A New Optimization Technique for the Inspector-Executor Method

Daisuke Yokota
Doctoral Program in Engineering
University of Tsukuba
Tsukuba, Ibaraki, Japan
email: daisuke@hlla.is.tsukuba.ac.jp

Shigeru Chiba
Department of Mathematical and Computing Sciences
Tokyo Institute of Technology
Meguro-ku, Tokyo, Japan
email: chiba@is.titech.ac.jp

Kozo Itano
Department of Information Science and Electronics
University of Tsukuba
Tsukuba, Ibaraki, Japan
email: itano@is.tsukuba.ac.jp

ABSTRACT
This paper presents our HPF compiler using our modified inspector-executor method for implementing accesses to a distributed array. In our modified method, a compiler runs an inspector during compile time to obtain the information of data dependency among node processors, and it uses that information to optimize communication code included in the executor. This paper presents our idea, performance improvement shown by our prototype compiler, and limitations of our method.

KEY WORDS
distributed memory machine, compiler, optimizing communications, inspector-executor

1 Introduction

Computer simulation is nowadays a significant research tool in natural sciences like physics and astronomy. Researchers use high-performance parallel computers with more than a thousand processors and they simulate natural phenomena. For example, their programs repeatedly calculate the energy of every particle in a simulated space at every time tick so that they could investigate a new physical model.

Since their programs exchange a large amount of data between node processors, the ability of compilers to optimize inter-processor communication is significant. In this application domain, long compilation time for strong optimization is acceptable; a single run of the compiled code often takes a week or more. Manual optimization of inter-processor communication by programmers is not realistic. Since the programmers are not computer engineers but usually physicists or astronomers, we should not expect that they have detailed knowledge of underlying hardware or complicated programming magic.

This paper presents our Fortran compiler providing HPF-like directives [8]. Our compiler uses the inspector-executor method [3] for producing communication code for exchanging data among node processors. A unique feature of our compilation is that data dependency among node processors, which is reported by the inspector, is used for statically optimizing the communication code in the executor. In the current implementation, our compiler first produces only inspector code and then runs it as part of compilation process. After the inspector reports the data dependency among node processors, our compiler collects that dependency information and uses it for producing executor code optimized with respect to inter-processor communication. This optimization includes constant folding at compile time. The final compiler-output includes only the executor code statically optimized with the given data dependency. The users run that executor for their simulation. Due to this architecture, our compiler cannot correctly compile a program if data dependency in the program changes during runtime.

This somewhat odd compiler has been experimentally developed for studying the feasibility of our idea. Our research goal is to develop a compiler producing executable binary that can dynamically recompile the executor to be optimal if the inspector reports that data dependency is changed during runtime. However, we believe that the current prototype of our compiler is already useful in practice. According to our interview with our users, who are natural scientists, typical simulations for computational sciences show complex data dependency that the inspector-executor method is needed to analyze. However, the dependency is not chaos; there is some regularity that can be used for static optimization. Also, the data dependency never changes during simulation.

The organization of the rest of the paper is as follows. Section 2 presents overview of our compiler architecture. Section 3 shows optimization techniques that our compiler applies. Section 4 mentions the flow of the compilation with our compiler. Section 5 illustrates the performance of the optimization by our compiler. Section 6 discusses related work and Section 7 concludes this paper.

2 Our Compiler Architecture

Our compiler accepts a Fortran program written with a subset of HPF directives. The target machine is CP-PACS and Pilot3 [13], which are parallel computers of Center for Computational Physics at University of Tsukuba. CP-PACS has 2048 processors and Pilot3 has 128 processors. CP-PACS and Pilot3 are not shared memory machines; every node processor has local memory. The node processors are connected with each other by hyper crossbar network, which is three dimensional in CP-PACS and is two dimensional in Pilot3. The theoretical peak performance of CP-PACS is 614Gflops.

Since CP-PACS and Pilot3 have unique hardware, called *remote DMA (Direct Memory Access)*, for network communication, the executable binary generated by our compiler exploits that hardware to improve the execution performance. (1) Remote DMA enables to transfer data between node processors with the minimum software intervention. The receiver processor does not have to explicitly read network port and store the read data on memory. (2) Remote DMA also enables *block-stride communication*, in which the transferred data does not need to be stored on contiguous memory area. The hardware can read multiple data blocks placed on memory at regular intervals and transfer to network as a single packet. (3) The last feature of remote DMA is communication reusing TCWs (transfer control words). If a node processor repeatedly transfers data to the same memory address on the same destination node processor, it can set up the network hardware in advance and performs low-latency data transfer.

Our compiler automatically distributes elements of array data to each node processor according to HPF directives embedded in the source program. Data exchanges among processors are implemented by an extended inspector-executor method we have developed. The inspector-executor method is appropriate for parallelizing a program performing complicated data access.

A typical simulation for computational sciences is described as a program consisting of nested loop statements. We call them the *inner* loops and the *outer* loop (Figure 1). The inner loops are for computing new values of array elements from old values. Some of them are marked with the `independent` directive. The compiler may parallelize those loops with the inspector-executor method. On the other hand, iterations of the outer loop are sequentially executed. They increment the clock of the simulated time and invokes the inner loops.

In the original inspector-executor method, the body of the inner loop is compiled so that it is executed in two stages. The first stage is called *inspector* and the second is called *executor*. The inspector goes through the loop body and records when and which array elements are accessed during the execution. It does not actually perform inter-processor communication. Then, the executor runs for performing inter-processor communication according to the data dependency obtained by the inspector. It also

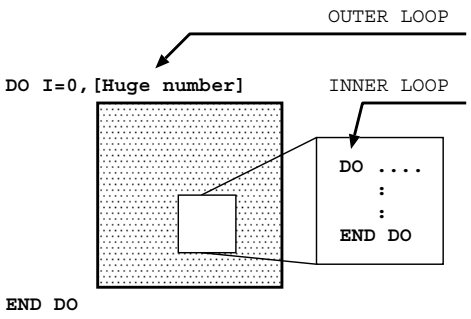

Figure 1. The inner loop and the outer loop

performs computation described in the loop body.

Since both the inspector and the executor are run at runtime, optimizing network communication is not easy with the original inspector-executor method. If the programmer guarantees that the data dependency between processor nodes in the inner loop never changes among the iterations of the outer loop, the inspector can be run only once at the first iteration of the outer loop. Although this significantly reduces the runtime overheads due to running the inspector, the executor still includes runtime overheads. The executor must access the data-dependency information obtained by the inspector and decide what array elements node processors must exchange with each other. This runtime inquery decreases the execution performance.

Our compiler separately generates the inspector code and the executor code. It first generates the inspector code and runs it as a stage of compilation. Then it analyzes the data-dependency information obtained by the inspector and it finally generates the executable binary of the compiled source program, which includes only the executor code. Since our compiler can apply various static optimization techniques to the executor code, the executable binary can include statically optimized executor code. Since the executable binary does not include the inspector code, our compiler requires the programmers to guarantee that the data dependency does not change among iterations of the outer loop. We do not think that this feature is a major defect of our compiler because typical simulations for computational sciences satisfy this feature.

For deeply analyzing data dependency, we have implemented our compiler on a parallel computer, which is not CP-PACS but a PC cluster. Our compiler runs an inspector in parallel as the original inspector-executor method runs an inspector in parallel. However, the inspector generated by our compiler performs more aggressive examination of data dependency as presented in the next section.

3 Optimization

Our compiler applies three optimization techniques to the executor code with respect to network communication. They are constant folding, packing multiple messages into a single one, and optimal allocation of loop iterations.

3.1 Constant folding

Our compiler performs constant folding as much as possible. In a typical implementation of the original inspector-executor method, an inspector produces a table representing data dependency. An executor looks up that table at runtime to get the node processors that array elements must be sent to or received from. Our compiler eliminates this table lookup since the contents of the table are constant. The table lookup is replaced with a linear expression including only loop index variables and constants (Figure 2).

If the node processors that array elements are exchanged with are statically known, an executor can improve the speed of data transfer by reusing TCWs (Transfer Control Words). TCWs are data blocks specifying the destination of a network packet. The executor can prepare TCWs in advance to avoid redundantly preparing TCWs at every iteration of the loop (Figure 3).

```
DO I=1, n
    IF (TABLE[I]) THEN
        SETTING(TABLE[I])
        SEND(TABLE[I])
    END IF
        :
        :
END DO
```

a) before optimization

```
SETTING(CONST/SIMPLE EXPR)
SEND(CONST/SIMPLE EXPR)
DO I=1, n
        :
        :
END DO
```

b) after optimization

Figure 2. eliminating table lookup

```
DO I=1, n
      :
    SETTING(CONST in this LOOP)
    SEND(CONST in this LOOP)
      :
END DO
```

a) before optimization

```
ID=SETTING(CONST in this LOOP)
    :
DO I=1, n
      :
    SEND(ID)
      :
END DO
```

b) after optimization

Figure 3. Reusing TCWs

3.2 Packing multiple messages

Our compiler merges several messages sent through network into a single message if possible. This reduces the number of messages and thus improves average throughput because every message implies inherent handling overhead.

Messages sent in the iterations of the loop with the `independent` directive can be merged into a single message if they are transferred from the same source to the same destination. The `independent` directive specifies that the result of the execution of that loop is independent of the execution order of the iterations. In other words, the iterations can be executed in parallel.

Our compiler examines the source and destination of the messages sent in the iterations. To do this, it uses the data-dependency information obtained by the inspector. Then, it collects the messages sent from the same source to the same destination. Our compiler merges messages if the transferred data are stored at contiguous memory addresses. The hardware of our target machines, called remote DMA, directly transfers such data without extra memory copies. Our compiler also uses the block-stride communication provided by the remote DMA. If the transferred data are stored in memory blocks at regular intervals, our compiler also merges messages. The hardware packs those memory blocks into a single message to send. At the destination, the hardware unpacks the message and stores the transferred data on memory at regular intervals.

3.3 Allocating iterations

Our compiler distributes array elements to node processors according to HPF directives specified by the programmers. The whole array is divided into multiple blocks and every node processor holds only one of the blocks on the local memory. The programmers are responsible for how the array is divided and which node processor holds each block.

If our compiler encounters a loop statement with the `independent` directive, it also distributes iterations of that loop statement to node processors. However, the iterations are distributed without the programmers' support.

Suppose that a `do` loop iterates its body n-times. If the number of node processors is N, a naive distribution of the loop iterations is that every node processor executes contiguous $\frac{n}{N}$ iterations. However, this naive distribution may not be optimal with respect to the amount of data exchanged among node processors through network.

To optimize the amount of the exchanged data, our compiler can allocate adjacent iterations to different node processors. It selects the best node processor that each iteration is allocated to under the data distribution specified by the programmer. To do this, the inspector examines all the possible allocations of loop iterations and selects the best one.

If loop statements with the `independent` directive are nested, our compiler parallelizes only one loop statement so that it minimizes the amount of data exchanged among node processors; it does not parallelize the others. Our compiler examines all the loop statements for selecting the best one.

For each iteration and each node processor, the inspector computes the amount of exchanged data during the iteration in the case that the node processor executes that iteration. Then, the inspector selects a node processor for every (from the first to the last) iteration. (1) For the i-th iteration, if the inspector finds the single best node processor that minimizes the amount of exchanged data, that iteration is allocated to that node processor. (2) If the inspector finds multiple best for the i-th iteration, the iteration is allocated to the node processor that executes the $(i-1)$-th iteration

among the best ones. (3) If no node processor executes the $(i-1)$-th iteration, the iteration is allocated to the node processor that executes the smallest number of iterations among the best ones. (4) Otherwise, if there are multiple candidates in the rule (3), then the iteration is allocated to the node-0 processor.

In this algorithm, all the iterations can be allocated to the single node processor. Although this allocation means that those iterations are sequentially executed, our compiler selects this allocation since parallel execution of those iterations would cause serious communication overheads.

4 Implementation

Our compiler accepts Fortran 77 with some of the HPF directives [9], which are `independent`, `processor`, and `distribute(block)`. It also supports our original directive `outer`, which are used to explicitly specify outer loops.

Figure 4 illustrates the flow of the compilation. Our compilers run on a PC-cluster computer so that several compilation stages are executed in parallel and thus aggressive optimization is feasible. (1) First, a source program is sent to all the node PCs, which generate the inspector code. (2) Then, the inspectors runs on every node PC. It produces the log including the data dependency information of the compiled program. (3) The produced log is analyzed on every node PC. The amount of exchanged data is computed for all the possible allocations of iterations of the inner loops. (4) The results of the previous stage are collected on the master node PC. Then the master PC determines the allocation of the iterations to node processors of the target machine. (5) The determined allocation is sent to all the node PCs. Each node PC attempts to pack multiple messages into one. (6) All the information of messages sent from every node processor is exchanged through the master node PC. (7) After that, every node PC generates final code including the executor code. (8) All the final code is collected on the master node PC, which merges it into a SPMD code. This SPMD code is compiled by a backend Fortran compiler into executable binary.

The last stage (8) is necessary since the backend compiler of our target machine only accepts a SPMD program. At execution time, our target machine loads a single binary image into all the node processors. Although the simplest way to produce a SPMD program is to just concatenate all the programs for every node processor, it extremely increases the length of the produced program. Hence our compiler merges those programs at statement level so that the length of the produced program is acceptable.

Our compiler can correctly compile only the programs satisfying the following restrictions. First, the data dependency among node processors must not change for every iteration of the outer loop. Second, the data dependency must be independent of the values of the array elements distributed to every node processor. For example, our compiler cannot correctly compile the IS program in the NAS parallel benchmarks.

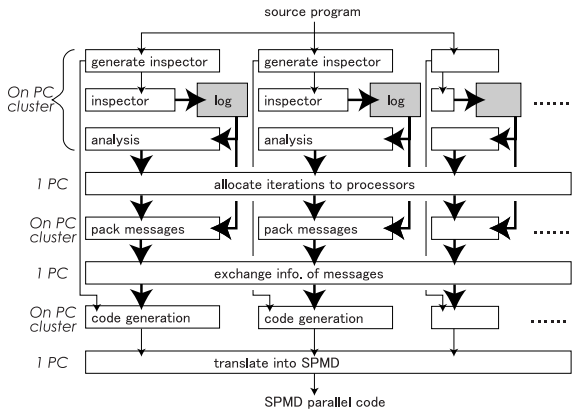

Figure 4. The flow of compilation

5 Experiment

We measured the execution performance of a few benchmark programs compiled by our compiler. The benchmark programs are pde1 from the genesis distributed benchmarks[10],[12], FT and BT from the NAS parallel benchmarks[11],[12]. The task size parameter N of pde1 was 7. FT and BT were class A.

The target machine was Pilot3 with 16 nodes processors. The PC-cluster computer we used for compilation consisted of 16 node PCs, which run Redhat 7.1 Linux on PentiumIII 733MHz with 512 Mbyte memory. The backend compiler was a commercial Fortran compiler by Hitachi (version 02-06-/C + 02-06-XF + 02-06-XJ) running on a single node of Pilot3.

5.1 Execution time

We first shows comparison between the execution time of the compiled executable code and the number of node processors (Figure 5, 6, and 7). In these figures, we also showed the execution time of the executable code compiled by a commercial HPF compiler (version 02-05) by Hitachi for the target machine. Our compiler showed definitely better results against Hitachi's compiler with respect to both the absolute execution time and the scalability. This is because Hitachi's compiler is a traditional optimizing compiler and hence it cannot generate efficient executable code without detailed control by the programmers with more HPF directives than we inserted in the benchmark programs.

Figure 7 also shows the execution time of the executable code compiled with the original inspector-executor method. This executable code dynamically performs optimization equivalent to one presented in Section 3.2 although our compiler statically performs it. At runtime, this code tries to dynamically pack multiple messages. Thus, the difference between our compiler and this inspector-executor method is effects of constant folding.

Figure 7 shows that our compiler was 4.9% faster than this inspector-executor method when 16 node processors are used.

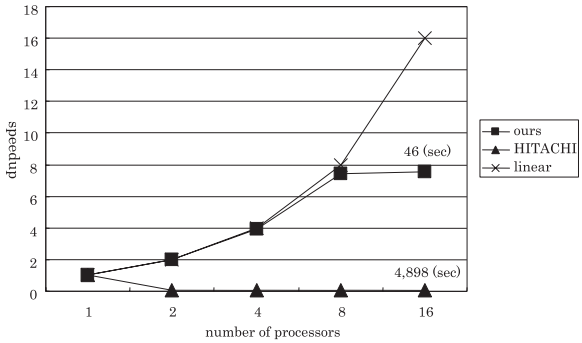

Figure 5. FT-classA Speedup

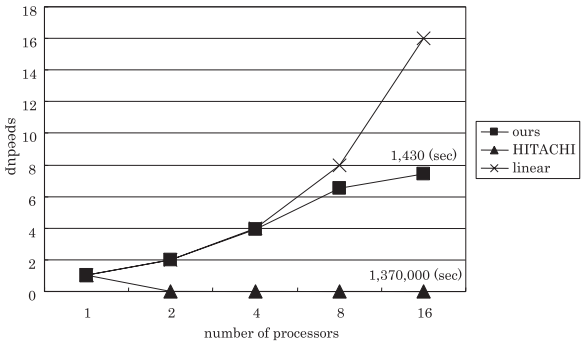

Figure 6. BT-classA

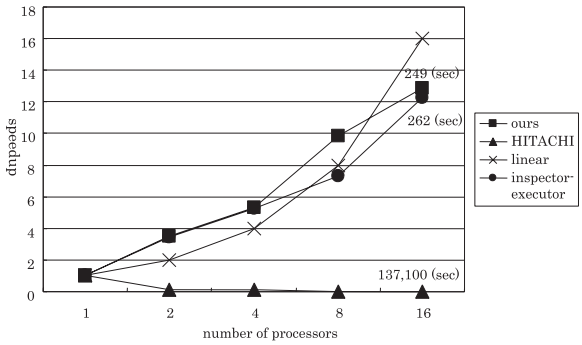

Figure 7. pde1,N=7

5.2 Compilation time

Our compiler takes longer compilation time than traditional compilers. For example, our compiler takes 207 seconds to compile the pde1 benchmark for 16 node while the original inspector-executor method takes 85 seconds. The difference is 122 ($= 207 - 85$) seconds, which is bigger than the improvement of the execution time ($262 - 249 = 13$ seconds). To pay this extra compilation time off, we have to modify the benchmark program so that it iterates the outer loop more than 9,400 times instead of the original 1,000 times. However, we do not think that this is a serious problem since the applications of our compiler is simulation programs, which iterate the outer loop a huge number of times.

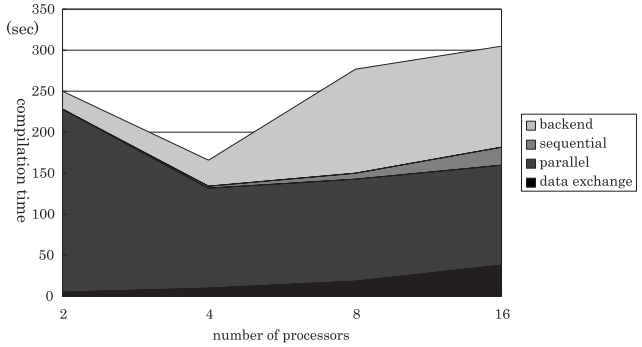

Figure 8. FT-classA compilation time

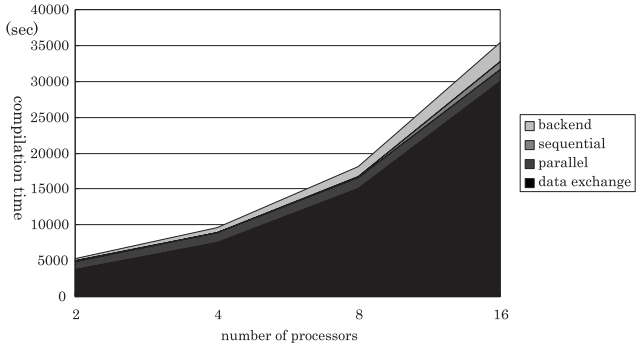

Figure 9. BT-classA compilation time

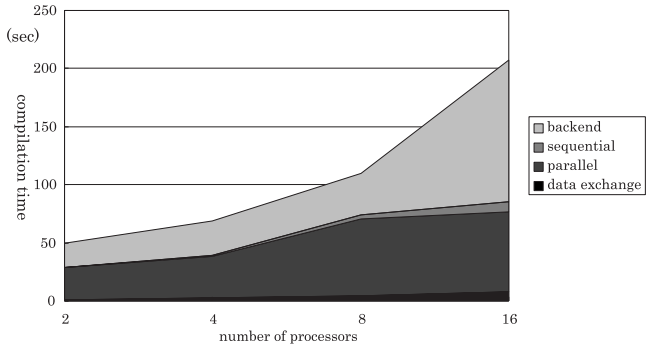

Figure 10. pde1,N=7 compilation time

Figure 8, 9, and 10 show the break down of the compilation time by our compiler. To compile the benchmark programs, we used a PC-cluster computer with the same number of node PCs as the number of node processors of the target machine. If we compile for 4 node processors, we used 4 node PCs for compilation. In the figures, backend represents the time consumed by the backend compiler, sequential represents the time by the master node PC, parallel represents the time by all the node PCs running, and data exchange represents the time for exchanging data among node PCs.

These figures illustrate that the backend compiler took proportional time to the number of node processors since the generated SPMD code for a larger number of node processors tends to be longer. The time consumed by the back-

end compiler could be reduced if our compiler can optimally merges programs into a SPMD code at the last stage.

The compilation time of BT was unacceptably long. Most of the time was spent to exchange data among node PCs. Since accesses to the distributed array spread over the program, the size of the exchanged data was huge.

6 Related Work

Although there have been a number of research activities on optimization techniques for the inspector-executor method, few activities have been dealing with a technique enabling constant folding like our proposal. Most of the activities have been dealing with runtime optimization that does not need to generate specialized code. Ponnusamy et al studied a scheduling algorithm of data transfer and so on [5]. Wu et al proposed a technique of packing multiple packets into a single one to reduce overheads due to packet handling [7]. Viswanathan et al studied a distributed memory system performing optimization with runtime information [6]. Das et al developed a middleware system using the inspector-executor method for transferring data among node processors [1]. Ding et al studied the use of runtime information for optimizing cache consistency maintenance[2].

As for research activities on code specialization, few activities have been dealing with parallel computing. Philippsen et al [4] studies the use of compile-time and runtime information for optimally allocating objects on a node processor so that inter-node data exchange is minimized. If the best allocation is not statically determined, objects are dynamically allocated with runtime information.

7 Conclusion

This paper presented our Fortran compiler providing a subset of HPF directives. It uses our extended version of inspector-executor method for implementing accesses to a distributed array specified with HPF directives. A unique feature of our compiler is that the inspector runs during compilation time. The compiler exploits the data dependency information reported by the inspector and it generates the executable code including only the executor. This allows the compiler to generate executable code statically specialized for the reported data dependency. For this aggressive optimization, our compiler runs in parallel on a PC-cluster computer. This paper presented effects of this optimization with results of experiments. Although the experiments showed the execution performance was significantly improved, the compilation took long time. Furthermore, our compiler can compile only a program in which data dependency does not change for every iteration of the outer loop since the executable binary does not include the inspector. However, these problems are not serious in the application domain of our compiler, which is simulation for natural science.

References

[1] R. Das, M. Uysal, J. Saltz and Y. S. Hwang, Communication optimizations for irregular scientific computations on distributed memory architectures, *Technical Report CS-TR-3163, University of Maryland*, Oct., 1993.

[2] C. Ding and K. Kennedy, Improving cache performance in dynamic applications through data and computation reorganization at run time, *In Program. Language Design and Imple.*, 1999, 229-241.

[3] C. Koelbel and P. Mehrotra, Compiling Global Name-Space Parallel Loops for Distributed Execution, *IEEE Trans. on parallel and distr. systems*, 2(4), 1991, 440-451.

[4] M. Philippsen and B. Haumacher, Locality optimization in JavaParty by means of static type analysis, *In Proc. Workshop on Java for High Performance Network Computing at EuroPar '98, Southhampton*, Sep., 1998.

[5] R. Ponnusamy, J. Saltz, A. Choudary, Y. S Hwang and G. Fox, Runtime Support and Compilation Methods for User-Specified Irregular Data Distributions, *IEEE Trans. on parallel and distr. Systems*, 6(8), 1995, 815-831.

[6] G. Viswanathan and J. R. Larus, Compiler-directed shared-memory communication for iterative parallel computations, *In Proceedings of Supercomputing '96, Pittsburgh, PA*, Nov., 1996.

[7] J. Wu, R. Das, J. Saltz, H. Berryman and S. Hiranandani, Distributed memory compiler design for sparse problems, *IEEE Trans. on computers*, 44(6), 1995, 737-753.

[8] D. Yokota, S. Chiba and K. Itano, A Parallelizing Compiler Optimizing for Communication Devices, *Infomation processing society of Japan journal*, 42(4), 2001, 860-867.

[9] *High Performance Fortran Forum*
http://www.crpc.rice.edu/HPFF/home.html

[10] *Nas Parallel Benchmarks*
http://www.nas.nasa.gov/Software/NPB/

[11] J. Klose and M. Lemke, *GENESIS Distributed Memory Benchmarks*
http://www.pallas.de/

[12] *Portland Group*
ftp://ftp.pgroup.com/pub/HPF/examples

[13] *Center for Computational Physics at Unversity of Tsukuba*
http://www.rccp.tsukuba.ac.jp