PPT: A PARALLEL PROGRAMMING TOOL FOR DISTRIBUTED MEMORY MULTIPROCESSORS

Yeh-Ching Chung*, Wu-Hsun Ho and Chia-Cheng Liu
Department of Information Engineering & Computer Science
Feng Chia University
Taichung, Taiwan 407, R.O.C.

Key Words: SPMD, distributed memory multiprocessors, parallel programming tools, program partitioning, scheduling.

ABSTRACT

Traditionally, to program a distributed memory multiprocessor, a programmer is responsible for partitioning an application program into modules or tasks, scheduling tasks on processors, inserting communication primitives, and generating parallel codes for each processor manually. As both the number of processors and the complexity of problems to be solved increases, programming *distributed memory multiprocessors* becomes difficult and error-prone. In a distributed memory multiprocessor, the program partitioning and scheduling play an important role in the performance of a parallel program. However, how to find the best program partitioning and scheduling so that the best performance of a parallel program on a distributed memory multiprocessor can be achieved, is not an easy task. In this paper, we present a parallel programming tool, PPT, to aid programmers to find the best program partitioning and scheduling and automatically generate the parallel code for the *single program multiple data* (SPMD) model on a distributed memory multiprocessor. An example of designing a parallel FFT program by using PPT on an NCUBE-2 is also presented.

I. INTRODUCTION

Many commercial distributed memory multiprocessors have been introduced, such as NCUBE-2[10] and the Connection Machine 5 (CM-5)[18]. In general, they provide the following execution models for programmers:

1. *Single Program Multiple Data (SPMD) Model* - The same code runs on multiple processors, with each processor working on its own portion of the data. The processors are loosely synchronized in that certain statements execute at approximately the same time on all processors.
2. *Host Model* - One or more SPMD programs are invoked, as needed, by a master program running on a host computer. This model arises frequently when "parallelizing" code originally written for a sequential computer.
3. *Heterogeneous Model* - Multiple parallel programs on multiple processors, passing data to each other in a pipeline, or in customer/sever fashion.
4. *Asynchronous Model* - Each processor or group of processors run its own program, communicating with other processors through a protocol created by the applications programmer. The

*Correspondence addressee

asynchronous model places no constraints on the interactions of the processors.

However, it is not an easy task to design a parallel program on a distributed memory multiprocessor. Jobs such as choosing the execution model, partitioning the problem into proceses, grouping these processes into tasks, assigning each task to a processor, and inserting synchronization primitives for proper execution have to be performed (manually or automatically).

In general, three approaches are used to develop a parallel program on a distributed memory multiprocessor:

1. *Manual Approach* - A programmer is responsible for performing all these jobs [15].
2. *Automatic Approach* - A restructuring compiler is responsible for extracting parallelism and restructuring sequential programs into parallel programs automatically [12].
3. *Hybrid (semi-automatic) Approach* - A hybrid approach includes partial automation.

Programmers are error-prone to handle tedious chores, like communication primitives insertion. For example, system deadlock is the most common problem, and is difficult to detect once the program has been developed. Thus the manual approach is not very useful for large applications. A recent study on the performance of automatic parallelization compilers by Cheng and Pase [3] has shown that "automatic tools produce insufficient performance improvement due to false dependencies". Many researchers have used the hybrid approach to develop parallel programs on distributed memory multiprocessors [19,20]. They show that automatic scheduling and synchronization produce better results than the manual approach.

In this paper, we present a parallel programming tool PPT, which uses a hybrid approach, for the SPMD model on a distributed memory multiprocessor. PPT can be used for distributed memory multiprocessors with different interconnection networks, such as the hypercube network, the fully-connected network, the mesh network, and the fat tree network (CM-5). It is easy to design and debug a parallel program for the SPMD model. However achieving a balanced load among processors in the SPMD model is not a trivial task since the computation load for each processor may be different (especially for irregular problems). Another important issue for programming a distributed memory multiprocessor is to minimize the communication cost among processors. The goal of PPT is to provide tools for programmers to design a parallel program that can be run on a distributed memory multiprocessor efficiently (i.e. balanced load and low communication cost).

Since program partitioning and scheduling play an important role in the performance of a parallel program on a distributed memory multiprocessor, in PPT, we provide a performance estimation scheme for programmers to evaluate their program partitioning. For the scheduling part, in the present development, we provide six scheduling algorithms for *static scheduling problems*. Each scheduling algorithm is characterized by using some parameters provided in PPT. To use PPT to design a parallel program on a target distributed memory multiprocessor, the programmer is responsible for the program partitioning. PPT will help the programmer to generate the corresponding *directed-acyclic graph (DAG)* of the partitioned program, evaluate the properties of the DAG, choose a scheduling algorithm according to the properties of the DAG, perform scheduling, insert the communication primitives, generate the parallel code for the partitioned program, and produce the performance measures for each processor (such as the execution time).

Many parallel program developing tools have been developed by researchers. A survey of related parallel program developing tools is given in Section 2. The proposed parallel programming tool is described in Section 3. An example of designing a parallel FFT program by using PPT on an NCUBE-2 parallel computer is presented in Section 4.

II. RELATED WORK

Many parallel program developing tools have been addressed in the literature. In [17], Snyder proposed a parallel programming environment tool, POKER, for distributed memory multiprocessors. POKER provides a graphical representation of communication structures.

In [2], Allen et al. presented a parallel programming assistant, PTOOL. PTOOL performs sophisticated dependency analysis, including advanced interprocedural flow analysis. It identifies parallel loops, extracts global variables, and provides a simple explanation facility. It also transforms control dependencies into data dependencies. However, it only tests loops for independences and does not provide partitioning and synchronization mechanisms for nonparallel loops.

CAMP [13] partitions both parallel and nonparallel loops, and reduces dependencies by using process alignment and minimum-distance algorithms. Since it extracts more parallelism and eliminates many dependencies, efficiency loss because of processor suspension is reduced. CAMP also inserts synchronization primitives, and estimates performance for different partitioning strategies.

In [8], Dongarra and Sorensen proposed a parallel Fortran program developing and analyzing tool, SCHEDULE, which uses centralized dynamic scheduling algorithms. This algorithm performs well for the shared memory multiprocessors but is not suitable for the distributed memory multiprocessors.

TASK GRAPHER [9] schedules task graphs onto arbitrary machines. It provides a graphical interface for users to interact with the tool. However, it does not generate parallel code.

In [14], Polychronopoulos et al. described a parallelizing compiler system, PARAFRASE-2, which performs dependence analysis, program partitioning and dynamic scheduling on shared memory machines.

Hypertool [19] is a programming aid tool for the SPMD model on message-passing systems. It performs scheduling, mapping, and communication primitives insertion automatically. It also generates performance estimates and quality measures for the parallel code. However, the generated code is for the simulator SIMON.

In [20], Yang and Gerasoulis proposed a parallel programming tool, *PYRROS*, for scheduling on distributed memory multiprocessors. *PYRROS* is designed for coarse grain programs. It performs scheduling, mapping, communication primitives insertion, and parallel code generation automatically.

The work presented in this paper is similar to [19] and [20]. PPT is designed for SPMD model which is the same as Hypertool. It performs scheduling and communication primitives insertion automatically which are also provided in [19] and [20]. It generates the parallel program automatically which is also provided in [20]. However, there are two major differences between our work and those of [19] and [20]. First, the tool of [19] is designed for the medium grain programs and the tool presented in [20] is aimed for the coarse grain programs. PPT can be used for the coarse, medium, and fine grains programs.

The second major difference between our work and [19] and [20] is the approach used for scheduling. In [19] and [20], a partitioned program (a *macro-dataflow graph*) is first scheduled on a virtual machine, which is generated by the scheduling algorithms. The processor number is not decided until the scheduling is done. Then a one-to-one mapping algorithm is used to map the nodes of the virtual machine on the processors of the target machine. In our approach, the scheduling and mapping is performed simultaneously. In stead of scheduling a partitioned program on a virtual machine, the properties of the partitioned program, such as its graph parallelism and grain size, are analyzed first. According to the properties of the partitioned program, the scheduling algorithm and the number of processors are chosen. Then the scheduling is performed by considering the number of processors used and the interconnection network of the target machine.

III. PPT

PPT takes a user partitioned program, which is represented by a DAG, and the interconnection network of the target machine as input, generates the corresponding DAG of the partitioned program, analyzes the properties of the partitioned program, selects the number of processors, performs scheduling, inserts communication primitives, generates the parallel program of the paritioned program, and produces the performance measures for each processor.

The outline of PPT is shown in Fig. 1. From Fig. 1, we can see that PPT consists of six components, an *X-window user interface*, a *DAG handler*, a *scheduler*, a *communication analyzer*, a *code generator*, and a *performance evaluator.*

The X-window user interface is responsible for interactions between programmers and other components of PPT. A programmer can ask PPT to perform the desired actions through this interface.

The DAG handler consists of two components,

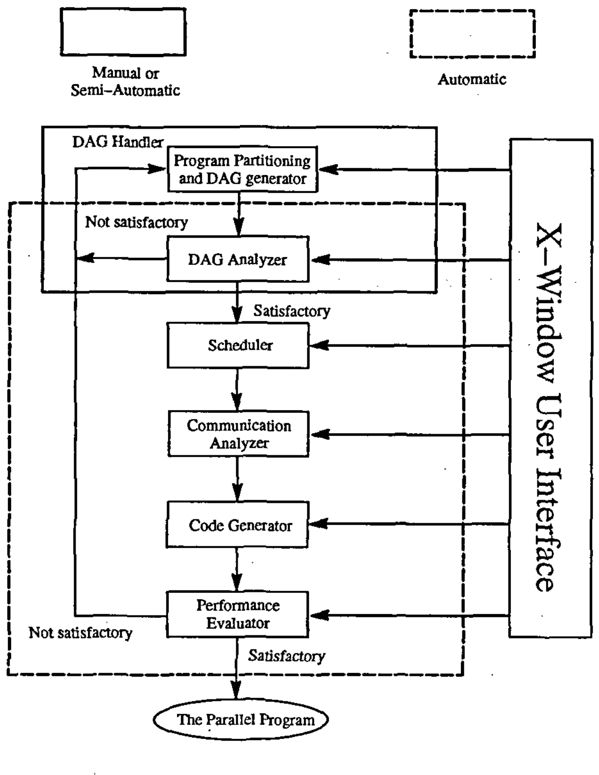

Fig. 1. The outline of PPT.

the program partitioning and DAG generator, and the DAG analyzer. In the program partitioning and the DAG generation phase, a program is partitioned into tasks and a corresponding DAG to the partitioned program is generated. PPT provides a DAG generator for the programmer to generate the DAG in an interactive fashion. The *attribute table*, which contains information about the computation cost of each task, the communication cost between two tasks, and the data which must be sent from one task to another, is also built in this stage. The DAG analyzer is responsible for the property analysis of a DAG. It outputs the *property table* of a DAG.

According to the property table, the scheduler selects a scheduling algorithm and the number of processors that will be used for executing the parallel program (the selection of a scheduling algorithm and the number of processors can be done by programmers as well). The scheduler produces a (task, processor) table, the scheduling length of each processor, and the earliest start time of each task on a processor.

The communication analyzer generates a *task block table* and a *communication table* according to the (task, processor) table and the earliest start time of each task. From the task block table and the communication table, the parallel program corresponding to the partitioned program is generated by the code generator for the target machine.

The performance evaluator is responsible for executing the parallel program on a target machine. It provides information about the execution time of each processor, the speedup of each processor, and the predicted (simulation) and real (experimental) speedups of the parallel program. The details of PPT is given in the following subsections.

1. The X-window user interface

The main menu of the X-window user interface is composed of seven selection items: "Program Partitioning & DAG Generator", "DAG Analyzer", "Scheduler", "Communication Analyzer", "Code Generator", "Performance Evaluator" and "Exit", which is shown in Fig. 2.

The "Program Partitioning & DAG Generator" selection item is for drawing the corresponding DAG of a partitioned program. When this item is selected, a DAG generator window will pop-up. The DAG generator has seven icons, which represent seven functions and is shown in Fig. 7, for programmers to draw DAGs. The first icon is used to select a node or a link. The second icon is used to generate a node of a DAG and input the corresponding data of the generated node. The third icon is used to create a link of a DAG and input the corresponding data of the created link. The fourth icon is used to modify the data of a node or a link. The fifth icon is used to delete a node or a link. The sixth icon is used to save a DAG, which is generated by the DAG generator, to a file. The file can be read by PPT as well. The seventh icon is used to quit from the DAG generator and go back to the main menu.

The "DAG Analyzer" selection item is for analyzing the properties of a DAG.

The "Scheduler" selection item is for selecting the number of processors, scheduling a DAG on processors, and adding scheduling algorithms to PPT. When this item is selected, a submenu, which consists of "Processor Numbers", "Scheduling Algorithms", and "Add Scheduling Algorithms" selection items, will pop-up. The "Processor Numbers" selection item is for determining the number of processors that will be used to execute a parallel program. When this item is selected, a submenu will pop-up (see Fig. 8). Programmers can select the number of processors they need or let PPT determine it. The "Scheduling Algorithms" selection item is for selecting a scheduling algorithm. When this item is selected, a submenu will pop-up (see Fig. 9). The selection of a scheduling algorithm can be done by a programmer or PPT. The "Add Scheduling Algorithms" selection item is for adding scheduling algorithms to PPT and enforcing the performance evaluation of scheduling algorithms.

The "Communication Analyzer" selection item is for analyzing the communication behavior of tasks among processors and creating a communication table for the code generator.

The "Code Generator" selection item is for generating the desired parallel program.

The "Performance Evaluator" selection item is for executing a prallel program on an NCUBE-2

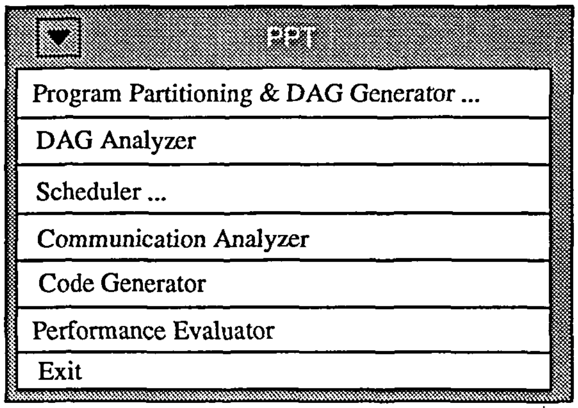

Fig. 2. The main menu of PPT.

machine and showing the execution results.

The "Exit" select item is used to exit the PPT tool.

The X-window user interface was implemented by C language and X library version 11 release 4.

2. The DAG handler

(1) The program partitioning and the DAG generator

To execute a program on a distributed memory multiprocessor, the program must be partitioned into tasks. The purpose of the program partitioning is to determine the grain size of tasks such that the best performance of the program on a distributed memory multiprocessor can be achieved. In general, the finer the grain size, the higher the parallelism. However, if a very fine grain partitioning is used, the communication overhead due to data sent from one processor to other processors may greatly increase the execution time of a program. If a coarse grain partitioning is used, a lot of parallelism available in a program may be lost. This would result in a low speedup. Therefore, it is important to balance the trade-off between parallelism and grain size so that a better partitioning can be obtained.

Since PPT is designed for SPMD model, we prefer to use the modular programming style in which a program is composed of a set of procedures called by the main program. The same programming style is also used in Hypertool. There are three advantages by using the modular programming style:

1. The program is easy to design, maintain, and debug.
2. The program partitioning is relatively easy to perform.
3. The DAG can easily be generated from the partitioned program (manually or automatically).

The programmer is required to partition a program into tasks. PPT provides a DAG generator for a programmer to generate the corresponding DAG of the partitioned program in an interactive fashion. The programmer is also responsible for the attribute table generation. The attribute table stores the information about the corresponding procedure that a task is associated with, the computation cost of each task, the communication cost between tasks, and the data that must be sent to other processors.

(2) The DAG analyzer

The DAG analyzer is responsible for the property analysis of a DAG. The properties of a DAG, such as the *graph parallelism* (the ratio of the total computation time of a DAG to the total computation time of tasks on the critical path of a DAG) [16] and the ratio of the average communication cost to the average computational cost (CCR) [4], have great influence on the scheduling length of a scheduling algorithm and the number of processors used for execution. For example, if the graph parallelism is equal to 4 and the CCR is less than 1, using the *highest level first with estimated time* (HLFET) scheduling algorithm [1] with 4 processors may produce a better scheduling length than using the HLFET scheduling algorithm with 8 processors. Therefore, it is important to study the relationship between the scheduling algorithms and the properties of DAGs and embed these properties in PPT.

In [5], we have performed extensive simulation to study the relationship between the list scheduling algorithms and the properties of DAGs. The simulation results show that the graph parallelism and CCR of a DAG are the most important properties that have a great influence on the scheduling length of a scheduling algorithm. Therefore, the DAG analyzer designed in this tool is responsible for the analysis of graph parallelism and CCR of the DAG. From the values of graph parallelism and CCR, a programmer can check if the partitioned program meets the requirement. If it does not, a programmer needs to change the partitioning until the desired partitioning is obtained.

3. The scheduler

The scheduler is responsible for selecting a scheduling algorithm and the number of processors for execution (the scheduling algorithm and the number of processors selection can also be made by the programmer), scheduling a DAG on a target machine, and producing a (task, processor) table as well as the earliest start time table of tasks on every processor.

In the current development, PPT provides six list scheduling algorithms; the *highest level first with estimated time* (HLFET) [1], HLFET-BTDH [5], HLFET/BTDH [5], the *earlier task first* (ETF) [11], ETF-BTDH [5], and ETF/BTDH [5]. HLFET is a list scheduling algorithm which does not consider the interprocessor communication overhead, while ETF is a list scheduling algorithm that takes the interprocessor communication overhead into account. HLFET-BTDH, HLFET/BTDH, ETF-BTDH, and ETF/BTDH are list scheduling algorithms that use a task duplication heuristic, BTDH, to minimize the scheduling length. In [5], we have performed extensive simulation to study the relationship between the efficiency of different list scheduling algorithms and the properties of DAGs. A *relationship table* is constructed to describe the relation between those scheduling algorithms and the properties of DAGs. According to the output of the DAG analyzer, the scheduler consults the relationship table to find the best candidate scheduling algorithm and determine the number of processors for execution.

PPT allows a new scheduling algorithm as a member of the list scheduling algorithms. The only restriction is that the scheduling algorithm should perform scheduling and mapping simultaneously. This is true for nearly all variants of list scheduling algorithms. The clustering algorithms proposed in [19] and [20] do not fit into this category. Before a scheduling algorithm can be added as a member of the scheduling algorithm of the scheduler, the performance evaluation simulator, which is provided by PPT, must be executed for the scheduling algorithm in order to obtain the relationship between the scheduling algorithm and the properties of DAGs.

4. The communication analyzer

The communication analyzer is responsible for detecting redundant communication, removing redundant communication, and creating a *communication table*, which will be used by the code generator.

To detect and remove redundant communication and create the communication table, a communication analyzer must build a *task block table* according to the (task, processor) table and the earliest start time table. In the task block table, some consecutive tasks on the same processor are labeled with the same block number (if only the first and the last tasks of those consecutive tasks must send or receive data from other processors). According to the labeled communication table, a new communication table is created and redundant communication is removed.

5. The code generator

The code generator is responsible for generating the corresponding procedure call for each task on processors and inserting the communication primitives. According to the task block table, the corresponding procedure calls for tasks on processors are generated by consulting the attribute table. The communication primitives are inserted according to the communication table. The information about the data that must be transferred to tasks on other processors is provided by the attribute table.

Since the syntax of the basic communication primitives are machine dependent, there is a need for a communication primitive insertion routine. In the current development, we provide communication primitives insertion routines for NCUBE-2. One can potentially use syntax of commercial packages like EXPRESS to achieve portability. However, we do not use the communication primitives provided by these packages in our tool. This is because that the communication primitives provided by these packages are implemented by using the communication primitives provided by a machine. The overhead is significant and will usually reduce the performance of a parallel program. For example, on NCUBE-2, the time to execute the communication primitives provided by EXPRESS, in general, is 20% more than the time to execute the communication primitives provided by NCUBE-2.

6. The performance evaluator

The performance evaluator is responsible for executing the parallel program on a target machine and papering the execution time of the parallel program. It provides information about the execution time of each processor, and the predicted (simulation) and real (experimental) speedups of the parallel program. Since many distributed memory multiprocessors, such as NCUBE-2 and CM-5, provide the execution profiler for the programmer to check the time spent in the various subroutines and functions on each processor, the programmer can use the information provided by the performance evaluator to make further refinement/modification of the program partitioning.

IV. A PARALLEL FFT PROGRAM DESIGN EXAMPLE

In the following, we show how to design a parallel FFT program on NCUBE-2 by using PPT.

The program partitioning and DAG generator: An FFT algorithm, in general, can be described as follows [6]:

```
Algorithm FFT(A)
1. $n$ = length ($A$); /*$n$ is a power of 2*/
2. if ($n = 1$) then return ($A$);
3. $Y^{(0)} = FFT(A[0: n\text{-}2:2])$;
4. $Y^{(1)} = FFT(A[1: n\text{-}1:2])$;
5. $\omega_n = e^{2\pi i/n}$; $\omega = 1$;
6. for $k = 0$ to $n/2\text{-}1$ do
7.    { $Y[k] = Y^{(0)}[k] + \omega * Y^{(1)}[k]$;
8.      $Y[k+n/2] = Y^{(0)}[k] - \omega * Y^{(1)}[k]$;
9.      $\omega = \omega * \omega_n$;}
10. return($Y$); /*$Y$ is assumed to be column vector*/
```

In algorithm *FFT*, *A* and *Y* are arrays, where $A[0: n\text{-}2:2] = \{A[0], A[2], ..., A[n\text{-}2]\}$ and $A[1: n\text{-}1:2] = \{A[1], A[3], ..., A[n\text{-}1]\}$. Since we prefer to use the modular programming style, algorithm FFT needs to be rewritten. A sequential FFT program, which is written in modular programming style, is given in Fig. 3.

To partition a program, a programmer needs to study the behavior of the program. The behavior of algorithm FFT with input vector size = 4 is shown in Fig. 4. In Fig. 4, the computations of FFT consists of two operations, the *input vector operation* (IVO) (lines 3 and 4 in algorithm FFT) and the *butterfly operation* (BO) (lines 5 to 9 in algorithm FFT). Fig. 5 shows the general behavior of FFT. From Fig. 5, the programmer can decide how to partition an FFT program. In our example, we assume that the input vector size is 1024 and an FFT program is partitioned into 39 tasks. The corresponding DAG of the partitioned FFT program is shown in Fig. 6. In Fig. 6, there are seven IVO tasks (perform the IVO procedure), 8 FFT tasks (perform the FFT procedure), and 24 BO1 and

```
#define len 16384
double temp[len];
double A[len][2];

/****** Program ******/
main()
  {
    FFT(0, len);
  }

/****** FFT ******/
FFT(p, n)
int p, n;
  {
    if (n == 1) return;
    IVO(p, n);
    FFT(p, n/2);
    FFT(p+n/2, n/2);
    BO1(p, p+n/2, n/2);
    BO2(p, p+n/2, n/2);
  }

/****** Butterfly Operation *****/
/* Butterfly operation for        */
/* Y[k+n/2] = Y(0)[k] - ω * Y(1)[k];*/

BO2(p, q, n)
int p, q, n;
{
  int k;
  double w[2], x[2], u;

  for (k=0; k<n; k++)
   {
     u = 2 * 3.1416 * k / n;
     w[0] = cos(u); w[1] = sin(u);
     x[0] = A[q+k][0]*w[0] - A[q+k][1]*w[1];
     x[1] = A[q+k][0]*w[1] - A[q+k][1]*w[0];
     A[q+k][0] = A[p+k][0] - x[0];
     A[q+k][1] = A[p+k][1] - x[1];
   }
}

/****** Butterfly Operation *****/
/* Butterfly operation for        */
/*    Y[k] = Y(0)[k] + ω * Y(1)[k];   */

BO1(p, q, n)
int p, q, n;
{
  int k;
  double w[2], x[2], u;

  for (k=0; k<n; k++)
    {
      u = 2 * 3.1416 * k / n;
      w[0] = cos(u); w[1] = sin(u);
      x[0] = A[q+k][0]*w[0] - A[q+k][1]*w[1];
      x[1] = A[q+k][0]*w[1] - A[q+k][1]*w[0];
      A[p+k][0] = A[p+k][0] + x[0];
      A[p+k][1] = A[p+k][1] + x[1];
    }
}

/****** Input Vector Operation ******/
IVO(p, n)
{
  for (i=0; i<n; i=i+2) temp[p+i] = A[p+i][0];
  for (i=1; i<n; i=i+2) temp[p+i] = A[p+i][0];
  for (i=0; i<n; i++) A[p+i][0] = temp[p+i];
}
```

Fig. 3. A sequential FFT program with modular programming style.

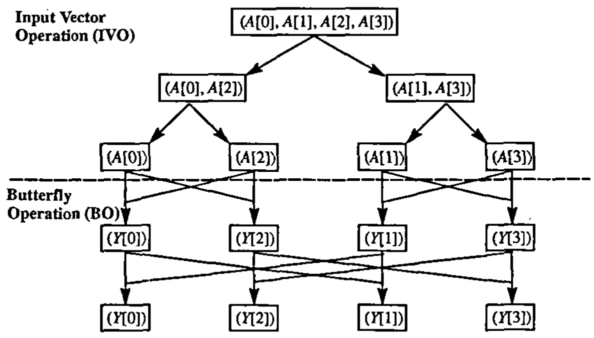

Fig. 4. The behavior of FFT with 4 points.

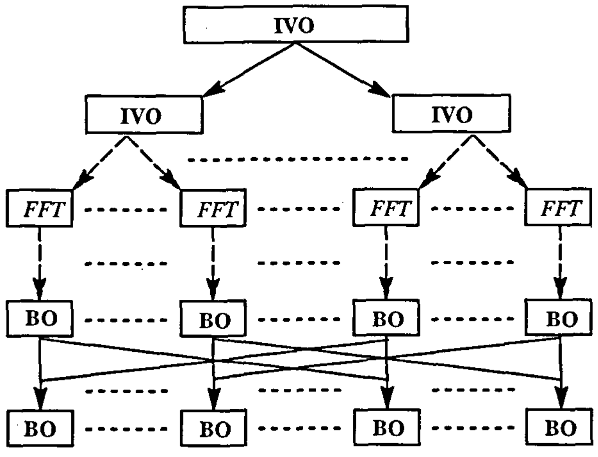

Fig. 5. The general behavior of FFT.

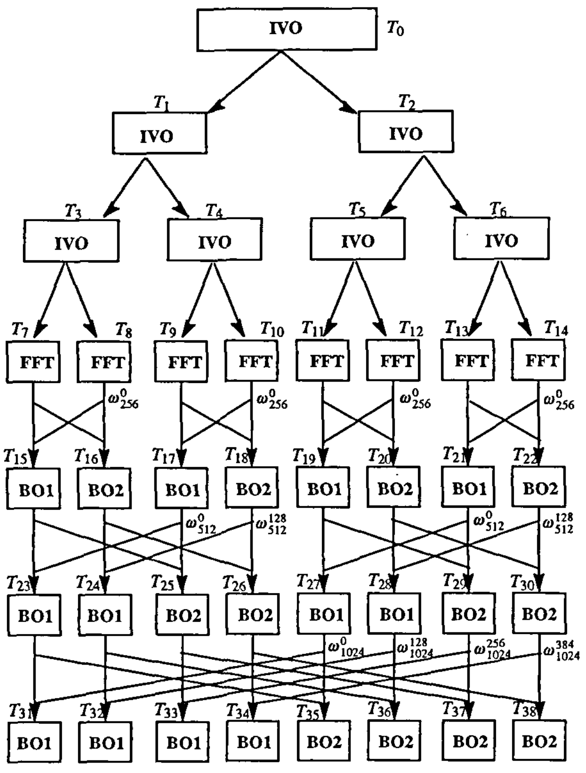

Fig. 6. The DAG generated for PPT.

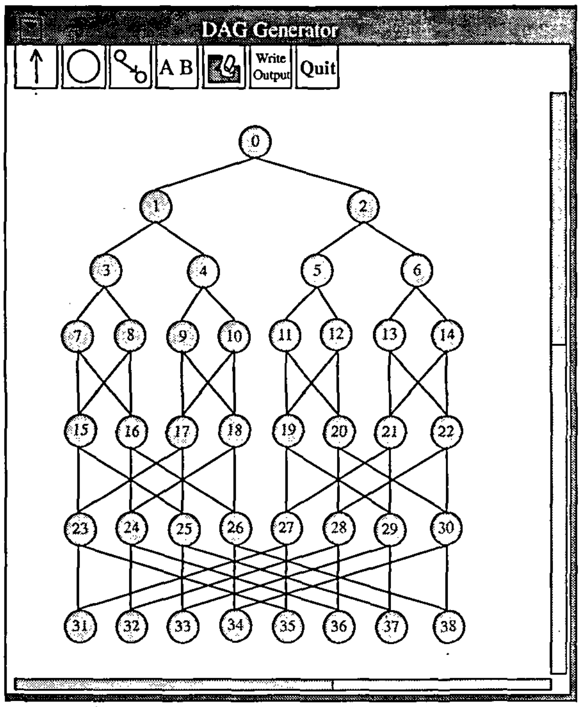

Fig. 7. The DAG generator & the corresponding DAG of FFT.

BO2 tasks (perform the BO1 and BO2 procedures). In Fig. 7, the corresponding DAG of Fig. 6 is drawn by using the DAG generator. During the drawing of a node or a link, a programmer will be asked to input the corresponding data for the node or the link. When the drawing of a DAG is completed, the DAG can be saved as a data file. The data file can be read by the DAG generator as well.

The DAG generator also generates the attribute table for a DAG. The information in the attribute table includes the computational load of each task, the communication cost between two tasks, the corresponding procedure name of each task, and the data must be sent from one task to another. The programmer is responsible for estimating this information (This could potentially be done automatically). The attribute table of the DAG shown in Fig. 6 is given in Table 1. The DAG and the attribute table are the inputs of the DAG analyzer.

The DAG analyzer: The values of CCR and the graph parallelism of the DAG shown in Fig. 6 are 0.34 and 7.08, respectively. In our example, we assume that the programmer is satisfied with the values of CCR and the graph parallelism.

The Scheduler: The scheduler can automatically select the number of processors and a scheduling algorithm based on the values of CCR and the graph parallelism of a DAG. Programmers can select the number of processors and a scheduling algorithm by themselves as well. The number of processors and scheduling algorithm selection submenus are shown in Fig. 8 and Fig. 9, respectively. Since the DAG is a coarse grain graph and the graph parallelism is 7.08, the scheduler selects the "ETF/BTDH" [5] as the scheduling algorithm and the number of processors used for scheduling is equal to 8 if automatic selection is chosen. The selecting criteria of the scheduling algorithm and the number of processors are based on the simulation results presented in [5]. The DAG is then scheduled and the scheduler produces the (task, processor) table and the earliest start time table, which are shown in Table 2 and Table 3, respectively.

The communication analyzer: According to the (task, processor) table and the earliest start time table, the communication analyzer first builds the task block table, which is shown in Table 4. The first task of a block must receive data from some other tasks and the last task of a block must send data to some other tasks. Tasks in between the first and the last tasks of a block do not need to send (or receive) data to (or from) other tasks. Based on the block table, a communication table is gener-

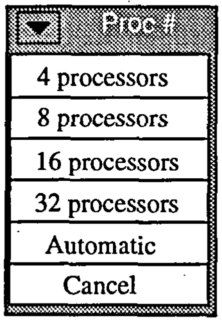

Fig. 8. The number of processor selection submenu.

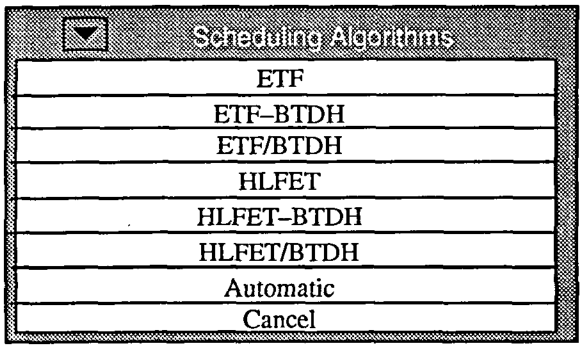

Fig. 9. The scheduler submenu.

Table 1. The attribute table for the DAG shown in Fig. 6

	Procedure	Vector Size	Start Position	Computation Cost (μs)
T_0	IVO	-	-	4198
T_1	IVO	-	-	2022
T_2	IVP	-	-	2022
T_3	IVO	-	-	998
T_4	IVO	-	-	998
T_5	IVO	-	-	998
T_6	IVO	-	-	998
T_7	FFT	-	-	32870
T_8	FFT	-	-	32870
T_9	FFT	-	-	32870
T_{10}	FFT	-	-	32870
T_{11}	FFT	-	-	32870
T_{12}	FFT	-	-	32870
T_{13}	FFT	-	-	32870
T_{14}	FFT	-	-	32870
T_{15}	BO1	256	0	3366
T_{16}	BO2	256	0	3366
T_{17}	BO1	256	0	3366
T_{18}	BO2	256	0	3366
T_{19}	BO1	256	0	3366
T_{20}	BO2	256	0	3366
T_{21}	BO1	256	0	3366
T_{22}	BO2	256	0	3366
T_{23}	BO1	512	0	3366
T_{24}	BO1	512	128	3366
T_{25}	BO2	512	0	3366
T_{26}	BO2	512	128	3366
T_{27}	BO1	512	0	3366
T_{28}	BO1	512	128	3366
T_{29}	BO2	512	0	3366
T_{30}	BO2	512	128	3366
T_{31}	BO1	1024	0	3366
T_{32}	BO1	1024	128	3366
T_{33}	BO1	1024	256	3366
T_{34}	BO1	1024	384	3366
T_{35}	BO2	1024	0	3366
T_{36}	BO2	1024	128	3366
T_{37}	BO2	1024	256	3366
T_{38}	BO2	1024	384	3366

ated and is given in Table 5.

The code generator: From the attribute table, the block table, and the communication table, the code generator generates the corresponding parallel code for each processor. After clicking the "Code Generator" selection item, the programmer will be asked to give a file name (and path) to save the corresponding parallel code, which is shown in Fig. 10, generated by PPT.

The performance evaluator: After clicking the "Performance Evaluator" selection item, the generated parallel code is executed on 8 processors of an NCUBE-2. The table of the execution time of each processor is created and is shown in Table 6. The comparison of the predicted speedup (simulation) and the real speedup (experimental) is also given. In our example, the predicted and real speedups are 6.08 and 5.67, respectively.

V. CONCLUSIONS AND FUTURE WORK

As both the number of processors and the

Table 1. (cont'd) The attribute table for the DAG shown in Fig. 6

	Send					
	Task	Data	Communication Cost (μs)	Task	Data	Communication Cost (μs)
T_0	T_1	A[0..511]	4296	T_2	A[512..1023]	4296
T_1	T_3	A[0..255]	2248	T_4	A[256..511]	2248
T_2	T_5	A[512..767]	2248	T_6	A[768..1023]	2248
T_3	T_7	A[0..127]	1224	T_8	A[128..255]	1224
T_4	T_9	A[256..383]	1224	T_{10}	A[384..511]	1224
T_5	T_{11}	A[512..639]	1224	T_{12}	A[640..767]	1224
T_6	T_{13}	A[768..895]	1224	T_{14}	A[896..1023]	1224
T_7	T_{15}	A[0..127]	712	T_{16}	A[0..127]	712
T_8	T_{15}	A[128..255]	712	T_{16}	A[128..255]	712
T_9	T_{17}	A[256.383]	712	T_{18}	A[256..383]	712
T_{10}	T_{17}	A[384..511]	712	T_{18}	A[384..511]	712
T_{11}	T_{19}	A[512..639]	712	T_{20}	A[512..639]	712
T_{12}	T_{19}	A[640..767]	712	T_{20}	A[640..767]	712
T_{13}	T_{21}	A[768..895]	712	T_{22}	A[768..895]	712
T_{14}	T_{21}	A[896..1023]	712	T_{22}	A[896..1023]	712
T_{15}	T_{23}	A[0..127]	712	T_{25}	A[0..127]	712
T_{16}	T_{24}	A[128..255]	712	T_{26}	A[128..255]	712
T_{17}	T_{23}	A[256.383]	712	T_{25}	A[256.383]	712
T_{18}	T_{24}	A[384..511]	712	T_{26}	A[384..511]	712
T_{19}	T_{27}	A[512..639]	712	T_{29}	A[512..639]	712
T_{20}	T_{28}	A[640..767]	712	T_{30}	A[640..767]	712
T_{21}	T_{27}	A[768..895]	712	T_{29}	A[768..895]	712
T_{22}	T_{28}	A[896..1023]	712	T_{30}	A[896..1023]	712
T_{23}	T_{31}	A[0..127]	712	T_{35}	A[0..127]	712
T_{24}	T_{32}	A[128..255]	712	T_{36}	A[128..255]	712
T_{25}	T_{33}	A[256.383]	712	T_{37}	A[256.383]	712
T_{26}	T_{34}	A[384..511]	712	T_{38}	A[384..511]	712
T_{27}	T_{31}	A[512..639]	712	T_{35}	A[512..639]	712
T_{28}	T_{32}	A[640..767]	712	T_{36}	A[640..767]	712
T_{29}	T_{33}	A[768..895]	712	T_{37}	A[768..895]	712
T_{30}	T_{34}	A[896..1023]	712	T_{38}	A[896..1023]	712
T_{31}	-	-	-	-	-	-
T_{32}	-	-	-	-	-	-
T_{33}	-	-	-	-	-	-
T_{34}	-	-	-	-	-	-
T_{35}	-	-	-	-	-	-
T_{36}	-	-	-	-	-	-
T_{37}	-	-	-	-	-	-
T_{38}	-	-	-	-	-	-

complexity of problems to be solved increase, programming distributed memory multiprocessors becomes difficult and error-prone. In this paper, we have presented a parallel programming tool, PPT, to aid programmers to design a parallel program for the *single program multiple data* (SPMD) model on a distributed memory multiprocessor. PPT provides functions for DAG generation, scheduling, communication primitives insertion, and parallel code generation. An example of designing a parallel FFT program by using PPT on an NCUBE-2. is also presented. Our experiments with PPT show that an automatically generated parallel program can attain good performance. PPT can help the programmer to design a parallel program by using the coarse grain, medium grain, or fine grain DAGs. It can also increase programming productivity by automating parts of parallelizing tasks.

Table 2. The (task, processor) table

order \ processor	0	1	2	3	4	5	6	7
1	0	0	0	0	0	0	0	0
2	1	1	2	1	2	1	2	2
3	3	3	5	4	6	4	5	6
4	7	8	11	10	13	9	12	14
5	16	15	19	18	21	20	17	22
6	26	23	27	29	24	28	25	30
7	35	31	36	37	32	33	38	34

The current version of PPT uses static scheduling approaches and is applied to the static problems. That is, all tasks must be created before starting execution. Also, the computation and communication costs must be known at compile time. In the future, we plan to integrate an interactive dependency analyzer to facilitate the specification of task parallelism and run-time compilation techniques used in PARTI [7].

The main advantage of the SPMD model is that a parallel program is easy to design under this

Table 3. The earliest start time table for tasks

order \ processor	0	1	2	3	4	5	6	7
1	0	0	0	0	0	0	0	0
2	4198	4198	4198	4198	4198	4198	4198	4198
3	6297	6297	6297	6297	6297	6297	6297	6297
4	7346	7346	7346	7346	7346	7346	7346	7346
5	42976	42976	42976	42976	42976	42976	42976	42976
6	49102	49102	49102	49102	49102	49102	49102	49102
7	55228	55228	55228	55228	55228	55228	55228	55228

Table 4. The task block table

Order \ Processor	0		1		2		3		4		5		6		7	
	Task	Block No.	Task	Block No.	Task	Block No.	Task	Block No.	Task	Block No.	Task	Block No.	Task	Block No.	Task	Block No.
1	0	0	0	4	0	8	0	12	0	16	0	20	0	24	0	28
2	1	0	1	4	2	8	1	12	2	16	1	20	2	24	2	28
3	3	0	3	4	5	8	4	12	6	16	4	20	5	24	6	28
4	7	0	8	4	11	8	10	12	13	16	9	20	12	24	14	28
5	16	1	15	5	19	9	18	13	21	17	20	21	17	25	22	29
6	26	2	23	6	27	10	29	14	29	18	28	22	25	26	30	30
7	35	3	31	7	36	11	37	15	37	19	33	23	38	27	34	31

Table 5. The communication table

	Send	Receive		Send	Receive		Send	Receive
B_0	B_5	-	B_{12}	B_{25}	-	B_{24}	B_9 B_{21}	-
B_1	B_{18}	B_4	B_{13}	B_2 B_{18}	B_{20}	B_{25}	B_6	B_{12} B_{20}
B_2	B_{27} B_{31}	B_{13}	B_{14}	B_{23}	B_9 B_{17}	B_{26}	B_{15} B_{23}	B_5
B_3	-	B_6 B_{10}	B_{15}	-	B_{26}	B_{27}	-	B_2 B_{30}
B_4	B_1	-	B_{16}	B_{29}	-	B_{28}	B_{17}	-
B_5	B_{26}	B_0	B_{17}	B_{10} B_{14}	B_{28}	B_{29}	B_{22}	B_{16}
B_6	B_3	B_{25}	B_{18}	B_{11}	B_1 B_{13}	B_{30}	B_{27}	B_2 B_{21}
B_7	-	B_{10}	B_{19}	-	B_{22}	B_{31}	-	
B_8	B_{21}	-	B_{20}	B_{13} B_{25}	-			
B_9	B_{14}	B_{24}	B_{21}	B_{30}	B_8 B_{24}			
B_{10}	B_7	B_{17}	B_{22}	B_{11} B_{19}	B_{29}			
B_{11}	-	B_{18} B_{22}	B_{23}	-	B_{14} B_{26}			

```
#define len 1024
double temp[len];
double A[len][2];

/****** Program ******/
main()
  {
    my_addr = npid();
    switch (my_addr)
      {
        case 0 : PE0(); break;
        case 1 : PE1(); break;
        case 2 : PE2(); break;
        case 3 : PE3(); break;
        case 4 : PE4(); break;
        case 5 : PE5(); break;
        case 6 : PE6(); break;
        case 7 : PE7(); break;
      }
  }

/****** Butterfly Operation *****/
/* Butterfly operation for          */
/* Y[k+n/2] = Y(0)[k] – ω * Y(1)[k];*/

BO2(p, q, n, vs, sp)
int p, q, n, vs, sp;
{
   int k;
   double w[2], x[2], u;

   for (k=0; k<n; k++)
     {
       u = (2 * 3.1416 * k + sp) / vs;
       w[0] = cos(u); w[1] = sin(u);
       x[0] = A[q+k][0]*w[0] – A[q+k][1]*w[1];
       x[1] = A[q+k][0]*w[1] – A[q+k][1]*w[0];
       A[q+k][0] = A[p+k][0] – x[0];
       A[q+k][1] = A[p+k][1] – x[1];
     }
}

/****** Butterfly Operation *****/
/* Butterfly operation for          */
/*    Y[k] = Y(0)[k] + ω * Y(1)[k];  */

BO1(p, q, n, vs, sp)
int p, q, n, vs, sp;
{
  int k;
  double w[2], x[2], u;

  for (k=0; k<n; k++)
     {
       u = (2 * 3.1416 * k + sp) / vs;
       w[0] = cos(u); w[1] = sin(u);
       x[0] = A[q+k][0]*w[0] – A[q+k][1]*w[1];
       x[1] = A[q+k][0]*w[1] – A[q+k][1]*w[0];
       A[p+k][0] = A[p+k][0] + x[0];
       A[p+k][1] = A[p+k][1] + x[1];
     }
}

/****** Input Vector Operation ******/
IVO(p, n)
{
   for (i=0; i<n; i=i+2) temp[p+i] = A[p+i][0];
   for (i=1; i<n; i=i+2) temp[p+i] = A[p+i][0];
   for (i=0; i<n; i++) A[p+i][0] = temp[p+i];
}

/****** FFT ******/
FFT(p, n)
int p, n;
  {
    if (n == 1) return;
    IVO(p, n);
    FFT(p, n/2);
    FFT(p+n/2, n/2);
    BO1(p, p+n/2, n/2, n/2, 0);
    BO2(p, p+n/2, n/2, n/2, 0);
  }
```

Fig. 10. The generated parallel code.

```
PE0()
 {
   IVO(0, 1024);
   IVO(0, 512);
   IVO(0, 256);
   FFT(0, 128);
   nwrite(&A[0][0], 2048, pa[1], tag[7], 0);
   nread(&A[128][0], 2048, &pa[1], &tag[8], 0);
   BO2(0, 128, 128, 256, 0);
   nwrite(&A[0][0], 2048, pa[4], tag[16], 0);
   nread(&A[256][0], 2048, &pa[3], &tag[18], 0);
   BO2(128, 384, 128, 512, 1288);
   nwrite(&A[128][0], 2048, pa[6], tag[26], 0);
   nwrite(&A[128][0], 2048, pa[7], tag[26], 0);
   nread(&A[0][0], 2048, &pa[1], &tag[23], 0);
   nread(&A[512][0], 2048, &pa[2], &tag[27], 0);
   BO2(0, 512, 128, 1024, 0);
 }

PE1()
 {
   IVO(0, 1024);
   IVO(0, 512);
   IVO(0, 256);
   FFT(128, 128);
   nwrite(&A[128][0], 2048, pa[0], tag[8], 0);
   nread(&A[0][0], 2048, &pa[0], &tag[7], 0);
   BO1(0, 128, 128, 256, 128);
   nwrite(&A[0][0], 2048, pa[6], tag[15], 0);
   nread(&A[256][0], 2048, &pa[6], &tag[17], 0);
   BO1(0, 256, 128, 512, 256);
   nwrite(&A[0][0], 2048, pa[0], tag[23], 0);
   nread(&A[512][0], 2048, &pa[2], &tag[27], 0);
   BO1(0, 512, 128, 1024, 512);
 }

PE2()
 {
   IVO(0, 1024);
   IVO(512, 512);
   IVO(512, 256);
   FFT(512, 128);
   nwrite(&A[512][0], 2048, pa[5], tag[11], 0);
   nread(&A[640][0], 2048, &pa[6], &tag[12], 0);
   BO1(512, 640, 128, 256, 128);
   nwrite(&A[512][0], 2048, pa[3], tag[19], 0);
   nread(&A[768][0], 2048, &pa[4], &tag[21], 0);
   BO1(512, 768, 128, 512, 256);
   nwrite(&A[512][0], 2048, pa[0], tag[27], 0);
   nwrite(&A[512][0], 2048, pa[0], tag[27], 0);
   nread(&A[128][0], 2048, &pa[4], &tag[24], 0);
   nread(&A[640][0], 2048, &pa[5], &tag[28], 0);
   BO1(128, 640, 128, 1024, 128);
 }

PE3()
 {
   IVO(0, 1024);
   IVO(0, 512);
   IVO(256, 256);
   FFT(384, 128);
   nwrite(&A[384][0], 2048, pa[6], tag[10], 0);
   nread(&A[256][0], 2048, &pa[5], &tag[9], 0);
   BO2(256, 384, 128, 256, 0);
   nwrite(&A[256][0], 2048, pa[0], tag[18], 0);
   nwrite(&A[256][0], 2048, pa[4], tag[18], 0);
   nread(&A[512][0], 2048, &pa[2], &tag[19], 0);
   nread(&A[768][0], 2048, &pa[4], &tag[21], 0);
   BO2(512, 768, 128, 512, 0);
   nread(&A[512][0], 2048, &pa[5], &tag[29], 0);
   nread(&A[0][0], 2048, &pa[6], &tag[25], 0);
   BO2(256, 768, 128, 1024, 256 );
 }
```

Fig. 10. (cont'd) The generated parallel code.

```
PE4()
 {
   IVO(0, 1024);
   IVO(512, 512);
   IVO(768, 256);
   FFT(768, 128);
   nwrite(&A[768][0], 2048, pa[7], tag[13], 0);
   nread(&A[896][0], 2048, &pa[7], &tag[14], 0);
   BO1(768, 896, 128, 256, 128);
   nwrite(&A[768][0], 2048, pa[2], tag[21], 0);
   nwrite(&A[768][0], 2048, pa[3], tag[21], 0);
   nread(&A[0][0], 2048, &pa[0], &tag[16], 0);
   nread(&A[256][0], 2048, &pa[3], &tag[18], 0);
   BO1(128, 384, 128, 512, 384);
   nwrite(&A[128][0], 2048, pa[2], tag[24], 0);
   nread(&A[640][0], 2048, &pa[5], &tag[28], 0);
   BO1(128, 640, 128, 1024, 640);
 }

PE5()
 {
   IVO(0, 1024);
   IVO(0, 512);
   IVO(256, 256);
   FFT(256, 128);
   nwrite(&A[256][0], 2048, pa[3], tag[9], 0);
   nwrite(&A[256][0], 2048, pa[6], tag[9], 0);
   nread(&A[512][0], 2048, &pa[2], &tag[11], 0);
   nread(&A[640][0], 2048, &pa[6], &tag[12], 0);
   BO2(512, 640, 128, 256, 0);
   nwrite(&A[512][0], 2048, pa[7], tag[20], 0);
   nread(&A[768][0], 2048, &pa[7], &tag[22], 0);
   BO1(640, 896, 128, 512, 384);
   nwrite(&A[640][0], 2048, pa[2], tag[28], 0);
   nwrite(&A[640][0], 2048, pa[4], tag[28], 0);
   nread(&A[0][0], 2048, &pa[6], &tag[25], 0);
   nread(&A[512][0], 2048, &pa[3], &tag[29], 0);
   BO1(256, 768, 128, 1024, 768);
 }

PE6()
 {
   IVO(0, 1024);
   IVO(512, 512);
   IVO(512, 256);
   FFT(640, 128);
   nwrite(&A[640][0], 2048, pa[2], tag[12], 0);
   nwrite(&A[640][0], 2048, pa[5], tag[12], 0);
   nread(&A[256][0], 2048, &pa[5], &tag[9], 0);
   nread(&A[384][0], 2048, &pa[3], &tag[10], 0);
   BO1(256, 384, 128, 256, 128);
   nwrite(&A[256][0], 2048, pa[1], tag[17], 0);
   nread(&A[0][0], 2048, &pa[1], &tag[15], 0);
   BO2(0, 256, 128, 512, 0);
   nwrite(&A[0][0], 2048, pa[3], tag[25], 0);
   nwrite(&A[0][0], 2048, pa[5], tag[25], 0);
   nread(&A[128][0], 2048, &pa[0], &tag[26], 0);
   nread(&A[640][0], 2048, &pa[7], &tag[30], 0);
   BO2(384, 896, 128, 1024, 384);
 }

PE7()
 {
   IVO(0, 1024);
   IVO(512, 512);
   IVO(768, 256);
   FFT(896, 128);
   nwrite(&A[896][0], 2048, pa[4], tag[14], 0);
   nread(&A[768][0], 2048, &pa[4], &tag[13], 0);
   BO2(768, 896, 128, 256, 0);
   nwrite(&A[768][0], 2048, pa[5], tag[22], 0);
   nread(&A[512][0], 2048, &pa[5], &tag[20], 0);
   BO2(640, 896, 128, 512, 128);
   nwrite(&A[640][0], 2048, pa[6], tag[30], 0);
   nread(&A[128], 2048, &pa[0], &tag[26], 0);
   BO1(384, 896, 128, 1024, 896);
 }
```

Fig. 10. (cont'd) The generated parallel code.

Table 6. The execution time (computation and communication) and speedup table

processor	Execution Time (μs)
0	64603
1	63671
2	65646
3	63434
4	64608
5	64557
6	64718
7	64759
Maximum	65646
Sequential	371942
Predicted Speedup	6.08
Real Speedup	5.67

model. However, the main memory capacity of each processing node is the main restriction of using this model. Assume that every processing node of an NCUBE-2, which has 32 processing nodes, has 4 Megabytes memory. The total memory capacity of the NCUBE-2 is 128 Megabytes. If a node program requires memory over 4 Megabytes, it is impossible to execute the program on an NCUBE-2 if the program is written in the SPMD model. However, if the program is written in the host-node model, it can be executed on an NCUBE-2. This implies that the host-node execution model is more general than the SPMD model. In the future, we also plan to add a host-node code generator in the code generator phase to support the host-node execution model.

ACKNOWLEDGEMENT

The authors would like to thank the National Science Council of R.O.C. for financial support of this work through grant NSC82-0113-E035-044T.

REFERENCES

1. Adam, T.L., K.M. Chandy and J.R. Dickson, "A Comparison of List Schedules for Parallel Processing Systems," *Communication of ACM*, Vol. 17, No. 12, pp. 685-690 (1974).
2. Allen, R., D. Baumgartner, K. Kennedy and A. Porterfield, "PTOOL: A Semiautomatic Parallel Programming Assistant," *Proceedings of International Conference on Parallel Processing*, pp. 164-170, Chicago, IL (1986).
3. Cheng, D.Y. and D.M. Pase, "An Evaluation of Automatic and Interactive Parallel Programming Tools," *Proceedings of Supercomputing*, pp. 412-423 (1991).
4. Chung, Y.C. and S. Ranka, "A Compile-Time Optimization Approach for Scheduling of Tasks on Distributed Memory Multiprocessor," *Proceedings of International Conference of Parallel and Distributed Systems*, pp. 87-91, Taipei, Taiwan (1993).
5. Chung, Y.C. and S. Ranka, "Applications and Performance Analysis of a Compiler-Time Optimization Approach for List Shceduling Algorithms on Distributed Memory Multiprocessors," *Proceedings of Supercomputing'92*, pp. 512-521, Minneapolis, MN (1992).
6. Cormen, T.H., C.E. Leiserson and R.L. Rivest, *Introduction to Algorithms*, McGraw-Hill (1991).
7. Das, R., R. Ponnusamy, J. Saltz and D. Mavriplis, "Distributed Memory Compiler Methods for Irregular Problems: Data Reuse and Runtime Partitioning," ICASE Technical Report, No. 91-73 (1991).
8. Dongarra, J.J. and D.C. Sorensen, "SCHEDULE: Tools for Developing and Analyzing Parallel Fortran Program," *The Characteristics of Parallel Algorithms*, pp. 363-394, MIT Press, Cambridge, Massachusetts (1987).
9. El-Rewini, H. and T.G. Lewis, "Scheduling Parallel Program Tasks onto Arbitrary Target Machines," *Journal of Parallel and Distributed Computing*, Vol. 9, pp. 138-153 (1990).
10. Hayes, J. and T. Mudge, "Architecture of a Hypercube Supercomputer," *Proceedings of International Conference on Parallel Processing*, pp. 653-660, Chicago, IL (1986).
11. Hwang, J.J., Y.C. Chow, F.D. Anger and C.Y.Lee, "Scheduling Precedence Graphs in Systems with Interprocessor Communication Times," *SIAM Journal of Computing*, Vol. 18, pp. 244-257 (1989).
12. Padua, D.A., D.J. Kuck and D.L. lawrie, "High Speed Multiprocessors and Compilation Techniques," *IEEE Transactions on Computers*, Vol. 29, No. 9, pp. 763-776 (1980).
13. Peir, J.K. and D.D. Gajski, "CAMP: A Programming Aid for Multiprocessors," *Proceedings of International Conference on Parallel Processing*, pp. 475-482, Chicago, IL (1986).
14. Polychronopoulos, C.D., M.B. Girkar, M.R. Haghighat, C.L. Lee, B.P. Leung and D.A. Schouten, "The Structure of Parafrase-2: An Advanced Parallelizing Compiler for C and Fortran," in *Languages and Compilers for Parallel Processing, D. Gelernter, A. Nicolau and D. Padua*, eds. (1990).
15. Seitz, C.L., "The Cosmic Cube," *Communications of ACM*, Vol. 28, pp. 22-23, January 1985.
16. Sih, G.C. and E.A. Lee, "Scheduling to Account for Interprocessor Communication within Interconnection-Constrained Processor Networks," *Proceedings of International Conference on Parallel Processing*, Vol. 1, pp. 9-16, Chicago IL (1990).
17. Snyder, L., "Parallel Programming and the POKER Programming Environment," *IEEE Computer Magazines*, July, pp. 27-36 (1984).
18. Tucher, L.W. and G.G. Robertson, "Architecture and Applications of the Connection Machine," *IEEE Computer Magazines*, August, pp. 26-38 (1988).
19. Wu, M.Y. and D.D. Gajski, "Hypertool: A Programming Aid for Message Passing Systems," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 1, No. 3, pp. 330-343 (1990).
20. Yang, T. and A. Gerasoulis, "A Parallel Programming Tool for Scheduling on Distributed Memory Multiprocessors," *Proceeding of Scalabile High Performance Computers Conference*, pp. 350-357, Williamsburg, MD (1992).

Discussions of this paper may appear in the discussion section of a future issue. All discussions should be submitted to the Editor-in-Chief.

Manuscript Received: April 28, 1994
and Accepted: Aug. 15, 1994

PPT：分散多重處理器之平行程式發展工具

鍾葉青　何武勳　劉嘉政

逢甲大學資訊工程學系

摘　要

傳統上，要在分散式多重處理器上發展應用平行程式，程式設計師要負責將應用程式分割成許多小單元或工作，將這此工作排表於處理器上，加入必要的傳送的指令，以及爲每一處理器寫相對的平行程式碼。當處理器的數目和欲解決問題的複雜度增加時，要在分散式多重處理器上發展程式就變得困難，而且容易犯許多的錯誤。

在分散式多重處理器上，應用程式的分割及排表對於應用程式的執行成效，扮演著非常重要的角色。但是，如何找到最好的程式分割及排表使得應用程式能在多重處理器上能有最好的執行成效並不是一件容易的工作。在這計畫，我們提出一程式設計輔助工具，PPT，來幫助程式設計師半自動地找出最好的程式分割，自動地加入適當的傳送指令，而且自動地爲每一處理器產生它的單一程式多重資料的平行程式碼。

關鍵詞：單一程式多重資料程式，分散式記憶體多處理器，平行程式輔助工具，程式分割，排表。