JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 17, 825-839 (2001)

# Short Paper

# Design and Implementation of a Communication-Efficient Data-Parallel Program Compiling System

KUEI-PING SHIH, CHING-YING LAI*, JANG-PING SHEU* AND YU-CHEE TSENG+
*Department of Computer Science and Information Engineering*
*Tamkang University*
*Tamshui, Taipei, 251 Taiwan*
*E-mail: kpshih@mail.tku.edu.tw*
**Department of Computer Science and Information Engineering*
*National Central University*
*Chungli, Taoyuan, 320 Taiwan*
*E-mail: sheujp@csie.ncu.edu.tw*
+*Department of Computer Science and Information Engineering*
*National Chiao Tung University*
*Hsinchu, 300 Taiwan*
*E-mail: yctseng@csie.nctu.edu.tw*

In this paper, we present the design and implementation of a data-parallel compiling system. The system has been implemented on a DEC Alpha 3000 workstation and incorporated into a parallel programming environment called UPPER (User-interactive Parallel Programming EnviRonment). Given an HPF program, the built-in compiler system can automatically analyze the access pattern of the HPF program, enumerate the computation and communication sets, and then generate the SPMD code for execution on nCUBE/2. Moreover, the user interface is designed to help the programmer to capture some information during the compilation and execution phases, including interprocessor communication, distribution of data elements onto processors, and execution results.

***Keywords:*** communication sets, distributed-memory multicomputers, high performance Fortran (HPF), parallelizing compilers, single program multiple data (SPMD)

## 1. INTRODUCTION

Programming on parallel computers, especially on distributed-memory multicomputers, is known to be very difficult and painful. It is time-consuming and error-prone to rewrite a sequential program to obtain a parallel program. However, investments have been made in software for sequential machines throughout the world. It is impossible to rewrite them all. Therefore, intelligent tools are needed to automatically transform sequential codes into equivalent concurrent codes that can be executed efficiently on parallel computers. Recently, several parallel programming environments designed to help

Received September 28, 1999; revised June 29, 2000; accepted August 29, 2000.
Communicated by Chu-Sing Yang.

programmers develop parallel programs have been developed.

The PARADIGM (PARAllelizing compiler for Distributed-memory General-purpose Multicomputers) compiler [1] is a source-to-source parallelizing compiler, based upon Parafrase-2 [2]. It strives to provide a fully automated way to parallelize programs for efficient execution on a wide range of distributed-memory multicomputers. PARADIGM currently accepts either a sequential Fortran 77 or High Performance Fortran (HPF) program and produces an optimized message-passing parallel program (in Fortran 77 with calls to the selected communication library and the PARADIGM runtime system).

The SUIF (Stanford University Intermediate Format) compiler [3], is a flexible infrastructure designed to support collaborative research in optimizing and parallelizing compilers. It takes a sequential FORTRAN-77 or C source program as input and translates it into a language-independent abstract syntax tree annotated with some necessary information for parallelization. It also supports automatic data partitioning for multiprocessor systems.

The Vienna Fortran Compilation System (VFCS) [4] was developed at the University of Vienna. VFCS performs a source-to-source translation from Vienna Fortran or HPF to explicitly parallel Message Passing Fortran, providing automatic parallelization as well as vectorization.

UPPER (User-interactive Parallel Programming EnviRonment) is an ongoing project at the National Central University [5]. UPPER can support either a sequential Fortran or an HPF program as input and generate SPMD code either for simulators or real machine (currently, the platform is nCUBE/2 [6]), up to users' choice. In addition to the compilation system, we also provide a friendly user interface, through which users can interact with the system. The interface provides not only an on-line editor and on-line help, but also a graphical demonstration subsystem for users to view the intermediate results produced during the compilation and simulation stages. This greatly helps users or programmers design and write parallel programs based on a variety of assertions and information generated by the compiler.

The rest of this paper is organized as follows. Section 2 reviews UPPER and gives an overview of the data-parallel compiling system proposed in this paper. In Section 3, we describe in detail the implementation of our data-parallel compiling system, including data structures, compilation techniques, and the user interface. Finally, conclusions are given and future works described in Section 4.

## 2. SYSTEM OVERVIEW

The major components of UPPER include the *user interface*, the *parallelizing compiler system* (consisting of a machine-independent phase and a machine-dependent phase), and *simulators* of several target machines. Recently, we have built a subsystem to extend the capability of UPPER to accept data-parallel programs as inputs. The main configuration of the system is shown in Fig. 1, where the modules in gray are the to-be-presented subsystem supporting data-parallel programs.

In the following, we first review UPPER and then give an overview of the data-parallel compiling system that we have newly added into UPPER.

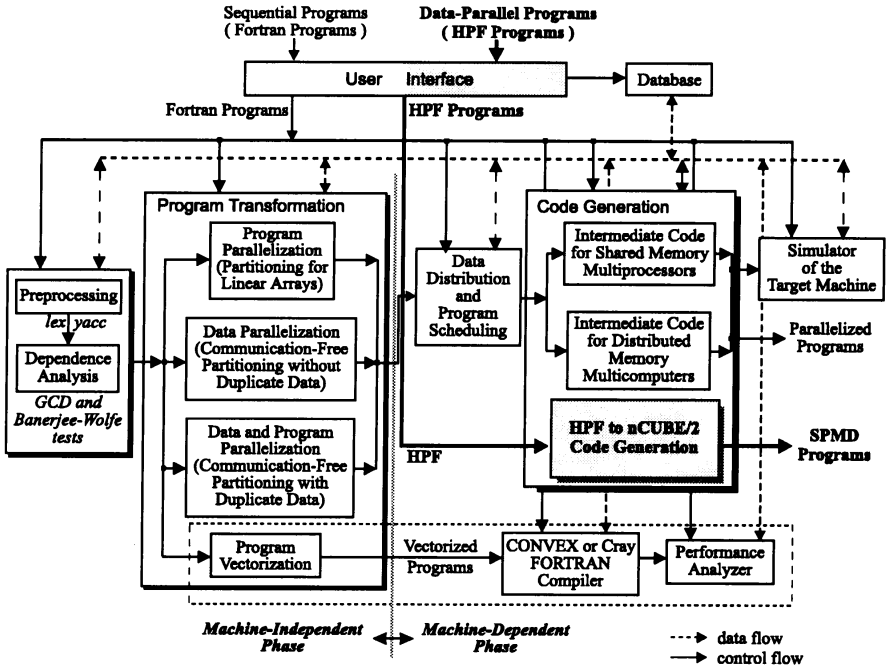


Fig. 1. The system structure of UPPER.

### 2.1 Structure of UPPER

In this subsection, we describe the original UPPER without extensions. The user interface is designed to facilitate user interaction with the system. Through the interface, users can edit their sequential or parallel programs, observe the data dependence information, and tune or restructure their programs into better forms to obtain more parallelism. Users can also set different system parameters to predict how their programs will perform on experimental machines or machines designed based on different technologies.

There are three modules in the machine-independent phase: *preprocessing*, *dependence analysis*, and *program transformation*. These modules are designed to exploit the parallelism of the given sequential program, regardless of the machine topologies and properties. The machine-dependent phase also has three modules: *data distribution*, *program scheduling*, and *code generation*. These modules accept the information produced by the machine-independent phase and generate parallel execution codes according to the machine topology, network size, and architecture. In addition, a *database* module (at the top of Fig. 1) stores all data and information generated or needed by each module.

The *preprocessing* module takes the user program and parameters of the target machine as inputs, scans and parses the program, and constructs the program representation and data-flow information for later use. Then, data dependence analysis is performed on the program representation. Several data dependence testing methods, including the GCD test [7] and Banerjee-Wolfe test [8], have been implemented in this system. The data dependence information is used to guide subsequent compilation analysis and optimiza-

tion, such as exploitation of parallelism and reusability of registers. The *program transformation* module utilizes the results of dependence analysis to improve program performance and transforms the sequential program, by using data dependence information, into a parallelizing or vectorizing form. Finally, the transformed program is mapped and scheduled for execution onto the given target machine by the *data distribution* and *program scheduling* modules.

The intermediate code for the simulator is then generated for the target machine in the *code generation* module. The simulator is a testbed for the development of this environment and a tool for evaluating parallelizing techniques. It simulates execution of the obtained parallel codes on target machine and evaluates the efficiency. The simulation outputs include the behavior records of each processor and statistical data. Based on the simulation results, users can predict whether the parallelized codes achieve the desired performance or not.

### 2.2 The Data-Parallel Program Compilation Subsystem

The subsystem is a source-to-source translator, which converts an HPF program into an equivalent SPMD program that can run on a distributed-memory multicomputer. It was implemented in the C language on a DEC Alpha 3000 workstation. We were targeting the nCUBE/2 parallel machine [6] with up to 16 nodes.

The structure of the subsystem is shown in Fig. 2. It performs a source-to-source translation from HPF programs to SPMD codes. Given an HPF program, the compilation module analyzes the array access pattern, extracts the regular behavior of the access patterns, and then creates a *class table* to record these patterns. For a program with complex data-processor mapping, the compilation module will additionally create a *compression table* to generate the compressed local array [9]. These two tables will be used frequently during the process of generating communication sets and local memory access sequences. The data structure and algorithms used to construct these tables will be presented in the next section. The methodology used the compilation module is illustrated in Fig. 3.

The user interface can ease the task of writing a data-parallel program. Information provided by the interface includes the intermediate results of compilation, interprocessor communication, distribution of data elements onto processors, and execution results. The details of user interface will be given in the next section.

## 3. IMPLEMENTATION ISSUES

In this section, we describe the implementation of the proposed data-parallel compiling system. The implementation has two major parts: the *compiler system* and the *user interface*.

### 3.1 Compiler System

HPF provides several useful directives, such as *PROCESSORS*, *TEMPLATE*, *ALIGN*, and *DISTRIBUTE*, for users to specify the mappings of data onto processors [10]. The compiler can analyze these directives and the array accessing patterns so that the

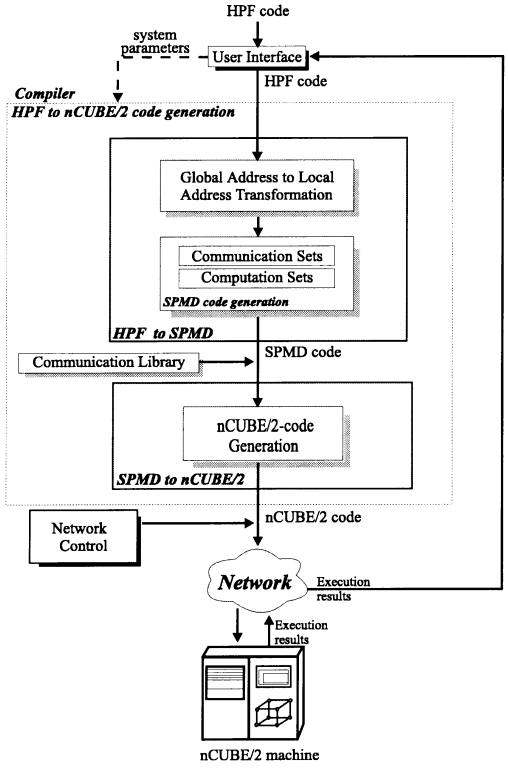


Fig. 2. Structure of the proposed data-parallel compiling system.

global references of array elements can be translated into local addresses on processors, and so that communication sets can be generated, if needed, so that processors to access non-local data. According to this information, the compiler can then generate an SPMD code for execution.

HPF supports a *two-level mapping* model which users can employ to specify data distribution. A two-level mapping is used to conceptually divide the data distribution into two steps: *aligning* data with a template and *distributing* the template onto the abstract processors. A *one-level mapping* is a degenerated case of a *two-level mapping* if the array elements are identically aligned on a template.

In Section 3.1.1, we will present how to represent an HPF program. Then, Section 3.1.2 will describe the kernel of the compiling system.

### 3.1.1 Preprocessing

Given an HPF program, the compiler first scans and parses the program to generate an *internal representation*, or *program representation*, for that program. The internal representation should not only preserve the semantic meanings of the source program, but also extract necessary information for subsequent processing.

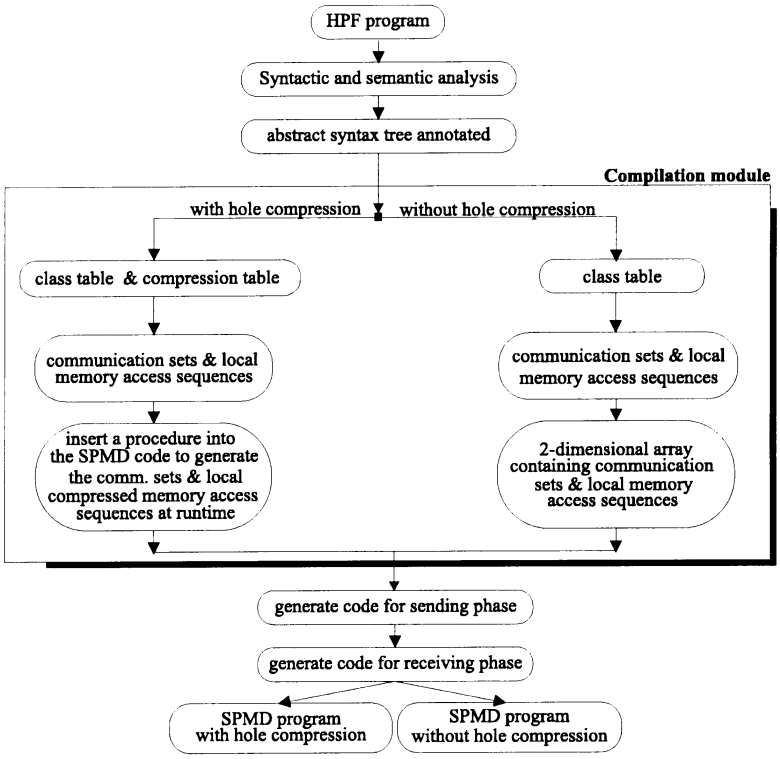


Fig. 3. Compilation phases in the compilation module.

The internal representation is shown in Fig. 4. A header, "Program", leads the representation. Following the header is a number of "Statement" nodes. Each "Statement" node can be either an "Assignment Statement," a "DO Statement," or an "IF Statement." The "Assignment Statement" node contains two children: "Left" and "Right," which respectively indicate the left- and right-hand side variables. The "Left" node in turn has two children: "Id" and "Subscript," which contain the variable's name and an expression to represent the subscript expression, respectively. The "Subscript" node can also represent multidimensional array structures. The "Right" node can have a few "Operand" nodes, each of which has an "Id" and a "Subscript," which indicate the operand's name and its subscript expression, respectively. If the operand is a constant, the 'Id' node is represented as '—'.

A "DO Statement" has three children: "Id," "Range," and "Body." The "Id" node carries the induction variable's name. The "Range" node has three children: "Lower Bound," "Upper Bound," and "Stride," which indicate how "Id" changes in value. The "Body" node points to a list of "Statement" nodes, each of which can be any one of the three statements.

As for the "If Statement" node, it contains two children: "True" and "False," each of which can have a list of "Statement" nodes as its children.

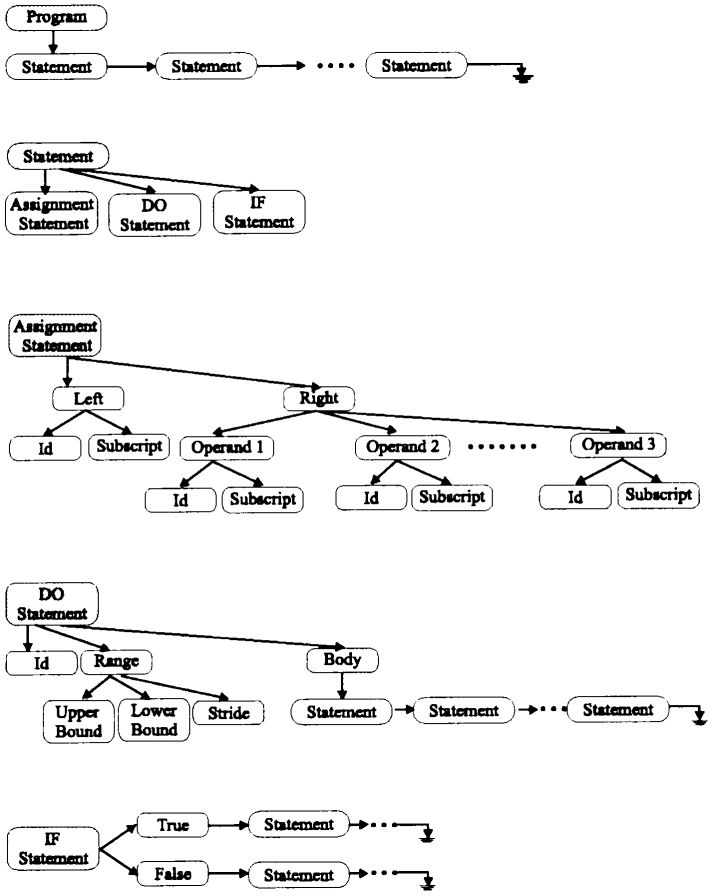


Fig. 4. The program representation design in the compiler system.

The above description does not contain the declaration part. We store such information separately for subsequent use. The data structure is designed for efficient access by compiler writers. The representation can be easily extended to multidimensional representation. The general representation is illustrated in Fig. 5.

### 3.1.2 Kernel design

Presently, the compiling system supports both the one-level and two-level mapping compilation techniques. The benefit is that we can process HPF programs in which array elements are block-cyclically distributed onto processors in either one-level or two-level mapping. However, the array statements that are allowed must be limited to *doall* array statements, the array dimension must be one-dimensional, and the subscript must contain only one induction variable. The extension to multi-dimensional structures is currently under investigation. We also plan to include the affined subscripts in the near future.

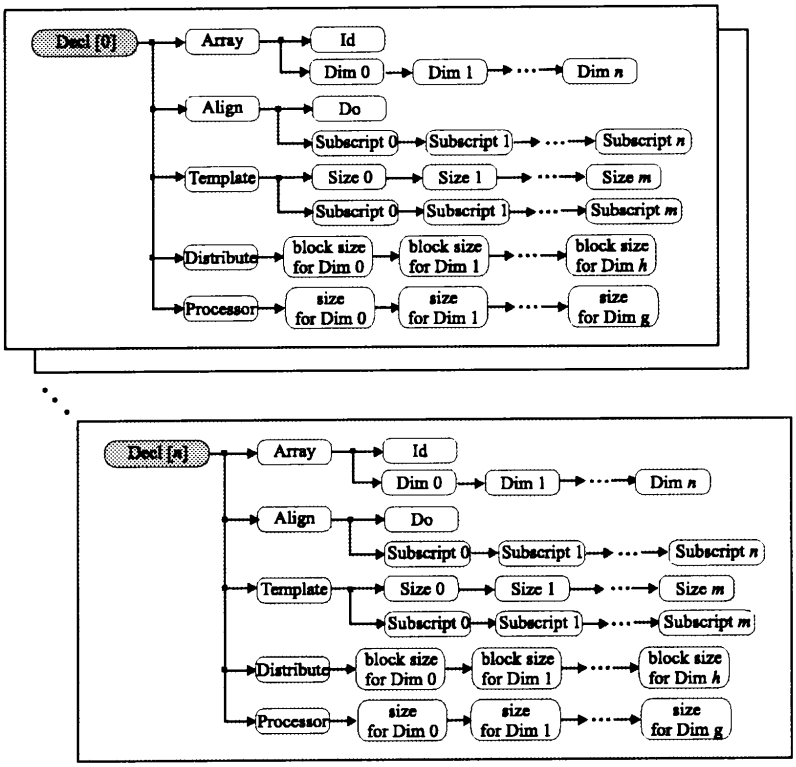


Fig. 5. General structure for representing the declaration area of an HPF program.

In the implementation, we apply some one-level mapping techniques proposed in [11], where efficient methods are presented to enumerate the local memory access sequence and to generate communication sets. Given a one-level mapping, the array reference patterns are generated, from which a *class table* to record the classification of blocks is created. As shown in [11], each array statement can be regularly partitioned into several repeating sets of classes. According to the class table, generation of communication sets and local memory access sequences can be efficiently achieved. Generating communication sets and local memory access sequences will cause a trade-off between execution time and memory usage. Hence, if the communication sets are small, communication set generation can be pre-computed at compile-time to save some run-time execution.

One interesting problem in two-level mapping is the *hole compression* problem [9]. It is known that aligning related data objects together and distributing them onto the same processor can reduce interprocessor communication. However, non-unit alignment stride will result in a lot of "memory holes."

Memory holes not only waste memory, but also degrade system performance because the memory spatial locality may be reduced. These holes should be removed. At the same time, the computation statements may also need to be changed, as the array elements are now located at difference places after compression. The hole compression technique used in this implementation is based on the one proposed in [9]. First, the

blocks are classified into classes. Then, a *class table* to record the classification of blocks is generated. From the class table, a *compression table* is generated. Since data distribution on a processor typically has some repetitive patterns, simply recording the attributes of the first repetitive pattern will be sufficient. Using the compression table, we can easily obtain the compressed local array. It is worth mentioning that hole compression only affects *where* the array elements are placed in a processor's local memory, not *which* array elements are assigned to which processors. Therefore, we can transform an array statement in a two-level mapping into an equivalent array statement in a one-level mapping. The techniques used to generate communication sets and local memory access sequences for array statements in one-level mappings can, thus, be applied to.

Although we can deal with one-level and two-level mappings in a consistent manner, it is still necessary to transform the communication sets and local memory access sequences into ones in the compressed local arrays. The transformations need not be performed on all array elements – only the first array element on every block needs to be recomputed. Doing so can save a lot of computation.

The user interface plays as an important role in a compiler system; without it, writing a parallel program would be a pretty difficult job. We have designed and implemented a user interface for our compiling system. Through the interface, a user can write, edit, and compile his/her programs, select and use compilation techniques, simulate and run the generated SPMD program, and view the results. In addition, performance statistics, intermediate compilation information, resultant SPMD codes, memory usage before and after hole compression, communication sets among processors, etc., are also provided. Wherever appropriate, graphs and tables are used to increase visual readability. These will greatly help users analyze their program and, thus, decide how to improve it by either adjusting the program parameters or tuning the system parameters.

Fig. 6 shows the first screen that is seen upon entering UPPER. The upper-left corner shows the source program (Fortran or HPF). After compilation, the parallelized or SPMD codes are listed in the upper-right corner. The execution results are shown in the lower half. There are 7 pull-down menus on the top bar: **File, View, Compile, Execution, Simulate, Options,** and **Help**, which are explained in the following.

In the **File** menu, three commands are supported: *Open*, *Editor*, and *Exit*. By means of the *Open* command, users can open an existing source program to compile. The *Editor* command will invoke an on-line editor. All of the jobs will stop and the system will halt if the *Exit* command is selected.

When a source program is opened, the **Options** menu can be opened to set system parameters. Two submenus are provided: *Set Machine Environment* and *Compilation Techniques*. In the *Set Machine Environment* submenu, users can set the execution environment, such as the machine type (distributed-memory multiprocessor, shared-memory multiprocessor, or supercomputer), the number of processors, and the network topology. These parameters will also be used by the simulator to simulate the execution of the generated parallelized code in the specified environment. For the case presented in this paper, the target machine will be an nCUBE/2 with up to 16 processors. From the submenu *Compilation Techniques*, users can choose one compilation technique to compile their program.

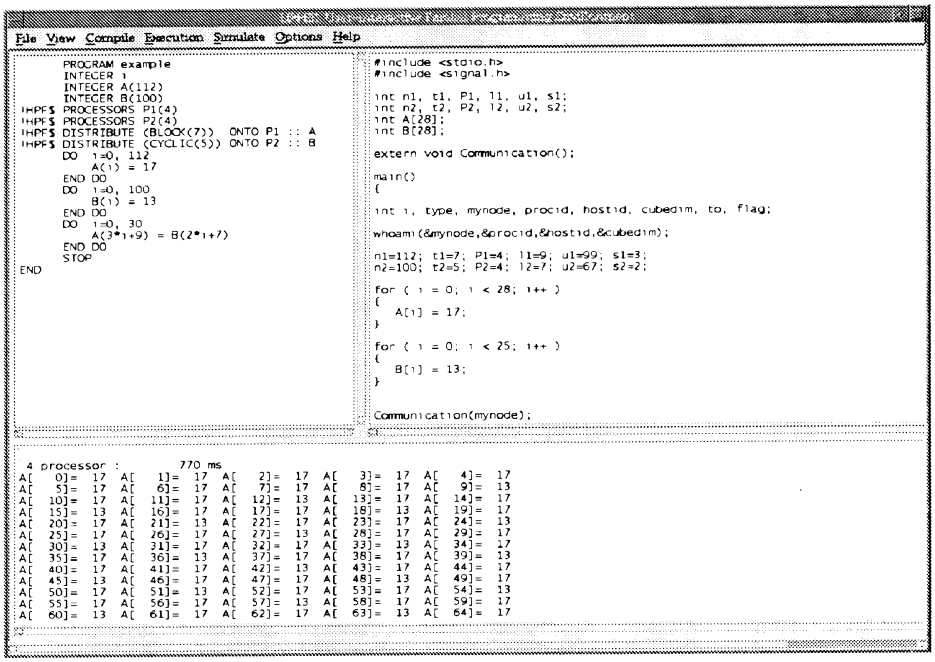


Fig. 6. A snapshot of UPPER's user interface. The upper-left part is the original HPF program, the upper-right part is the SPMD code generated by the compiling system, and the lower part contains the execution results.

Since UPPER is designed for academic research, several compilation techniques have been developed for experimental purposes. Currently, the choices include: communication-free partitioning with/without duplicate data [12], non-communication-free transformation [13], vectorization transformation [14], compilation for one-level mappings [11], and compilation for two-level mappings [9]. The first three items are for compiling sequential Fortran programs into parallelized or vectorized ones, and the last two are for compiling HPF programs into SPMD codes.

After choosing a suitable compilation technique, a user can use **Compile** on the menu bar to compile his/her program. The produced program, in parallelized, vectorized, or SPMD form, will be shown on the screen. The user can compare the source program with the produced codes and, to some extent, understand how the parallel compiler transforms the source program.

When the communication-free partitioning with/without duplicate data or non-communication-free transformation option is chosen, the resultant code will be intermediate codes for the simulator. Then the **Simulate** menu can be selected. Some simulation results will be generated and illustrated by the demonstration system. If the vectorization transformation option is chosen, **Run** under the **Execution** menu can be chosen to submit the vectorized codes to a Convex C3840 vector computer. On the other hand, the user should choose compilation for one- or two-level mappings if the input is an HPF program; the output will be some SPMD codes, which can be submitted to an

nCUBE/2 if **Run** command is selected. We support two execution models for users to run programs: *Run* and *Speedup Evaluation*. The former is used to run the SPMD program on nCUBE/2 by means of a user-specified number of processors, and the execution results will be shown in the lower half of the screen, below the source and the generated SPMD programs, such as in the lower half of Fig. 6. The latter, *Speedup Evaluation*, is for experiments on network size; once chosen, the SPMD codes will be run on nCUBE/2 several times using 1, 2, 4, 8, and 16 processors. The performance statistics will be fed back to the system for analyses. The user can easily grasp how his/her program performs under different conditions.

One feature new to UPPER is the **View** menu, which provides three new functions for visualizing our system: *View Communication*, *View Compression*, and *View Performance*. The former two functions are available after executing the **Compile** command, while the last is available after executing the *Speedup Evaluation* command. *View Communication* is used to demonstrate communication set generation, which is a key part of compiling an HPF program. Users can, thus, observe the communication pattern and the amount of communication among processors. An illustrative example of *View Communication* is shown in Fig. 7. In Fig. 7, the left half shows the communication pattern. Nodes in the left column are senders, and those in the right column receivers. The connection lines indicate necessary of interprocessor communication between the two nodes. The right half shows the communication sets. The upper-right part is for senders, and the lower-right part for receivers.

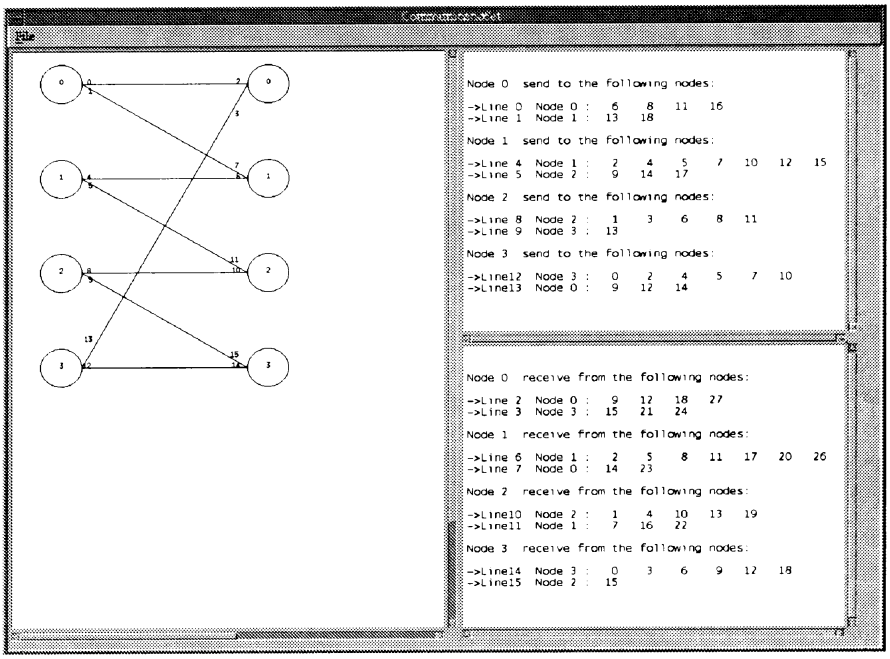


Fig. 7. A snapshot of *View Communication*, which illustrates the communication pattern and the amount of communication among processors.

The *View Compression* command can visually demonstrate the data distribution on processors with and without hole compression. By means of *View Compression*, users can easily visualize the memory usage. Fig. 8 shows a snapshot of *View Compression*. The upper half is the data layout on processors *with* hole compression, while the lower half is that without hole compression.

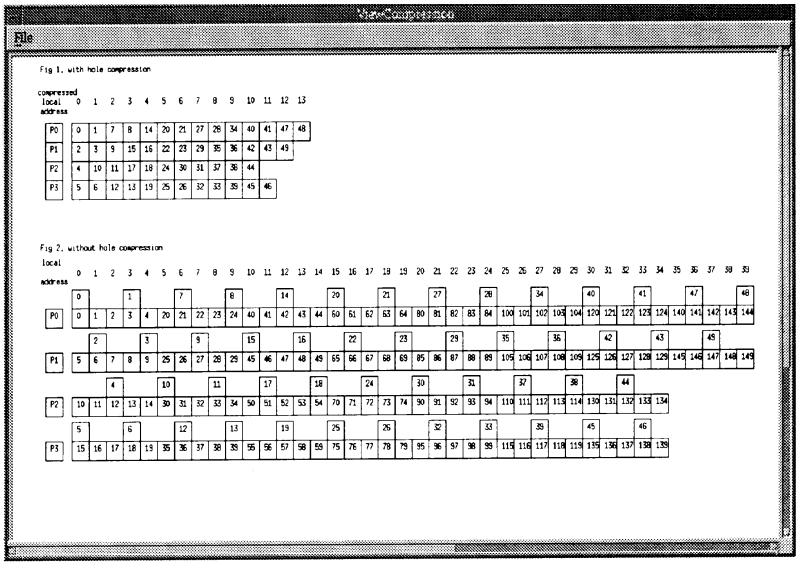


Fig. 8. A snapshot of the *View Compression* screen, which illustrates the data distribution on processors with and without hole compression.

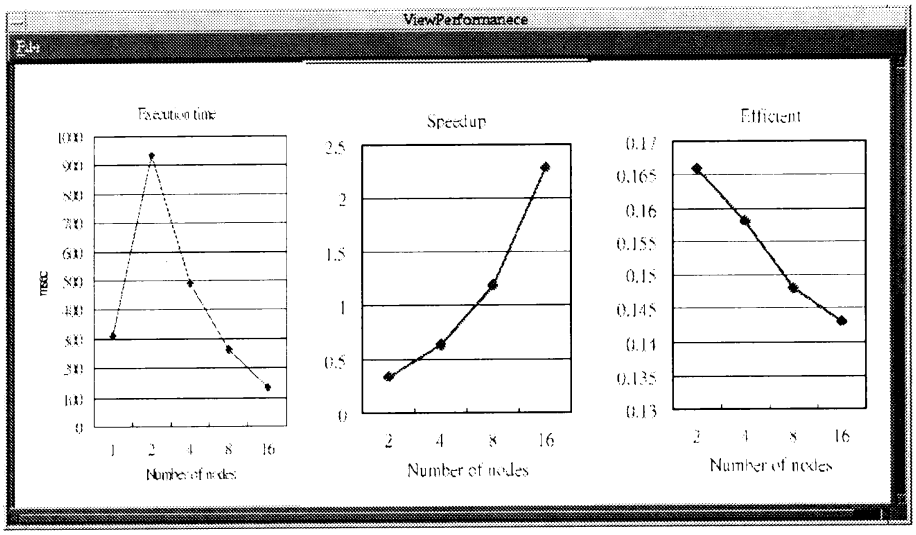


Fig. 9. A snapshot of the *View Performance* screen, which compares the 1-processor execution time, speedup, and efficiency, against the 2-, 4-, 8-, and 16-processor cases.

The *View Performance* command provides a visual comparison of the performance of the program when executed on a 1-, 2-, 4-, 8-, and 16-processor nCUBE/2. This enables comparison of the execution time, speedup, and efficiency, against the 1-processor case. *Speedup* is defined as $T_1/T_p$, where $T_1$ is the execution time with one processor and $T_p$ is the execution time with $p$ processors. *Efficiency* is defined as *speedup*/$p$. A snapshot of *View Performance* is illustrated in Fig. 9.

Finally, the **Help** menu provides on-line guidance regarding how to correctly use the UPPER environment.

## 4. CONCLUSIONS

In this paper, we have described the design and implementation of a communication-efficient data-parallel programs compiling system. The compiler has been incorporated into the UPPER environment, a multi-platform compiling environment that we have developed at National Central University. Given an HPF program, the compiler system can efficiently enumerate the computation and communication sets. Both one-level and two-level mappings can be effectively processed by the system. For the case of two-level mapping, the compiling system can even perform hole compression by eliminating the memory holes, which are caused by non-unit alignment stride in the program. Hole compression can save memory space, increase spatial locality, and further improve system performance.

The user interface of UPPER is designed to help users quickly obtain experimental results. Given an HPF program, an SPMD code can be generated for execution on nCUBE/2. Important information, such as intermediate compilation results, memory usage, hole compression results, and communication sets among processors, can be collected and shown to users to give them a better understanding of the internal parallelism in their programs. We expect that this environment can greatly reduce the work involved in writing parallel programs.

## ACKNOWLEDGEMENTS

This work was supported by the National Science Council of the Republic of China under grants NSC 89-2213-E-008-013, NSC 89-2213-E-008-023, NSC 89-2213-E-008-024, and NSC 89-2213-E-008-025.

## REFERENCES

1. M. Gupta and P. Banerjee, "PARADIGM: A compiler for automatic data distribution on multicomputers," in *Proceedings of the ACM International Conference on Supercomputing*, 1993, pp. 87-96.
2. C. D. Polychronopoulos, M. Girkar, M. R. Haghighat, C. L. Lee, B. Leng, and D. Schouten, "Parafrase-2 : an environment for parallelizing, partitioning, synchronizing and scheduling programs on multiprocessors," in *Proceedings of International Conference on Parallel Processing*, Vol. 2, 1989, pp. 39-48.

3. M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S. W. Liao, E. Bugnion, and M. S. Lam, "Maximizing multiprocessor performance with the SUIF compiler," *IEEE Computer*, 1996, pp. 84-89.
4. B. M. Chapman, P. Mehrotra, and H. P. Zima, "Programming in Vienna Fortran," *Scientific Programming*, Vol. 1, No. 1, 1992, pp. 31-50.
5. T. S. Chen, K. P. Shih, and J. P. Sheu, "Design and implementation of a user-interactive parallel programming environment," in *Proceedings of the National Science Council*, Republic of China, Part A: Physical Science and Engineering, Vol. 20, No. 4, 1996, pp. 474-490.
6. *nCUBE/2 Supercomputers Manual*, NCUBE Company, 1990, Gates94.
7. H. Zima and B. Chapman, *Supercompilers for Parallel and Vector Computers*, New York: ACM Press, 1991.
8. M. Wolfe, *High Performance Compilers for Parallel Computing*, Redwood City, CA: Addison-Wesley, 1996.
9. K. P. Shih, J. P. Sheu, C. H. Huang, and C. Y. Chang, "Efficient index generation for compiling two-level data-processor mappings in data-parallel programs," *Journal of Parallel and Distributed Computing*, Vol. 60, No. 2, 2000, pp. 189-216.
10. C. Koelbel, D. Loveman, R. Schreiber, G. Steele Jr., and M. Zosel, *The High Performance Fortran Handbook*, The MIT Press, 1994.
11. W. H. Wei, K. P. Shih, and J. P. Sheu, "Compiling array references with affined functions for data-parallel programs," *Journal of Information Science and Engineering*, Vol. 14, No. 4, 1998, pp. 695-723.
12. T. S. Chen and J. P. Sheu, "Communication-free data allocation techniques for parallelizing compilers on multicomputers," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 5, No. 9, 1994, pp. 924-938.
13. J. P. Sheu and T. S. Chen, "Partitioning and mapping of nested loops for linear array multicomputers," *The Journal of Supercomputing*, Vol. 9, No. 1/2, 1995, pp. 183-202.
14. C. Y. Chang, "Design and implementation of an assistant tool for vector compilers," Department of Computer Science and Information Engineering, National Central University, Taiwan, 1995.

**Kuei-Ping Shih (石貴平)** received the B.S. degree in Mathematics from Fu-Jen Catholic University, Taiwan, in June 1991 and the Ph.D. degree in Computer Science and Information Engineering from National Central University, Taiwan, in June 1998. After two years of military obligation, he joined the faculty of the Department of Computer Science and Information Engineering, Tamkang University, Taiwan, Republic of China, as an assistant professor in 2000. His research interests include parallelizing compilers, interconnection network, parallel and distributed computing, and wireless network protocol design.

**Ching-Ying Lai (賴靜瑛)** was born in Taiwan, Republic of China, on February 26, 1968. She received B.S. degree in applied mathematics from the Providence University, Taiwan, in 1992 and the M.S. degree in Computer Science and Information Engineering from the National Central University, Taiwan, in 1998. She has worked at the Com-

puter & Communications Research Laboratories, Industrial Technology Research Institute (CCL/ITRI) from 1998 so far. Her research interests include multimedia networking and mobile multimedia computing.

**Jang-Ping Sheu (許健平)** received the B.S. degree in Computer Science from Tamkang University, Taiwan, Republic of China, in 1981, and the M.S. and Ph.D. degrees in Computer Science from the National Tsing Hua University, Taiwan, Republic of China, in 1983 and 1987, respectively.

He joined the faculty of the Department of Electrical Engineering, National Central University, Taiwan, Republic of China, as an associate professor in 1987. He is currently a professor of the Department of Computer Science and Information Engineering, National Central University. From July of 1999 to April of 2000, he was a visiting scholar at the Department of Electrical and Computer Engineering, University of California, Irvine. His current research interests include parallelizing compilers, interconnection networks, and mobile computing.

Dr. Sheu is a senior member of the IEEE, a member of the ACM and Phi Tau Phi Society. He is an associate editor of Journal of Information Science and Engineering, Journal of the Chinese Institute of Electrical Engineering, and Journal of the Chinese Institute of Engineers. He received the Distinguished Research Awards of the National Science Council of the Republic of China in 1993-1994, 1995-1996, and 1997-1998.

**Yu-Chee Tseng (曾煜棋)** received his B.S. and M.S. degrees in Computer Science from the National Taiwan University and the National Tsing-Hua University in 1985 and 1987, respectively. He worked for the D-LINK Inc. as an engineer in 1990. He obtained his Ph.D. in Computer and Information Science from the Ohio State University in January of 1994. From 1994 to 1996, he was an Associate Professor at the Department of Computer Science, Chung-Hua University. He joined the Department of Computer Science and Information Engineering, National Central University in 1996, and has become a professor since 1999. Since August 2000, he has become a professor at the Department of Computer Science and Information Engineering, National Chiao Tung University, Taiwan. Dr. Tseng served as a Program Committee Member in the International Conference on Parallel and Distributed Systems, 1996, the International Conference on Parallel Processing, 1998, the International Conference on Distributed Computing Systems, 2000, and the International Conference on Computer Communication and Networks 2000. He was a Workshop Co-chair of the National Computer Symposium, 1999. His research interests include wireless communication, network security, parallel and distributed computing, and computer architecture.

Dr. Tseng is a member of the IEEE Computer Science and the Association for Computing Machinery.