though it might be possible to convert (b) to (d) as an optimization of the SPMD code, our *Parallelizer* finds loop partitioning corresponding to the ON directive in the source code (a) and generates an intermediate code (c). Note that it is not always possible to find a suitable loop partitioning even if the ON directive is specified, since the generalized clause `home(A(`*f*`(J)))` requires computation of the inverse function of *f* and the relation of ON and DO constructs may not be simple.

Other functions of the *Parallelizer*, such as the generation of ON directives and the searching of loop independencies, makes automatic loop parallelization available in simple cases.

The *Optimizer* reduces the number of interprocessor communications and runtime system calls and arranges blocks of communications into asynchronous data transfers.

2.4 Library Generator

The *Library Generator* [14] & [15], which works at compile time, generates optimized parallel code for the HPF Library and Fortran 95 transformational intrinsic functions. It accepts the name of the target function and characteristics of the arguments (such as type, rank, size, and distribution kind), then generates optimized code suitable for the target function in the form of an HPF subprogram, and finally passes the code to the HPF parser. This implementation solves the problem of the enormous number of specific functions in HPF Library due to the large number of combination of characteristics of the arguments, which makes that an optimized library would have an impossible size.

The generated code is not expanded into the source code using inline-expansion but it is linked as a subprogram with the source code. The name of the function as referred to in the source code is changed to the corresponding name in the subprogram. We called this method *online-expansion*. Online-expansion has the following merits compared with inline-expansion:

- The compilation time increases less because the online-expansion does not increase the size of each program unit in the source code. Optimization processes of the Fortran compiler sometimes spend $O(n^2)$ or $O(n^3)$ of computation time for a program of size n.
- Online-expansion is available even in contexts in which inline-expansion is difficult.
- The automatic remapping facility at the entry and exit points of the HPF subprogram is useful.

3 Application to VPP Series

This section describes features of the VPP5000 hardware and how the current HPF system utilizes them.

3.1 Distributed Memory Machine VPP5000

The VPP5000 is the latest generation of Fujitsu vector-parallel supercomputers [1]. It is a distributed memory machine with up to 512 processor elements (PE) connected with a high-speed *crossbar network*. The throughput of the crossbar is 1.6 Gbyte/s for both input and output of each PE. Each PE has a vector unit of 9.6 Gflop/s, a scalar unit with VLIW RISC architecture, and up to 16 Gbyte of 45 ns SDRAM memory.

The *Data Transfer Unit (DTU)*, which the VPP series computers have in each PE, can directly access local memory and communicate with other DTUs. The DTU enables hardware put/get communication through the crossbar network without interrupting any CPUs. The DTU recognizes remote data through the virtual global address. In order to write data into remote memory (i.e., the put communication), the DTU reads data from local memory with the specified access pattern and sends the data to the remote DTU. The remote DTU then writes the data into its local memory using the given global address and stride pattern. In order to read remote data into local memory (i.e., the get communication), the DTU sends a request packet to the remote DTU, asking him to send back the specified data as a put communication.

The DTU can handle packets with a two-dimensional stride, which support at least two lower dimensions of a Fortran 90 array section. For example, the array section `A(i1:i2:i3,j1:j2:j3,1:n)` can be sent and received with only `n` packets (unless the packet size exceeds the limit) of zero-copy communication without interrupting any CPU.

3.2 Runtime system

The parallel runtime system (RTS), which is called from the generated code of the HPF translator, was made solely for the VPP800 and VPP5000 series. Parallel RTS has two main features, parallel execution management and interprocessor communication. From our experience with VPP Fortran [16], we applied the put/get communication method in RTS in order to extract the highest performance of the VPP800/5000.

By taking advantage of the rich features of DTU and crossbar network topology, RTS achieved almost the same throughput as the hardware peak performance of 1.6 GB/s/PE for big and regular data transfer, such as the transpose of a block-distributed array. RTS can send not only contiguous data, but can also combine some Fortran 90 array sections into one packet for sending and receiving with the DTU.

3.3 Characteristics of Communication on VPP

While large array data can be treated well on the VPP, applications that include many irregular accesses of small data tend to be inefficient. The cause of low performance is a relatively high latency, several 10s of microseconds in total with software and hardware overhead. Therefore, if the size of each packet is not much greater than 16KB (=10 [microsecond] x 1.6 [GB/second]), the latency dominates the throughput speed.

Unlike send/recv communications that imply loose synchronization, the put/get communication method often requires barrier synchronization in order to confirm if the remote storage can be referred and overridden. In order to support such frequent synchronizations, the VPP has a high-speed hardware barrier facility.

4 Multi-platform Development

This section discusses the possibility of developing compilers for multiple platforms starting from the current HPF compiler on VPP series.

4.1 Development for High-end Computers

Fujitsu has designed a data parallel language VPP Fortran and provides a compiler for it [16]. Even though two program codes written in VPP Fortran and in HPF/JA [17] reach almost the same peak performance, the RTS of HPF is 20% larger than the RTS of VPP Fortran. The difference is caused by the variety of data mapping in HPF (e.g., block cyclic mapping, indirect mapping, replication of partially distributed array, alignment with stride, scalar template, and alignment between non-distributed arrays) and by the dynamic management (e.g. automatic remapping at subprogram entry, redistribution and realignment while keeping the linkage of alignment, etc.). The heaviness of HPF RTS increases the cost of initialization at the entry points of subprograms. The cost becomes more important in the case of smaller applications.

Assuming that the VPP Fortran compiler will exist for the future high-end computers, we are considering to replace the SPMD converter and the succeeding passes of the HPF compiler and the RTS of HPF with those of VPP Fortran, as shown in **Fig. 3**. The following issues must be solved:

- The IM converter in **Fig. 3** is needed. -- It would convert IM generated by the parallelizer and the optimizer into a form that is acceptable to the VPP Fortran compiler. Both IMs of HPF and VPP Fortran are basically compatible with the exception of some small differences.
- Some features of HPF that should be supported in RTS are not supported in the RTS of VPP Fortran. -- They will be supported in the parallelizer and optimizer of the compiler as much as possible so that the enhancement of RTS will be minimized. For example, most redistribution can be solved at compile time with flow analysis and program conversion.

Especially the second issue requires more research. We have had a good experience with a highly tuned HPF/JA program, which can be shown to work well on the RTS of

VPP Fortran. This is described in Section 5. Because the VPP Fortran RTS is lighter than the HPF RTS, the resulting system will be lightweight.

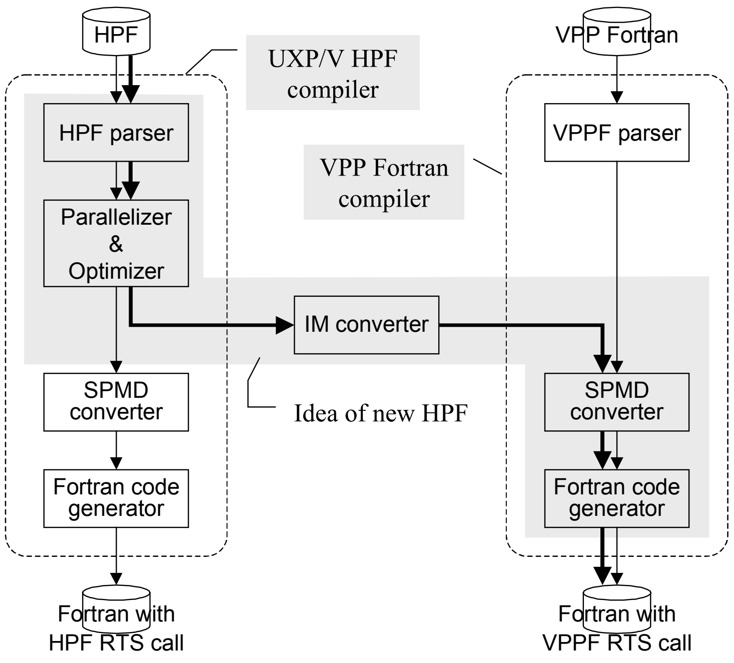


Fig. 3. HPF compiler on high-end platform

4.2 Development for A General Distributed Environment

Here we discuss how the UXP/V HPF compiler can be adapted to a general distributed environment that consists of multiple CPUs connected by a communication network, including PC clusters. Instead of RTS, we are trying *MPI* [4] as a communication layer. MPI is used on many distributed platforms. Implementation using MPI must be one of the following:

- Development of an RTS that calls MPI. The interface of the RTS will be changed from the current RTS.
- Changing the code generation so that the code contains direct MPI calls instead of RTS calls.

In both cases, most of the current passes including HPF parser, parallelizer, optimizer, and library generator can be used unmodified, but the SPMD converter must be modified. The advantage of the latter case is the portability of the compiler system because the RTS does not need to be recompiled for the different platforms. In the latter case, however, it is unclear if the runtime environment such as IDs of the active processors can be managed in the generated code without resorting to RTS. An online expansion technique similar to the library generator might be used.

While the put/get communication is available in MPI-2 [5], we adopt at first send/recv as the primitive of communication since it is more popular and is already evaluated on many platforms. Put/get communication is expected to be a good alternative in some cases.

The efficiency of communication is an open issue, since we do not rely on special communication hardware such as those of VPP. Communication aggregation will be a key technique in future development.

5 Performance Estimation

5.1 Performance Estimation for High-end Computers

This section estimates how the replacement of SPMD converter and RTS affects the peak performance of an HPF program for the high-end implementation. We have tuned the NAS Parallel BT benchmark code [18] on VPP5000 with HPF/JA language extensions and vector optimization [2]. In order to compare with this result, we estimated the performance of the generated code of the new compiler for high-end machines shown in **Fig. 3**. Instead of using the SPMD converter, we inspected the output code of the current HPF compiler and made the VPP Fortran compiler generate almost the same code, using VPP Fortran programming. This work is in effect an emulation of the ideal function of the SPMD converter. The conversion of HPF to VPP Fortran is described in **Table 1**.

The result of the comparison is shown in **Fig. 4**. For all data size classes S, A, B, and C, the new compiler was estimated to give a higher performance than the current compiler. The difference tends to become larger in smaller data size. To our impression, this is because the current HPF has the following expensive portions:

- Initialization at the entry points of user subprograms corresponding to dummy arguments.
- Handling of a variety of data mapping in RTS.

As a conclusion of this performance estimation, we confirmed that the intermediate code, which includes main features of HPF, can be translated into code that calls RTS of VPP Fortran. Because RTS of VPP Fortran is lighter than RTS of HPF, we expect that the resulting code will have a higher performance. Though they are not used in this tuned benchmark code, HPF has important features that VPP Fortran does not support, such as redistribution. Such features must be carefully implemented in order to keep the generated code and RTS lightweight. If the new compiler supports them with little help of RTS, it achieves almost the same performance as the VPP Fortran compiler on the same high-end computer.

Table 1. Conversion from HPF to VPP Fortran

HPF language items	Corresponding VPP Fortran language items
Data mapping with PROCESSORS, DISTRIBUTE, TEMPLATE, ALIGN, SHADOW, and SEQUENCE	Corresponding combination of PROCESSOR, INDEX PARTITION, GLOBAL, and LOCAL
INDEPENDENT directive	SPREAD DO directive with the loop decomposition generated by the HPF compiler
ON HOME construct and RESIDENT for execution of single processor	SPREAD REGION construct
ASYNCHRONOUS construct (in HFP/JA extension)	EQUIVALENCE (of local variable to global variable) and SPREAD MOVE construct
Asynchronous REFLECT directive (in HFP/JA extension)	OVERLAPFIX directive
Access of sequential and unmapped variables	Specified as LOCAL and careful manual maintenance of data consistency

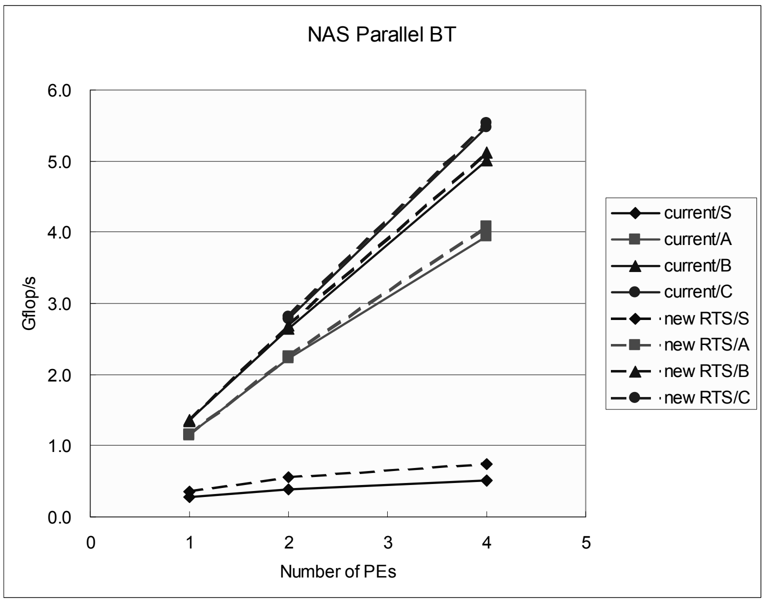


Fig. 4. Estimation of HPF compilers for high-end machines

5.2 Estimation of HPF calling MPI

This section estimates the characteristics of the performance of the new HPF compiler that employs MPI as a communication primitive. Using the medium size SPEC OMPM2001 SWIM benchmark [19], we made the following two executable codes and evaluated them on the VPP5000.

HPF on RTS	An HPF program compiled with the current HPF compiler, which employs effective RTS developed only for VPP800 and VPP5000
HPF on MPI	An SPMD program with MPI, written manually with the purpose of estimating the performance of code that the new compiler will generate

Fig. 5 shows the result. The new HPF on MPI was estimated to give better performance than the current HPF on RTS. This estimation does not guarantee that the new compiler gives high performance for all application programs, but offers us very good prospects.

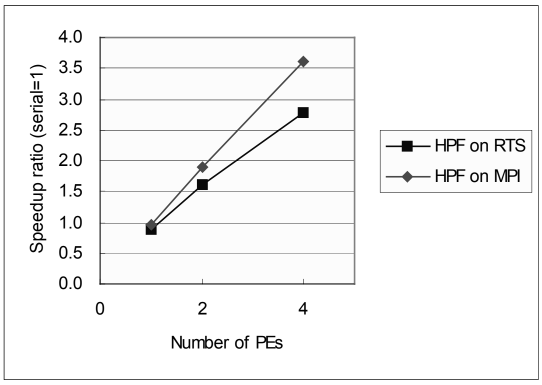
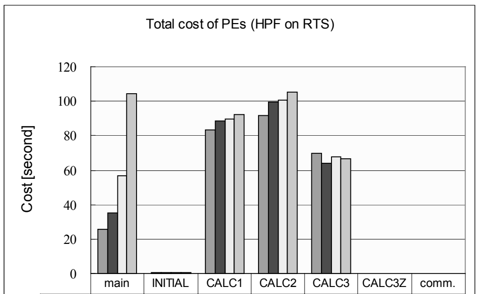


Fig. 5. Estimation of HPF compilers using MPI

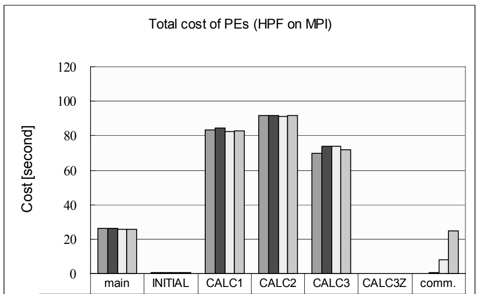
Using the analyzer tool, we measured the cost distribution. **Fig. 6** displays the total cost for each procedure of the benchmark program using all employed processors. Since the costs were measured on the basis of elapsed time, it means that there is a good load distribution and little overhead in the subprogram with the result that its cost is not much greater than the one of serial execution. In the MPI version, the column comm shows the total costs of MPI communication and the calling of MPI; in RTS version, the communication cost is included in each subprogram. We conclude from the performance estimation the following:

- The RTS version has a large cost in the main program, and this may break the scalability of the parallel execution. The cost (elapsed time) includes allocation to the virtual global memory, broadcasting the global addresses, and interprocessor fork operation with large initialized data. Such initialization costs can be ignored in the typical long-term jobs that appear on the VPP. However, if we take short-term execution without a high-speed network into account, it should be improved.

- In contrast, the MPI version shows perfect load balancing in the main program, in which all processors do not have to do extra work (except the communication cost summed up into comm). This is because the implementation does not use virtual global memory or global variables. The initialization cost of the MPI environment is not visible in this measurement on VPP5000 but it may appear in measurements on other distributed memory environments.
- The communication cost of the MPI version increases more than twice between two and four processors. We use in this performance estimation only MPI_SEND, MPI_RECEIVE, and a collective communication MPI_SENDRECV for the purpose of interprocessor communication. So, more improvement might be possible if we use other functions such as MPI_REDUCE and non-blocking communication and if we take account of communication scheduling.
- We wrote an MPI version of the program, in which MPI functions are not directly called from the source code but called from a shell, which itself is called from the source code. The total cost of shell routines was trivial (only 0.07 second) compared to the whole cost of the program.



	main	INITIAL	CALC1	CALC2	CALC3	CALC3Z	comm.
▨ serial	25.80	0.40	83.07	91.68	69.45	0.04	0.00
■ 1 PE	35.34	0.41	88.40	99.72	63.79	0.04	0.00
□ 2 PEs	56.56	0.41	89.85	100.38	67.55	0.04	0.00
▨ 4 PEs	104.18	0.40	92.17	105.28	66.32	0.04	0.00



	main	INITIAL	CALC1	CALC2	CALC3	CALC3Z	comm.
▨ serial	25.97	0.40	83.07	91.68	69.45	0.04	0.00
■ 1 PE	25.95	0.40	84.36	91.62	73.94	0.04	0.62
□ 2 PEs	25.79	0.40	82.44	91.04	74.00	0.05	8.08
▨ 4 PEs	25.89	0.40	82.70	91.55	71.96	0.04	24.47

Fig. 6. Cost analysis of current and new HPF compilers

6 Conclusion

The current UXP/V HPF compiler has achieved a high performance by using the strong data transfer and hardware barrier facilities of the VPP800/5000 effectively. The performance relies on high-level user tuning and a sufficient size of the problem. In order to develop compilers for a general distributed environment, we cannot rely on these facilities, high-level user tuning and large problem sizes. We have discussed how to develop compilers for multiple platforms starting from the current HPF compiler on VPP series. The keyword is *lightweight*. Technically, it is necessary to make the following items lightweight:

- Initialization cost at the beginning of the program and entry point of the subprograms, and
- Runtime information managed by RTS and the variety of data mapping handled by RTS.

In addition, the following lightness is also required:

- Reducing the user's (especially beginner's) load, in order to get a reasonable performance without perfect user tuning, and
- Reducing the redundancy of the development, in order to quickly support many platforms.

We have discussed performance estimations of the lightweight system on both high-end computers and on general distributed systems.

We consider common techniques such as MPI, Grid, and SCore as communication primitives. Instead of using put/get communication, loose synchronization, a feature of send/recv communication, will reduce the barrier synchronization and create opportunities for pipelined parallelism.

References

[1] VPP5000 Series. http://primepower.fujitsu.com/hpc/en/vpp5000e/index.html

[2] Hidetoshi Iwashita, Naoki Sueyasu, Sachio Kamiya, and Matthijs van Waveren. VPP Fortran and the Design of HPF/JA Extensions, *Concurrency: Practice and Experience.* To be published.

[3] Tatsuki Ogino. Global MHD Simulation Code for the Earth's Magnetosphere Using HPF/JA, *Concurrency: Practice and Experience.* To be published.

[4] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI - The Complete Reference Volume 1, The MPI Core.* MIT Press, Cambridge, Massachusetts, 1999.

[5] Message Passing Interface Forum. http://www.mpi-forum.org/

[6] Global Grid Forum. http://www.gridforum.org/

[7] OpenMP Architecture Review Board. http://www.openmp.org/

[8] John Bircsak, Peter Craig, RaeLyn Crowell, Jonathan Harris, C. Alexander Nelson, and Carl D. Offner. Extending OpenMP for NUMA Architectures. In *WOMPAT2000*, San Diego, CA, July 2000.

[9] Real World Computing Partnership. SCore Cluster System Software. http://pdswww.rwcp.or.jp/score/dist/score/html/index.html

[10] Mitsuhisa Sato, Hiroshi Harada, Atsushi Hasegawa, and Yutaka Ishikawa. Cluster-enabled OpenMP: an OpenMP compiler for Software Distributed Memory System SCASH. In Proceedings of *JSPP2001*, pp.15-22, 2001. (In Japanese)

[11] Tatsuya Shindo, Hidetoshi Iwashita, Tsunehisa Doi, and Junichi Hagiwara. An implementation and evaluation of a VPP Fortran compiler for AP1000. In *IPSJ SIGNotes High Performance Computing*, No.048-002, 1993.

[12] Tatsuya Shindo, Hidetoshi Iwashita, Tsunehisa Doi, Junichi Hagiwara, and Shaun Kaneshiro. HPF Compiler for the AP1000. In *1995 International Conference on Supercomputing*. pp. 190-194, 1995.

[13] AP3000 Homepage. http://primepower.fujitsu.com/hpc/en/ap3000-e/index.html

[14] Matthijs van Waveren, Cliff Addison, Peter Harrison, David Orange, Norman Brown, and Hidetoshi Iwashita. Code Generator for the HPF Library and Fortran 95 Transformational Functions, *Concurrency: Practice and Experience.* To be published.

[15] Matthijs van Waveren, Cliff Addison, Peter Harrison, David Orange, and Norman Brown. Code Generator for HPF Library on the Fujitsu VPP5000. *Fujitsu Scientific and Technical Journal,* vol. 35, no. 2, pp. 274-279, 1999. http://magazine.fujitsu.com/us/vol35-2/paper17.pdf

[16] Hidetoshi Iwashita, Shin Okada, Makoto Nakanishi, Tatsuya Shindo, and Hiroshi Nagakura. VPP Fortran and Parallel Programming on the VPP500 Supercomputer, In poster session proceedings of *1994 International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN'94)*. pp.165-172, 1994.

[17] Japan Association for High Performance Fortran (JAHPF). *HPF/JA Language Specification Version 1.0*, November 1999. http://www.hpfpc.org/jahpf/spec/hpfja-v10-eng.pdf

[18] NAS Parallel Benchmarks. http://www.nas.nasa.gov/Software/NPB/

[19] SPEC OMP2001 Benchmark Suite. http://www.spec.org/hpg/omp2001/