# Design and Implementation of a User-interactive Parallel Programming Environment

**Tzung-Shi Chen, Kuei-Ping Shih, and Jang-Ping Sheu**[1]

Department of Computer Science and Information Engineering
National Central University, Chung-Li, Taiwan
sheujp@csie.ncu.edu.tw

[1]To whom all correspondence should be addressed.

## Abstract

A state-of-the-art parallel programming environment called UPPER (User-interactive Parallel Programming EnviRonment) is presented in this paper. Parallel machines which execute programs concurrently on hundreds or thousands of processors provide far more computational power than does a uniprocessor. However, designing parallel programs on parallel machines manually is very difficult and error-prone. Due to these problems, many tools which help programmers translate sequential programs into parallelized programs or even help them design parallel programs have been developed. The proposed environment also has the same purpose. The major components of this environment include a parallelizing compiler system and simulators of the given target machines. The parallelizing compiler system introduces new and existing techniques for compiler-time analysis, and the simulator can simulate execution of the translated parallelized program on the target machine and show the simulated performance reports. This integrated environment attempts to provide convenience for users or programmers who can easily design or write their desirable parallel programs based on a variety of assertions and information generated by this environment. Using our environment, programmers can avoid the necessity of designing parallel programs and can obtain efficient parallelized programs from sequential programs easily.

**Keywords:** Distributed memory multicomputers, interprocessor communication, parallel programming, parallelizing compilers, shared memory multiprocessors, simulators.

# I. Introduction

Parallel processing is the most promising approach to designing and establishing high-performance computers. Parallel computers with hundreds of moderate-sized processors or thousands of simple processors are commercially available and are being used to solve various practical problems. The programming environment, a collection of software tools and system software, for parallel machines is more demanding than that for sequential machines. This is because engineers spend much time concentrating on designing the hardware of parallel computers instead of that on programming parallelism into programs running on parallel computers. To reduce the gap between hardware and software, we need a parallel programming environment which offers better tools for users to extract parallelism and to debug programs. The most important goal of a programming environment is to design excellent compilers with user-defined parallel constructs (parallel languages) or to develop parallelizing compilers which automatically translate a sequential program into a parallel executable form.

Therefore, several parallel programming environments, to be sure, have been designed and implemented on a variety of parallel machines for the purpose of saving the effort involved in developing parallel programs. The Parafrase-II project (Polychronopoulos et al., 1989) was one of the first attempts to design and implement a source to source multilingual restructuring compiler which supports the C and FORTRAN languages. It is portable, easy to extend,

and powerful due to its compiling capabilities. The Parallel TRANslator, PTRAN, (Allen et al., 1988) is a research system used to develop technology for automatic exploitation of parallelism. The Tiny research tool (Wolfe, 1991), which provides several elementary transformations, allows a user to interactively restructure the loops in a program. The ParaScope project (Kennedy et al., 1991) develops an integrated collection of tools to assist scientific programmers in implementing correct and efficient parallel programs. This environment can build dependences, provide expert advice, and perform complex transformations (Padua and Wolfe, 1986) (Wolfe, 1989) while the programmer determines which dependences are valid and chooses those transformations to be applied.

SUPERB is a semi-automatic parallelization system which includes SIMD and MIMD parallelization for the SUPRENUM multiprocessor (Zima et al., 1988). This system is oriented toward the parallelization of numerical programs which work in a mesh or mesh-like data domain where the computations at the mesh points are local. Rogers and Pingali (Rogers and Pingali, 1989) worked on compilation of the data flow language ID Nouveau for distributed memory machines. They used a fixed domain decomposition method to assign data to processors and to automatically generate individual send and receive pairs for passing of data blocks among processors. The compiler of Fortran D (Hiranandani et al., 1992) is used to compile a sequential program with specification of data alignment and data distribution. Its goals are to provide a machine-independent programming model for data-parallel applications and to shift the burden of machine-dependent optimization to the compiler. As described in (Koelbel et al., 1990) and (Koelbel and Mehrotra, 1991), KALI is a system which compiles a functional language with a parallel construct into a language which includes constructs for explicit process creation, data storage layout, and interprocessor communication. It is the first compiler to support both regular and irregular computations on MIMD distributed memory machines. However, KALI still leaves the tasks of parallelism extraction and data partition to the programmer since it only removes the task of communication generation from the programmer. CRYSTAL (Li and Chen, 1991) is a high-level functional language having a parallel construct compiled to distributed memory machines using both automatic data decomposition and communication generation. This compiler tries to choose a data decomposition so as to minimize the time spent on data communication. To achieve this goal, a part of the data layout will match the combination of the program reference pattern and communication aggregates.

Recently, the PARADIGM project (Gupta and Banerjee, 1992) (Su et al., 1993) has developed a fully automated technique for translating serial programs for efficient execution on distributed memory multicomputers. In addition, the Stanford SUIF compiler system (Tjiang et al., 1992) (Wolf and Lam, 1991) derives data and computation decomposition automatically for distributed memory machines. It solves the problem of global optimization for parallelism and data locality. It can also handle more flexible data decompositions and find more opportunities for communication optimization (Amarasinghe and Lam, 1993) (Anderson and Lam, 1993).

The parallel programming environment called UPPER (User-interactive Parallel Programming EnviRonment) is another attempt in this area. To develop parallelized programs, it records a given sequential program's the data and control information and their relationships for users, and indicates the effectiveness of various program transformations through the user interface. Moreover, the programmer can give a few suggestions to enable the parallelizing compiler to look for more parallelism in programs. This environment also provides convenience for programmers who can easily design desirable parallel programs. It differs from the other interactive environments mentioned above in its new compilation techniques (Chen and Sheu, 1994) (Sheu and Chen, 1995) and simulators of parallel computers. For different types of parallel machines, various program transformations have been deeply studied and designed (Chu, 1993) (Ni, 1993). Based on the execution performance as analyzed by simulators, programmers can decide whether to leave the parallelized codes as the final result or to apply different parallelizing techniques to the programs.

The rest of this paper is organized as follows. An overview of UPPER is briefly given in Section 2. In Section 3, we present the detailed implementation as well as the user interface responsible for interaction and exhibition of various modules and graphics. The machine-independent phase which deals with the preprocessing of a source program, dependence analysis, and our proposed compilation strategies, which have appeared in (Chen and Sheu, 1994) (Sheu and Chen, 1995), is presented in Section 4. Section 5 states the implementation issues of the machine-dependent phase of the parallelizing compiler. During the machine-dependent phase, we handle mapping and scheduling of a transformed program onto target machines specified by users. In section 6, we describe the simulators of shared memory multiprocessors and distributed memory multicomputers in which the resulting parallelized codes are simulated and analyzed. We finally give conclusions in Section 7.

## II. An Overview of UPPER

In this section, an overview of our parallel programming environment, UPPER, is presented. The whole integrated system has been implemented and designed on DEC workstations with the MOTIF environment. The interactive programming environment provides users with all of the information which is available to the automatic environment. According to the procedure for compiling a sequential program into a parallelized code, the description of each module in this environment and relations among the modules are briefly stated as follows.

The main configuration of the parallel programming environment is shown in Figure 1. The flow of control and data among the modules are represented by solid and dashed lines, respectively. In the remaining sections, we will only focus on descriptions of a parallelizing compiler for multicomputers, excluding the vector compiler enclosed in the dashed line in Figure 1. The major components of this environment include a parallelizing compiler system and simulators of given target machines. The parallelizing compiler system consists of

two phases: a machine-independent phase and a machine-dependent phase. The machine-independent phase, including the *preprocessing*, *dependence analysis*, and *program transformation* modules, exploits the parallelism of a given sequential program, regardless of machine topologies and properties. The next phase, the machine-dependent phase, including the *data distribution and program scheduling*, and *code generation* modules, uses the input data and information generated by the first phase to produce the parallel execution code according to machine topologies, size, and architectures. In addition, the *database* module is designed for all of the data and information generated or accessed by each module.

In this environment, the tasks of the *user interface* module are communication and interaction between the environment and users. The user interface can be made not only to easily use this environment such as by editing a sequential or parallel program, by showing the dependence information and the output results, etc., but also to interactively modify or restructure the sequential program into a parallelized or vectorized form so that better execution code performance can be obtained.

The *preprocessing* module with input from the sequential program and the target machine is used to scan, parse, and construct the program representation and the information on data flow for later use. The language used is FORTRAN. Currently, only its subset has been considered for implementing this environment since this parallelizing compiler is a research tool. The grammars which we have considered are shown in Appendix A. The structure of the procedures and function calls and complex constructions will be incorporated into this parallelizing compiler in the future. The major tools used here for scanning and parsing of a given program are *lex* and *yacc*, respectively.

After preprocessing, some information is created for use by the *dependence analysis* module. The control flow of a program is represented by a tree structure, which is referred as the *program representation* (Ferrante et al., 1987). A basic block in the program is identified as a basic structure in the corresponding program representation in order to clearly distinguish the control flow and to easily manipulate this tree structure. For each statement in the sequential program, its representation with one or several complex expressions is also incorporated into the program representation. Based on this representation, information about analyzed data dependence is also appended and is reported to users in a graphic manner. The data dependence information is used to guide subsequent compiler analysis and optimization such as by reporting bottlenecks in the program parallelization and opportunities for exploiting the parallelism of program. The popular and well-known methods of data dependence testing, including the GCD test and Banerjee-Wolfe test (Banerjee, 1988) (Wolfe and Banerjee, 1987) (Wolfe, 1992), have been implemented in this system. Several more powerful methods of data dependence testing such as the $\lambda$ test (Li et al., 1990), power test (Wolfe and Tseng, 1992), and omega test (Pugh, 1992) will be studied and implemented in the future.

The *program transformation* module contains the submodules of *program parallelization*, *data parallelization*, *data and program parallelization*, and *program vectorization*. This module utilizes the results of dependence analysis to improve program performance and to transform

the sequential program, based on the analyzed information about data dependence, into its corresponding parallelized or vectorized form. The transformation techniques can be incorporated into the program transformation module.

After program transformation, the transformed program is mapped and scheduled onto the given target machine using the *data distribution and program scheduling* module. The specifications of the target machine contain the topology, the number of processors, the startup time of message transmission, buffer size, the status of links, and so forth, of distributed memory multicomputers, shared memory multiprocessors, or supercomputers.

After the data distribution and program scheduling module process, the intermediate code and the parallelized or vectorized code are produced for different machines in the *code generation* module. The intermediate code is simulated in the *simulator of the target machine* module. The simulator plays two important roles here. First, the simulator evaluates the parallelized or vectorized code and monitor the target machine. It may be difficult to carry out program parallelizing and optimizing without evaluating the performance of the compiled code. Thus, the parallelizing compiler system performs parallelization based on evaluation of the compiled codes running on the target machine. Second, the simulator is a testbed for the development of this environment and a research tool for parallelizing techniques. The output of simulators includes the behavior records of each processor and statistical results. Based on the output results generated by the simulator of the target machine module, users can in advance predict whether the transformed program on the target machine can produce better performance or not. If the execution performance occurred from the transformed program is poor, users can interactively turn on the other transformation techniques and apply them to the original sequential program so that better execution performance can be generated.

For all information generated or accessed by modules, designing and implementing the *database* module is desirable for this environment and programmers. This information and data are stored in the main memory or on disk. In the main memory, there exist a symbol table, program representation, internal information about data dependence, loop restructuring information, and so on. On disk, there exist the source sequential program, parallelized or vectorized program, intermediate code, reported information about data dependence, simulation results, statistical results obtained after simulating the transformed program, a target machine description, and so on.

More detailed implementations and their corresponding complex data structures will be introduced in later sections for the parallelizing compiler on UPPER. In the next section, we will first describe the design techniques and various modules of the machine-independent phase of our parallelizing compiler.

## III. User Interface

In this section, we will describe the user-interface module, the bridge which enables users to communicate with this environment. By means of the user interface, users can easily edit and

compile the sequential source program. Furthermore, several graphs and tables derived from the analyzed results such as the program construction, the dependency relationship between data and control, execution performance evaluation after simulation of parallel programs, and so on, can be viewed by users. A snapshot of this environment is shown in Figure 2.

The main menu bar of this environment has six selection items: **File**, **View**, **Compile**, **Simulate**, **Options**, and **Help**. All the functions of this environment which have hierarchical structure are listed in Figure 3. For each function, some specifications are also depicted in Figure 3.

The **File** selection item supports the functions of *Open*, *Editor*, and *Exit*. At any time, users can open a file to be compiled by using the *Open* function key. When the *Exit* function key is chosen, all of the jobs stop, and the system halts. The environment also support an editor which can be used to edit a program and can be selected by using the *Editor* function key.

Within the **View** function, a data dependence graph can be shown by selecting the *dependence graph* selection subfunction. We can partition the sequential program into several segments by relating a nested loop to a segment of the program. Each program segment has its own data dependence graph. A snapshot of a data dependence graph is shown in Figure 4. The main window, titled the *Data Dependence Viewer*, is in the top-left corner of Figure 4. Each node of the graph represents a statement in a program segment. The arc between two nodes stands for the data dependence relationship. The number on the arc is the number of data dependences between two statements. In the menu bar, we can choose *Next* (*Previous*) to show the data dependence graph related to the next (previous) program segment or choose *Exit* to quit. The window titled *Information for Data Dependence Graph* is shown in the bottom-left corner. The dependence relations in the left hand table include forward data dependences. The dependence relations in the right hand table include backward data dependences. In each row of the table with five entries, there are numbers of data dependences associated with some arc. The first entry indicates the arc number. The last four entries indicate the number of true (flow) dependences, antidependences, input dependences, and output dependences, respectively. The window titled *Source Code* on the right-hand side shows the corresponding program segment and displays the line numbers on the leftside. The *data dependence information* is depicted in a table below the source code. Each row with four entries indicates a data dependence relation between two statements whose line numbers are the first two entries. That is, a statement with the second line number is data dependent on the statement with the first line number. The third entry is the variable name, and the last entry indicates the dependence vector or the dependence distance. For example, consider arc 1 shown in Figure 4. From the window titled *Information for Data Dependence Graph*, we know that there is a true dependence from the 26-*th* line to the 27-*th* line. From the *data dependence information*, we know that the 27-*th* line is true dependent on the 26-*th* line at the variable VA and that its dependence distance is $(0, 1, -2)$.

When setting the machine environment and compilation techniques, the user can select

**Compile** to compile the program. At the same time, the parallelized code will be shown beside the sequential code. The user can learn how the sequential code will be translated into parallelized code by comparing the parallelized code with the sequential code.

By **Simulate** selection item, the user can simulate execution of the parallelized code and simulation results can be displayed in the mean while. According to the various types of machine environments, the simulator will show different simulation results. For example, the simulation results for shared memory multiprocessors are shown in Figure 5 and those for distributed memory multicomputers are shown in Figure 6. In the simulation results, each node represents a processing element. Each processing element can be in an idle, working, or communication state. We use different colors to represent the state of the processing element to to let the user easily recognize the state of the processing element. The user can use the $Delay = 1$ button to speed up or use the $Delay = 10$ button to slow down the simulation. Using the *rerun* function, the user can rerun the simulation.

The **Options** selection item includes two subitems: *Set Machine Environment* and *Compilation Techniques*. After selecting a program, the user can set some aspects of the machine environment such as the topology of the target machine, the number of processors, and so on by choosing the *Set Machine Environment* subitem. In distributed memory multicomputers, the user can select *Linear*, *Ring*, *Mesh*, and *Torus Mesh* as the topology of the target machine. The user can also select the *Compilation Techniques* subitem to choose the compilation techniques. Presently, for distributed memory multicomputers, there exist three compilation techniques: *Communication-Free without Duplicate Data*, *Communication-Free with Duplicate Data*, and *Non-Communication-Free Transformation*, which have been described in detail in (Chen and Sheu, 1994) (Sheu and Chen, 1995).

The most important point is that the system supports a **Help** function. Within each menu, **Help** includes explanations of the selection items in the menu and can help the user use the system. The user can get help immediately when it is needed. From the user's point of view, this is the most friendly part of the environment.

# IV. Machine-Independent Phase of the Parallelizing Compiler

In this section, the machine-independent phase of our parallelizing compiler system is described. The implementation issues and internal data structures are illustrated using the following example.

**Example:** Consider the following TEST program "test.f".

```
PROGRAM TEST
INTEGER I, J
REAL A(10,10), B(10,10), C(10,10)
DO 10 I = 2, 5
   DO 20 J = 2, 5
```

```
      A(2*I,J) = C(I,J) * 7
      B(J,I+1) = A(2*I-2,J-1) + C(I-1,J-1)
   20 CONTINUE
10 CONTINUE
END
```

## 1. Preprocessing and Dependence Analysis

In this subsection, the preprocessing and dependence analysis processes for a given sequential program are discussed. We use *lex* and *yacc*, respectively, as tools to scan and parse the sequential programs. After scanning and parsing the sequential program within the *preprocessing* module, the *symbol table* and its *program representation* are constructed for easy manipulation of subsequent modules.

For each declared variable established in the *symbol table*, its symbol table entry has the following fields. *Declaration type* is a flag to indicate that this variable is declared to be either INTEGER or REAL. *Variable name* is a string to indicate the variable name. *Dimension* indicates the array dimension; if *dimension* is zero, this means that the variable is a scalar variable. *Declaration bounds of arrays* indicates the user-defined bounds of each array dimension. For example, there are two scalar variables, I and J, with INTEGER type and three two-dimensional array variables, A, B, and C, with REAL type in the TEST program. The ranges of these three array variables in each dimension are declared from 1 to 10.

The *program representation* of a given program can not only preserve the meaning of the original semantics but also indicate the control flow with DO and structured IF statements, assignment statements and operations, and the relationship between the program and symbols along with other information. There are three construction types of basic blocks within a program representation: IF statements, DO statements, and other statements (assignment statement, function call, and procedure call) whose graph constructions, regarded as IF nodes, DO nodes, and statement list nodes, are, respectively, shown in Figure 7(a), (b), and (c).

To clearly demonstrate the concept of program representation, the following segmentation code with complex structures is given. Its corresponding program representation is depicted in Figure 8.

```
S1
S2
DO 10 I = L, U
   IF B THEN
      S3
   ELSE
      S4
      S5
   ENDIF
   S6
10 CONTINUE
S7
```

Within a given program, basic units consisting of the three basic blocks are assignment statements, procedure calls, and function calls. The basic units consisting of an assignment statement, a procedure call, or a function call are expressions. However, the smaller basic units consisting of an expression are operators and operands. Detailed descriptions will be given below. An assignment statement consists of two expression: the left one is a write operand, and the right one consists of several operands and operators. A procedure call or function call is composed of its name and several arguments and is also expressed by an expression or a function call. An operand may be a scalar variable or an array variable composed of its array name and several subscripts, which are also expressed by an expression or a function call. Hence the entire sequential program can be recursively constructed and represented by the three basic block constructions and the small constructions described above. The program representation of the TEST program is shown in Figure 9 based on the above descriptions.

After preprocessing an input program, each loop can be classified into one of the following four types. The ALLDOALL type indicates that there exists no dependence in this loop. The UNIFORMLY_NESTED type indicates a nested loop with uniformly generated references (Gannon et al., 1988). The STAND_NESTED type indicates a nested loop with constant data dependence. Other loops are classified in OTHERS. In the *dependence analysis* module, applying the GCD test and Banerjee-Wolfe test produces the data dependence information, including dependence or independence, the dependence type with input, output, flow dependence and antidependence, and the direction or distance vectors for each pair of variables. Within the dependence analysis module, the Banerjee-Wolfe test which extends the Banerjee's inequalities to find the dependence distance or direction vector (Wolfe, 1992) is also implemented when the loop limits are triangular, meaning that the limits of the inner loop depend on the outer loop indices. In addition, we also extend the work to manipulate more complex non-perfect loops. Consider the TEST program again. There only exists one nested loop, identified as the UNIFORMLY_NESTED type. The data dependence information is generated in two files with the filenames "test.dep" and "test.var".

The format of each line in a produced file with data dependence information is specified as follows:

$$LT \;\; Loop^{th} \;\; line_s \;\; line_e \;\; Var^{th} \;\; \underbrace{i \;\; line_i \;\; j \;\; line_j}_{VarPair} \;\; DT \;\; Flag \;\; n \;\; \underbrace{d_1 \;\; d_2 \;\; \cdots \;\; d_n}_{\bar{d}}.$$

The description and definition of each of the above terms are described below. The symbol $LT$ denotes one of the loop types classified above. $Loop^{th}$ denotes the loop number arranged in a given program, automatically produced by our compiler. The terms $line_{start}$ and $line_{end}$, respectively, denote the start and end line numbers within the given source program. $Var^{th}$ denotes the variable number in a given program, produced by our compiler. $VarPair$ indicates the variable pair which is tested using dependence tests. For the first variable, $i$ denotes

the number of $Var^{th}$ which is stored in the $i$-th variable of the file whose filename has the extension "var". The line number of $i$-th $Var^{th}$ appearing in the source file is denoted as $line_i$. For the second variable, $j$ denotes the number of $Var^{th}$. $line_j$ denotes the line number of $j$-th $Var^{th}$. The symbol $DT$ represents the type of data dependence, whether flow, input, output dependence, and antidependence. The term $Flag$ is a flag to indicate either the dependence vector or dependence direction for this data dependence relation. The dimension of this dependence vector or direction is denoted by $n$. $\bar{d}$ with $n$-tuple is either a dependence vector or direction depending on the flag $Flag$. If $\bar{d}$ is a direction vector, each $d_i$, $1 \leq i \leq n$, is one value depending on its direction specified within Table 1. Hence, a data dependence viewer is designed in this system to establish the data dependence graph of each loop and then to display it to programmers based on all of the information in these two files. To illustrate, a snapshot of our developed data dependence viewer with a source loop, its data dependence graph, and its information concerning data dependence is shown in Figure 4.

## 2. Program Transformation

By using the generated information about data dependence, we can apply to a program various compilation techniques integrated into the *program transformation* module so as to produce a parallelized or vectorized code. Within the program transformation module, we implement the submodules of *program parallelization*, *data parallelization*, and *data and program parallelization*. Within the program parallelization submodule, a compilation technique aimed at partitioning for linear array multicomputers has been designed (Sheu and Chen, 1995). Within the data parallelization submodule, a compilation technique has been designed, aimed at communication-free partitioning without duplicate data during parallel execution (Chen and Sheu, 1994). Within the data and program parallelization submodule, a compilation technique has been designed, aimed at communication-free partitioning with duplicate data during parallel execution (Chen and Sheu, 1994). The three compilation techniques were originally proposed and designed on distributed memory multicomputers to reduce the communication overhead. However, they can be also applied to shared memory multiprocessors so as to eliminate as much as possible cache or local memory thrashing (Lu and Fang, 1992).

Loops are the most time-consuming parts and implicitly provide a large amount of parallelism in a program. Therefore, we currently only consider loop transformations within the program transformation module. While a program is processed through the program transformation module, a DO loop can be translated into one of three types: DOSER, DOALL, and DOACR. The DOSER type, which is not changed in the original program, means that this loop via transformation is still performed sequentially. The DOALL type means that each iteration of this loop via transformation is independently performed in parallel. The DOACR type means that this loop can be performed in parallel but still needs communication or synchronization primitives to keep the relationship of data dependence and preserve

the semantics of the original program.

For the program transformation module, an example shown below is given to illustrate the designed flow and the change of internal structures depending on different compilation methods. Through the compilation techniques of communication-free partitioning with or without duplicate data, the DO loop within the TEST program is translated into the following program segment with a parallel construct, DOALL, written in the form of FORTRAN:

```
DOALL 10 I' = −3, 3
      DO 20 I = MAX(2, I'+2), MIN(5, I'+5)
         J = I − I'
         A(2∗I,J) = C(I,J) ∗ 7
         B(J,I+1) = A(2∗I−2,J−1) + C(I−1,J−1)
      20 CONTINUE
10 CONTINUE
```

Within our parallelizing compiler, both the tree structure (program representation) and the symbol table are the heart or kernel. This is because all the information such as the original program semantics, the translated program representation, the mapped and scheduled program representation, etc., are included for the process of each module. While applying any analysis or compilation technique, the tree structure and internal structure of the symbol table are adjusted. Each adjustment may cause a drop, insertion, or movement of internal structures, for example, movement of basic blocks, insertion of new basic blocks, modification of expressions, or insertion of new symbols. Therefore, the corresponding program representation has to be modified and translated into another tree structure. By means of the above program segmentation, an additional symbol, I', must be incorporated into the symbol table. Two induction variables, I and J, of DO constructs are changed to the new variable I' and the original variable I, respectively. Their loop lower bounds and upper bounds are also modified. An additional statement is appended to the original loop body. Hence, the program representation is translated and is depicted in Figure 10. Depending on the different compilation techniques within the program transformation module, the tree structure (program representation) is, therefore, modified and changed to form another tree structure.

After transformation of a program, the most important work is scheduling of tasks for the target machine according to the type of parallel machine, the architecture, and the number of processors. In the next section, we will describe the implementation techniques and various modules of the machine-dependent phase of this parallelizing compiler.

# V. Machine-Dependent Phase of the Parallelizing Compiler

In this section, the implementation of each module within the machine-dependent phase of our parallelizing compiler is described, with respect to the machine architecture, topology, and size.

## 1. Data Distribution and Program Scheduling

The *data distribution and program scheduling* module is described in this subsection. For parallel machines, data distribution, program partitioning and scheduling significantly determine the execution behaviors and performance.

Now, we will discuss the approaches to partitioning on parallel computers. Because we currently only consider the topology, *mesh*, for distributed memory multicomputers, program scheduling for the communication-free partitioning and projection methods is simple. The complex and optimal assignment algorithms were presented in (Chen and Sheu, 1994) (Sheu and Chen, 1995). Generally speaking, the purpose of these strategies is to eliminate or reduce as much as possible interprocessor communication. The communication-free data allocation technique can totally eliminate interprocessor communication. Another strategy can reduce interprocessor communication by allocating necessary data to the location where it is used or involve only neighbor-to-neighbor communication. Hence, the methods we use can not only reduce the communication overhead on distributed memory multicomputers, but also increase the data locality and cache hit ratio on shared memory multiprocessors. Thus, these methods are suitable for the two categories of parallel computers, distributed memory multicomputers and shared memory multiprocessors. It should be pointed out that, during program scheduling on shared memory multiprocessors, we adapt static partitioning.

When the process of compiling a sequential program into a parallel form, maintained and represented in the program representation, has been completed, the program's parallel intermediate form will finally be produced by the code generation module.

## 2. Code Generation

In this subsection, the *code generation* module for simulators of distributed and shared memory multiprocessors is described. By means of the intermediate parallel form, users can easily understand the power of parallelism extraction and the capability of program transformation.

By using the program representation of a transformed program, we can translate the transformed program into an intermediate code written in the C language. In addition to the constructs supported by the original C language, we integrate the parallel constructs and synchronization primitives shown in Table 2 into our specified intermediate code for shared memory multiprocessors. Table 3 shows a list of supported message-passing functions in the C language for distributed memory multicomputers.

For more details, readers can refer to several examples in references (Chu, 1993), (Ni, 1993). In the following section, the simulators of parallel computers which simulate the above mentioned intermediate parallel forms will be introduced.

# VI. Simulator

In this section, the *simulator of the target machine* module for evaluating and measuring performance during execution of a parallel program is specified.

A simulator of shared memory multiprocessors presented in (Chu, 1993) is first discussed. In this simulator, we integrate several parallel constructs and synchronization primitives depicted in Table 2 into the DLXsim (Hennessy and Patterson, 1990). The DLXsim is a simulator for DLX which has a theoretical load/store architecture and is derived from RISC architecture. There exists a C compiler supported by DLX for compiling a given program with appropriate parallel constructs and synchronization primitives into DLX assembly code. After simulating a given intermediate code of shared memory multiprocessors, the executed results include total execution cycles, processor utilization, the amount and cycle time of communication, etc.

The framework of the simulator of shared memory multiprocessors is described below. In designing it, many data structures are needed. The most important one is a tree which is used to represent the relationship among processors and to capture the information and status while running a program. We model the execution flow of the program coded in the intermediate form for our simulator as follows. Each node in the tree indicates one processor. At the beginning, only one processor executes the program, so it is modeled as the root node in the tree. Once the processor deals with a DOALL$(L, U, S)$ (or DOACR$(L, U, S)$) construct, the statements between DOALL$(L, U, S)$ and ENDDOALL() (or DOACR$(L, U, S)$ and ENDDOACR()) will be executed in parallel by $\lceil (U - L + 1)/S \rceil$ processors. Hence, there $\lceil (U - L + 1)/S \rceil$ nodes are generated as the children nodes of the root node. Then, these processors begin to execute the statements between the two parallel constructs while the parent node plays two roles; one is the root node and another is then one of the children. Thus, each edge in the tree indicates the relation between two index instances (processors) in the two contiguous parallel loop constructs (DOALL or DOACR). While processors execute the communication primitives, some information and the status up to that point including the total execution cycle time, the amount of synchronization, the synchronous cycle time, and so on, have to be recorded. If one parallel construct is met, the actions described above will be recursively applied to establish the tree. The detailed implementation can be found in (Chu, 1993). A snapshot of the simulation results is shown in Figure 5. A shared memory multiprocessor with 8 processors and the statistical results are shown in the table and in a graphic manner.

Next, a simulator of distributed memory multicomputers presented in (Ni, 1993) will be discussed. Because the simulator we have developed can simulate torus mesh architectures, all of the mesh, linear array, and ring topologies can be simulated. The overall schema of our simulator is illustrated in Figure 11. An oblong shape represents a data structure or a storage unit that keeps a particular set of data. A rectangle represents a function, action, submodule, or manipulation which executes some sort of operation on a related data structure (oblongs).

The rectangles with shadows represent software modules of the simulator. An arrow indicates the data flow and/or control flow among the modules of the data structures.

The primary input of this simulator is a set of programs running on each PE. The assembly source programs are assembled by the assembler in the *preprocessing module*. After that, the object codes are pre-coded and linked to the message-passing supporting library to generate the simulator-executable object codes. The *object code loader* loads the executable codes into the *object* module (PE objects) and generates initial events for the *event-driven engine*. The object and event-driven engine modules perform the actual simulation tasks. The event-driven engine maintains the ordered event list data structure, from which the module picks out the current event for the object module and the output generation module. The event-driven engine also maintains a synchronizing object list which keeps a list of objects that need to be synchronized. When the object module receives a current event, it simulates the actions of the activated objects. The states of objects are updated. New events are generated and sent to the event-driven engine, where the new events are scheduled for future simulation.

In addition to the object module, the current events (behaviors) are sent to the output generation module. The current event is filtered by the *event filter* of the output generation module to produce behavior records of the interested behaviors of the simulated system. The *object statistic extractor* generates statistical results from objects after the simulation is completed. The detailed implementation can be found in (Ni, 1993). A snapshot of a simulation of a matrix multiplication program on a $4 \times 4$ torus is shown in Figure 6.

Our designed simulators not only simulate a parallelized code, but also evaluate and measure its execution performance. If the performance is not acceptable, the user can modify the source program or apply other compilation techniques to produce more efficient parallelized code.

## VII. Conclusions

We have described the implementation and design issues of the parallel programming environment **UPPER**, including the user-interface, the parallelizing compiler system with machine-independent and machine-dependent phases, and the simulator. Playing the most important role of communication between users and the environment is the user-interface. The machine-independent phase of the parallelizing compiler system deals with the preprocessing of a source program, dependence analysis, and our proposed compilation strategies. During the machine-dependent phase, we first deal with the mapping and the scheduling of a transformed program onto the target machine specified by the user. Then, the simulators of shared memory multiprocessors and distributed memory multicomputers measure the execution performance of a resultant parallelized code. This environment enables programmers to easily design parallelized programs by means of interaction between the programmer and this system.

In order to implement our parallel programming environment, the following approaches

to compilation and design aspects will be considered and adapted in the future. The first goal will be to consider interprocedural analysis (Burke and Cytron, 1986) (Triolet et al., 1986) (Li and Yew, 1988). We will construct a call graph to directly explore the parallelism of procedure calls or functional parallelism in order to extract a large amount of parallelism in a program. Second, we will consider the model of several nested loops together in a program. For multiprocessor systems, we will design an approach which minimize parallel execution time by analyzing data dependence and determining data layout. Third, due to the need to manage data and information, the design of an efficient database system will become our focus. Finally, we will improve and enhance the applicability of user interface by adding a new graphical demonstration system and visualization system, and we will then integrate each of these future works into UPPER so that it will have powerful compiling capability.

# References


Allen, F., Burke, M., Charles, P., Cytron, R., and Ferrante, J. (1988). An overview of the PTRAN analysis system for multiprocessing. *Journal of Parallel and Distributed Computing*, 5(5):617–640.

Amarasinghe, S. P. and Lam, M. S. (1993). Communication optimization and code generation for distributed memory machines. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation*, pages 126–138.

Anderson, J. M. and Lam, M. S. (1993). Global optimizations for parallelism and locality on scalable parallel machines. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation*, pages 112–125.

Banerjee, U. (1988). *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, Norwell, Massachusetts.

Burke, M. and Cytron, R. (1986). Interprocedural dependence analysis and parallelization. In *Proceedings of the SIGPLAN'86 Symposium on Compiler Construction*, pages 162–175.

Chen, T. S. and Sheu, J. P. (1994). Communication-free data allocation techniques for parallelizing compilers on multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 5(9):924–938.

Chu, C. C. (1993). A simulator of shared-memory multiprocessor systems for parallelizing compilers. Master's thesis, Institute of Computer Science and Electrical Engineering, National Central University, Taiwan, R.O.C.

Ferrante, J., Ottenstein, K. J., and Warren, J. (1987). The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349.

Gannon, D., Jalby, W., and Gallivan, J. (1988). Strategies for cache and local memory management by global program transformations. *Journal of Parallel and Distributed Computing*, 5(5):587–616.

Gupta, M. and Banerjee, P. (1992). Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):179–193.

Hennessy, J. L. and Patterson, D. A. (1990). *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc.

Hiranandani, S., Kennedy, K., and Tseng, C. W. (1992). Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80.

Kennedy, K., McKinley, K. S., and Tseng, C. W. (1991). Interactive parallel programming using the ParaScope editor. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):329–341.

Koelbel, C. and Mehrotra, P. (1991). Compiling global name-space parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):440–451.

Koelbel, C., Mehrotra, P., and Rosendale, J. V. (1990). Supporting shared data structures on distributed memory architectures. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 177–186.

Li, J. and Chen, M. (1991). Compiling communication-efficient programs for massively parallel machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):361–376.

Li, Z. and Yew, P. C. (1988). Efficient interprocedural analysis for program parallelization and restructuring. In *Proceedings of ACM SIGPLAN'88 Parallel Programming: Experience with Applications, Languages and Systems*, pages 85–99.

Li, Z., Yew, P. C., and Zhu, C. Q. (1990). An efficient data dependence analysis for parallelizing compilers. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):26–34.

Lu, M. and Fang, J. Z. (1992). A solution of the cache ping-pong problem in multiprocessor systems. *Journal of Parallel and Distributed Computing*, 16:158–171.

Ni, S. Y. (1993). A simulator of distributed-memory multicomputers for parallelizing compilers. Master's thesis, Institute of Computer Science and Electrical Engineering, National Central University, Taiwan, R.O.C.

Padua, D. A. and Wolfe, M. J. (1986). Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29:1184–1201.

Polychronopoulos, C. D., Girkar, M., Haghighat, M. R., Lee, C. L., Leung, B., and Schouten, D. (1989). Parafrase-2: An environment for parallelizing, partitioning, synchronizing, and scheduling programs on multiprocessors. In *Proceedings of International Conference on Parallel Processing*, volume II, pages 39–48.

Pugh, W. (1992). A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):102–114.

Rogers, A. and Pingali, K. (1989). Process decomposition through locality of reference. In *Proceedings of the ACM SIGPLAN'89 Conference on Programming Language Design and Implementation*, pages 69–80.

Sheu, J. P. and Chen, T. S. (1995). Partitioning and mapping of nested loops for linear array multicomputers. *The Journal of Supercomputing*, 9:183–202.

Su, E., Palermo, D. J., and Banerjee, P. (1993). Automating parallelization of regular computations for distributed-memory multicomputers in the PARADIGM compiler. In *Proceedings of International Conference on Parallel Processing*, volume II, pages 30–38.

Tjiang, S., Wolf, M., Lam, M., Pieper, K., and Hennessy, J. (1992). Integrating scalar optimizations and parallelization. In *Languages and Compilers for Parallel Computing*, pages 137–151.

Triolet, R., Irigoin, F., and Feautrier, P. (1986). Direct parallelization of call statements. In *Proceedings of the SIGPLAN'86 Symposium on Compiler Construction*, pages 176–185.

Wolf, M. E. and Lam, M. S. (1991). A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471.

Wolfe, M. J. (1989). *Optimizing Supercompilers for Supercomputers.* London and Cambridge, MA: Pitman and the MIT Press.

Wolfe, M. J. (1991). The Tiny loop restructuring research tool. In *Proceedings of International Conference on Parallel Processing*, volume II, pages 46–53.

Wolfe, M. J. (1992). Triangular Banerjee's inequalities with directions. Technical report, Department of Computer Science and Engineering, Oregon Graduate Institute.

Wolfe, M. J. and Banerjee, U. (1987). Data dependence and its application to parallel processing. *International Journal of Parallel Programming*, 16(2):137–178.

Wolfe, M. J. and Tseng, C. W. (1992). The power test for data dependence. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):591–601.

Zima, H. P., Bast, H.-J., and Gerndt, M. (1988). SUPERB: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6(1):1–18.

Zima, H. P. and Chapman, B. (1991). *Supercompilers for Parallel and Vector Computers*. ACM Press, New York.

# Appendix A

In this Appendix, we list all of the grammars written in the Backus-Naur Form (BNF) which we have used.

```
<program-start>   : <program-front> <statement-list> <program-end> | ϵ
<program-front>   : <newline0> "PROGRAM" ID
<program-end>     : "END" <newline0>
<newline0>        : <newline0> NLINE | ϵ
<statement-list>  : <statement-list> NLINE <statement> | <statement>
<statement>       : <var-declaration> | <do-stmt> | <if-stmt>
                    | <assign-stmt> | ϵ
<var-declaration> : "INTEGER" <var1> | "REAL" <var1>
<var1>            : <var1> "," <var2> | <var2>
<var2>            : ID | ID "(" <exp> <exp1> ")"
<do-stmt>         : "DO" INTLIT <var> "=" <exp> "," <exp> <step>
                    <statement-list> INTLIT "CONTINUE"
<step>            : "," <exp> | ϵ
<if-stmt>         : "IF" <exp> "THEN" <statement-list> <else-part>
                    "ENDIF"
<else-part>       : "ELSE" <statement-list> | ϵ
<assign-stmt>     : <var> "=" <exp> | <array-var> "=" <exp>
<exp>             : <exp> ".EQ." <exp>
                    | <exp> ".LT." <exp>
                    | <exp> ".LE." <exp>
                    | ".NOT." <exp>
                    | <exp> ".GE." <exp>
                    | <exp> ".GT." <exp>
                    | <exp> ".AND." <exp>
                    | <exp> ".OR." <exp>
                    | "(" <exp> ")"
                    | <exp> "+" <exp>
                    | <exp> "−" <exp>
                    | "−" <exp>
                    | <exp> "*" <exp>
                    | <exp> "/" <exp>
                    | <exp> "**" <exp>
                    | <exp> "MOD" <exp>
                    | "MAX" "(" <exp> <exp2> ")"
                    | "MIN" "(" <exp> <exp2> ")"
                    | "SQRT" "(" <exp> ")"
                    | "FLOOR" "(" <exp> ")"
                    | "CEILING" "(" <exp> ")"
                    | <array-var>
                    | <var>
                    | INTLIT
                    | FLOATLIT
<array-var>       : <var> "(" <exp> <exp1> ")"
<var>             : ID
<exp1>            : <exp1> "," <exp> | ϵ
```

<exp2> : <exp2> ”,” <exp> | $\epsilon$

Note that the lower-case words between the symbols < and > are regarded as nonterminal symbols; the upper-case words and the symbols between the symbols ” and ” are regarded as terminal symbols.

| | |
|---|---|
| ID | : A variable name with a legal string. |
| NLINE | : A new line character. |
| INTLIT | : An integer number. |
| FLOATLIT | : A floating point number. |
| $\epsilon$ | : An empty symbol. |

Table 1: The direction and its corresponding value of data dependence in each dimension.

| direction | $<$ | $=$ | $>$ | $\leq$ | $\geq$ | $\neq$ | $*$ |
|---|---|---|---|---|---|---|---|
| value | 0 | 1 | 2 | 3 | 4 | 5 | 6 |

Table 2: The parallel constructs and synchronization primitives supported in the parallel intermediate code of shared memory multiprocessors.

| Parallel Constructs and Synchronization Primitives | The Description of Each Parallel Construct and Synchronization Primitive. |
|---|---|
| ParBegin()<br>ParEnd() | The respective prologue and epilogue of a loop which is to be performed in parallel. |
| DOALL($L^a, U^b, S^c$)<br>ENDDOALL() | The respective prologue and epilogue of a loop without data dependence. |
| DOACR($L, U, S$)<br>ENDDOACR() | The respective prologue and epilogue of a loop with data dependence. |
| WAIT($from\text{-}node$) | While a processor performs this statement, it must wait for a signal from the $from\text{-}node$ processor within DOACR loops. |
| SIGNAL($to\text{-}node$) | While a processor performs this statement, it must send a signal to the $to\text{-}node$ processor within DOACR loops. |
| ENTRY()<br>EXIT() | The entry and exit of a critical section within DOACR loops. |
| BARRIER() | None of processors can perform the following statements until this statement within DOALL loops is performed. |

[a] An expression of the lower bound in this loop.
[b] An expression of the upper bound in this loop.
[c] An expression of the step in this loop.

Table 3: A list of the message-passing functions supported in the parallel intermediate code of distributed memory multicomputers.

| Function Names and Arguments | The Descriptions of Function Results |
|---|---|
| GetNodeInfo(&r, &c, &mr, &mc) | After calling, r and c contain the respective row and column number of this PE. mr and mc contain the respective total numbers of rows and columns in this mesh. |
| CircuitStartup(r, c) | Perform circuit routing from this PE to the PE at (r, c). The circuit will be built from this PE to the PE at (r, c). |
| Send(r, c, s, p, t) | Send a message with size s at the address p to PE (r, c). t is the type of message used as a message ID. |
| SendNC(r, c, s, p, t) | The action of Send(), but the circuit will be not kept in the cache after the message is sent. |
| SendDirection(d, s, p, t) | Send the message through link d. Links 0, 1, 2, and 3 are connected to the right, up, left, and down neighboring PE, respectively. |
| Receive(&r, &c, &s, p, &t) | Receive a message and place it at address p. If this message can be accepted, the values of r, c, and t compared to the incoming message should be the same. |
| ReceiveNB(&r, &c, &s, p, &t) | Non-blocked version of Receive(). If the function returns a non-zero value, a matched message has been received. |
| Broadcast(s, p, t) | Broadcast the message at address p whose size is s and type is t to all of other PEs. |

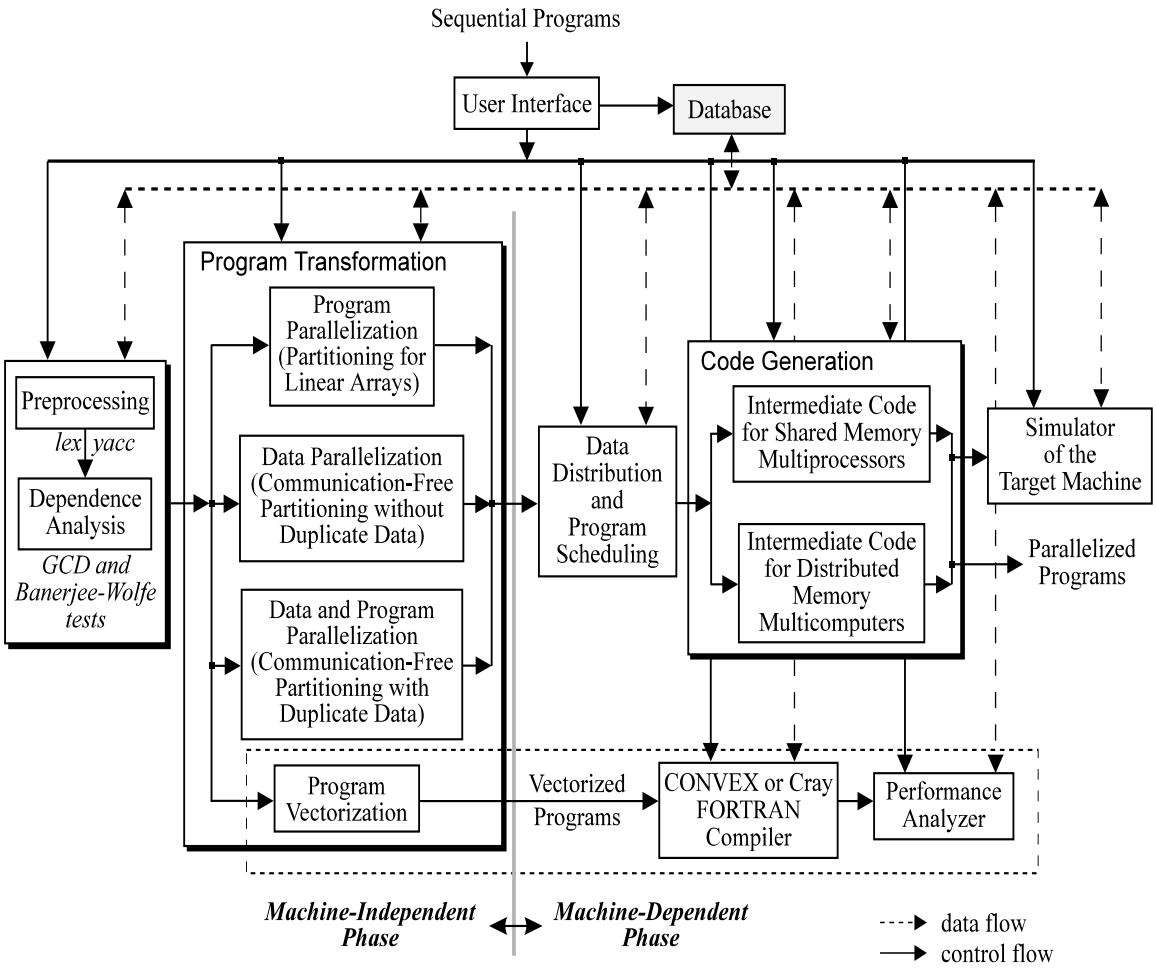



Figure 1: The configuration of the parallel programming environment.

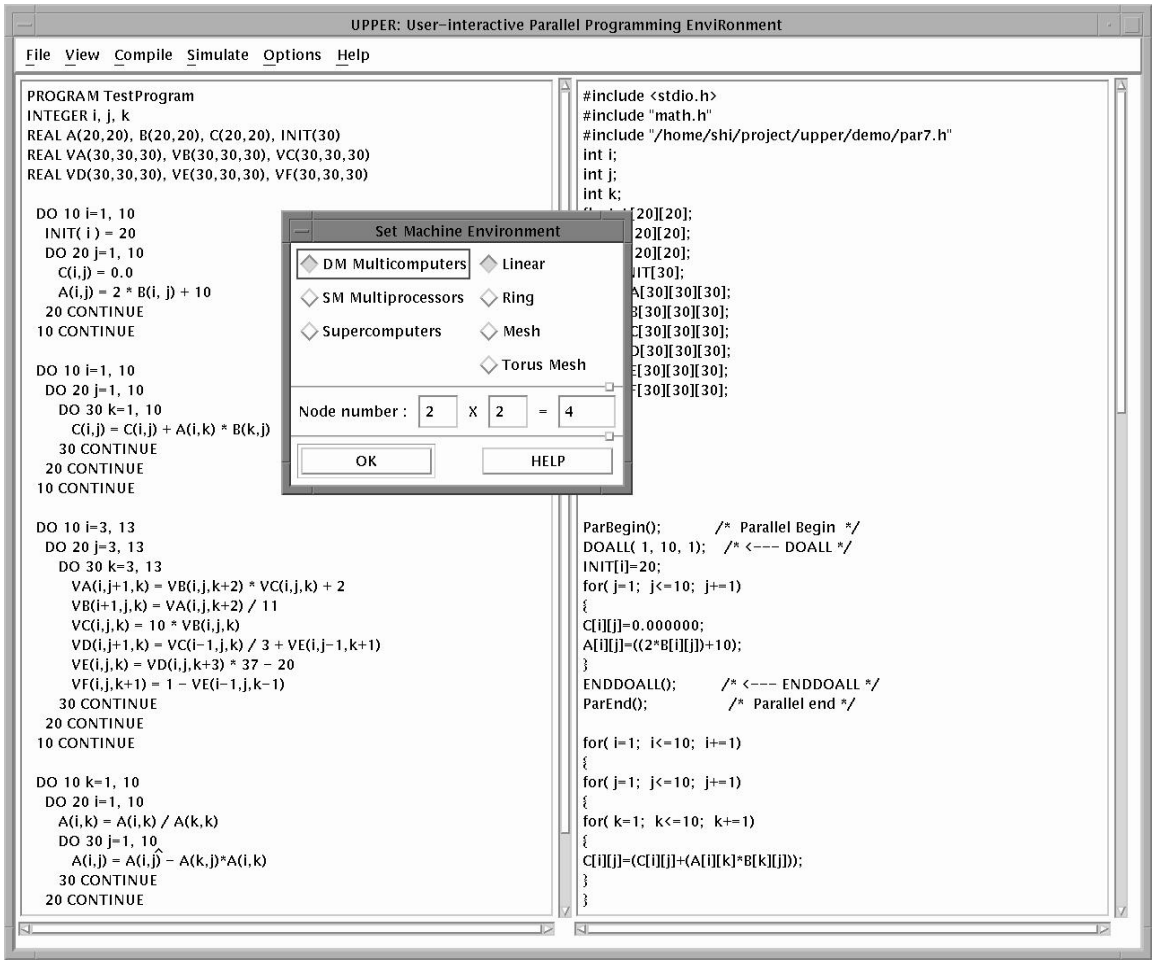



Figure 2: A snapshot of the parallel programming environment.

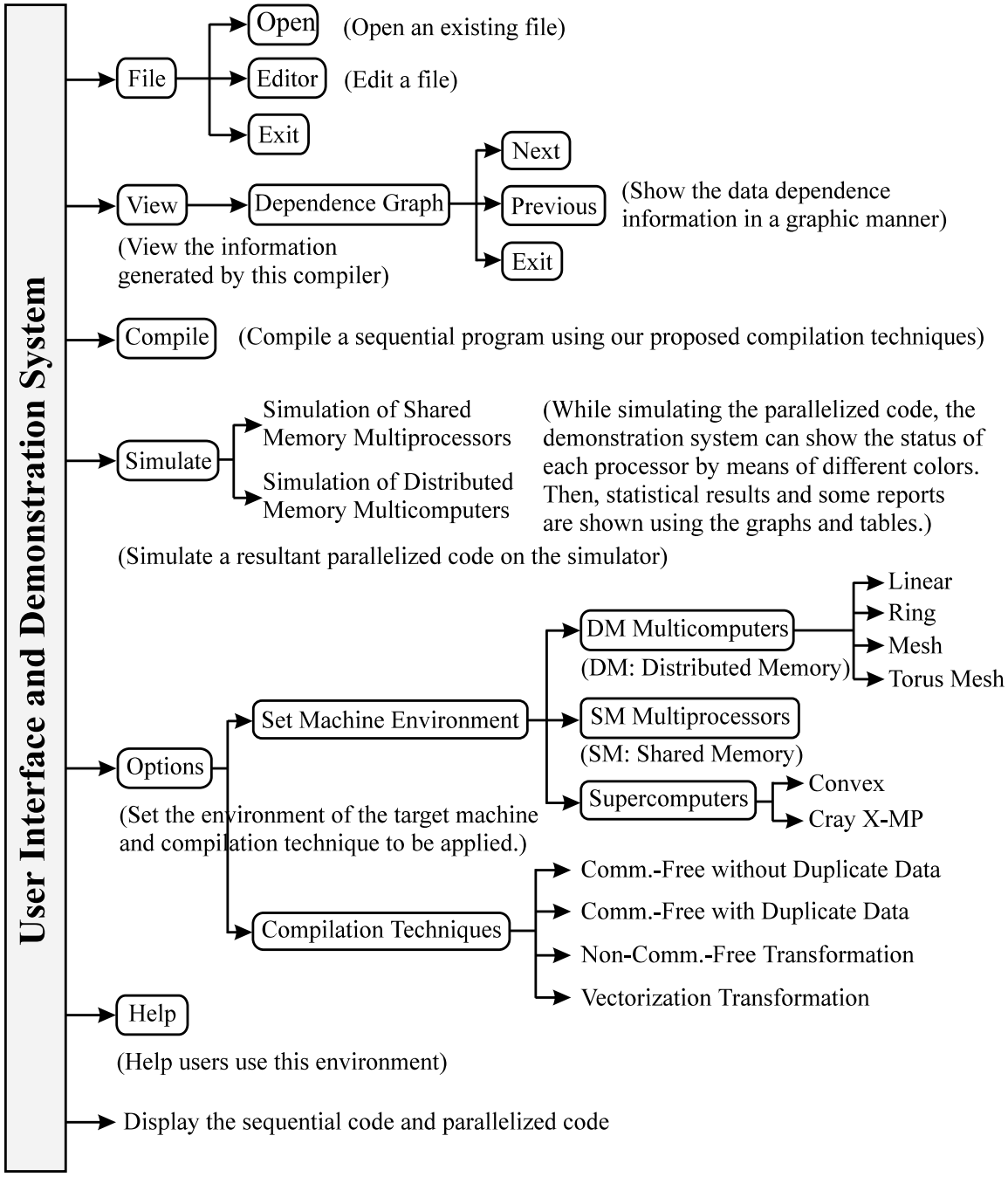



Figure 3: A list of all the functions in the parallel programming environment.

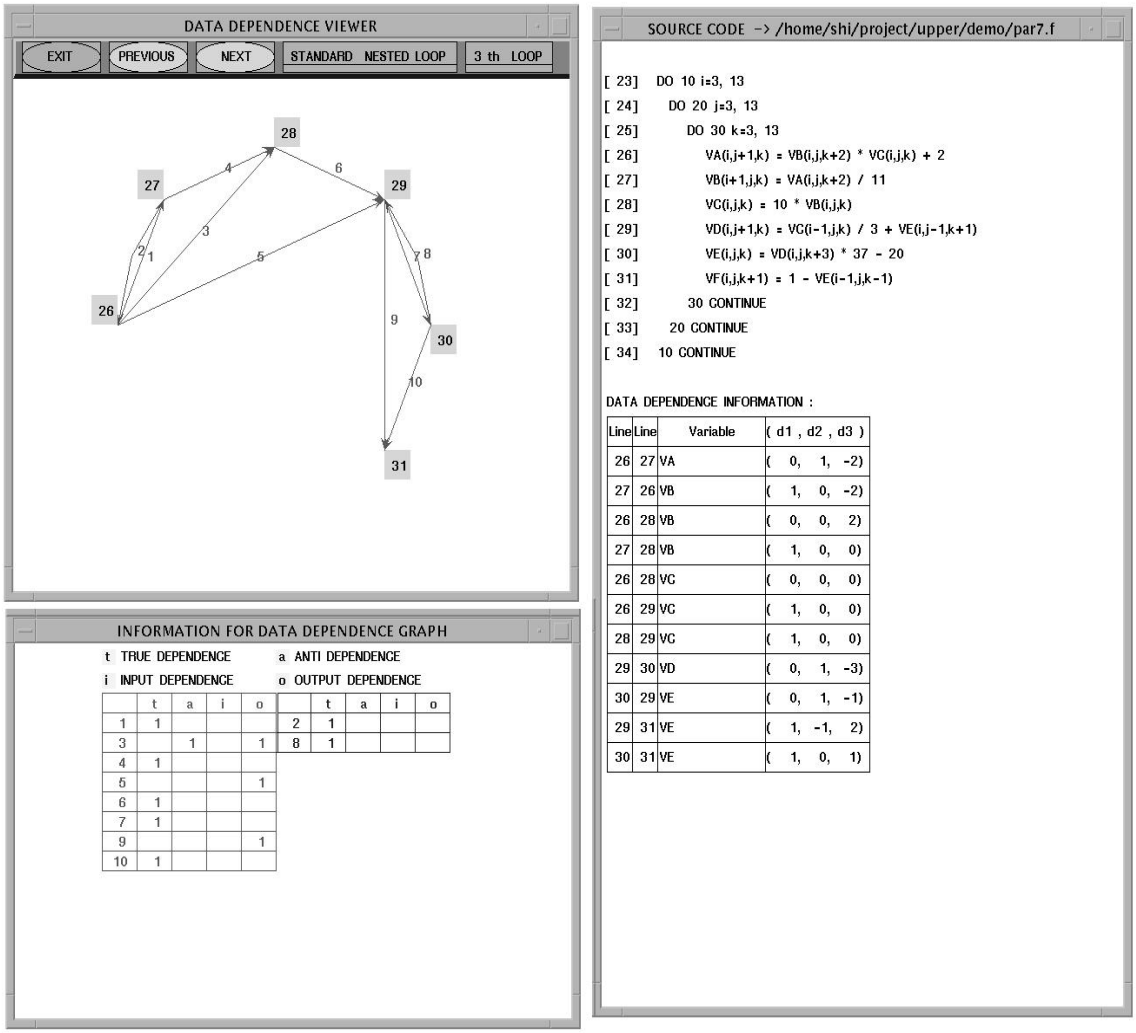



Figure 4: A snapshot of our developed data dependence viewer.

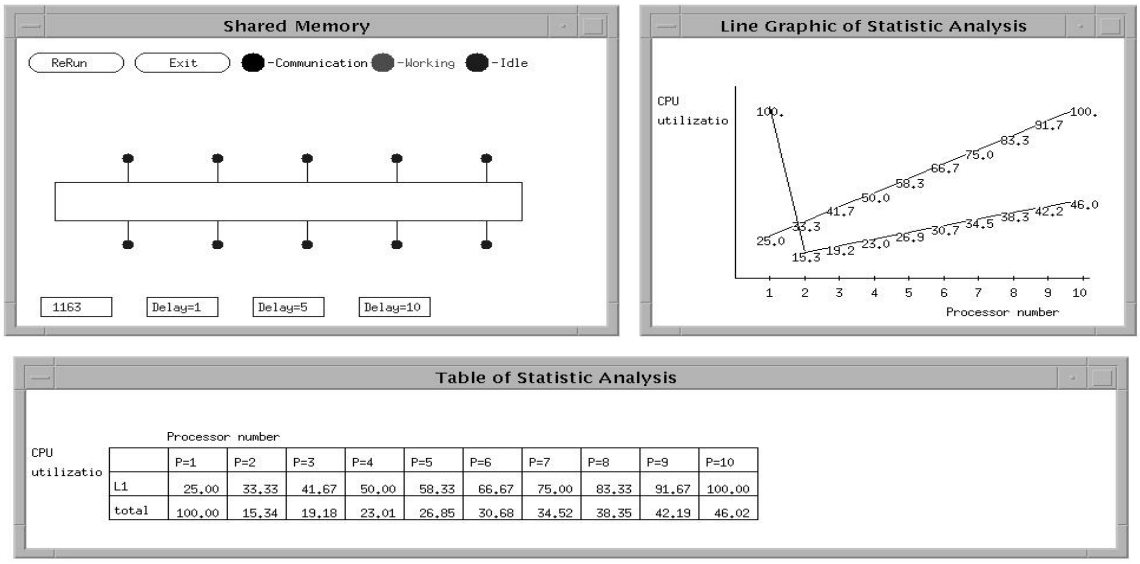



Figure 5: A snapshot of the simulation results of the shared memory multiprocessor simulator.

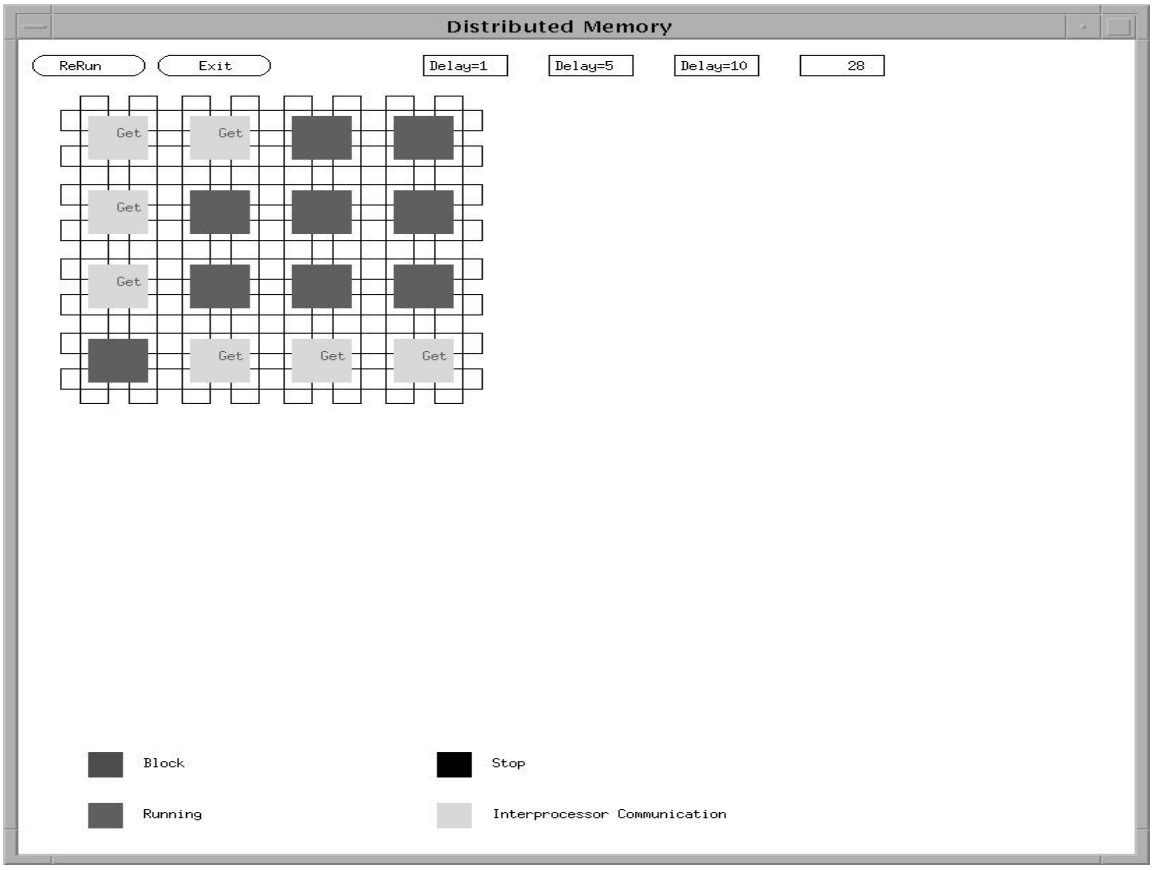
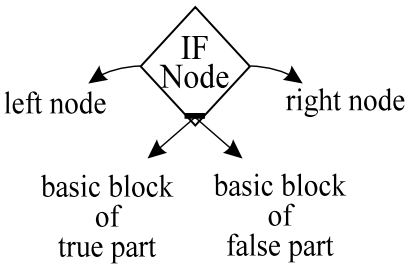
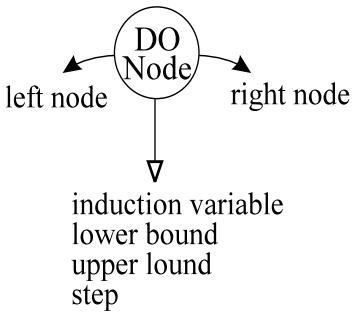


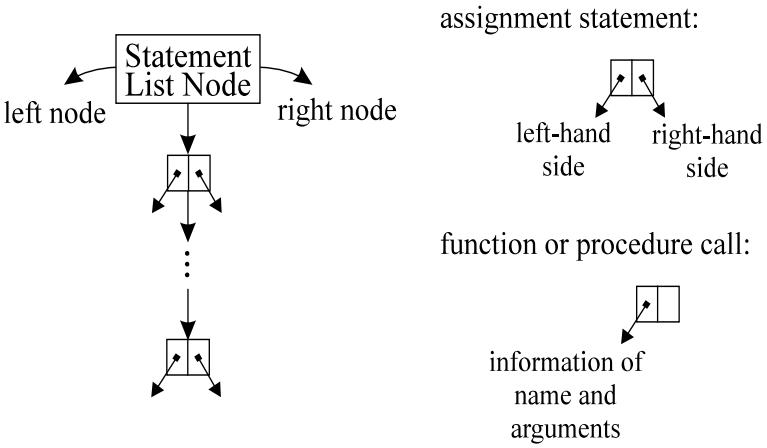

Figure 6: A snapshot of the simulation results of the distributed memory multicomputer simulator.




(a) IF node construction.




(b) DO node construction.




(c) Statement list node construction.

Figure 7: Three types of constructions of basic blocks.

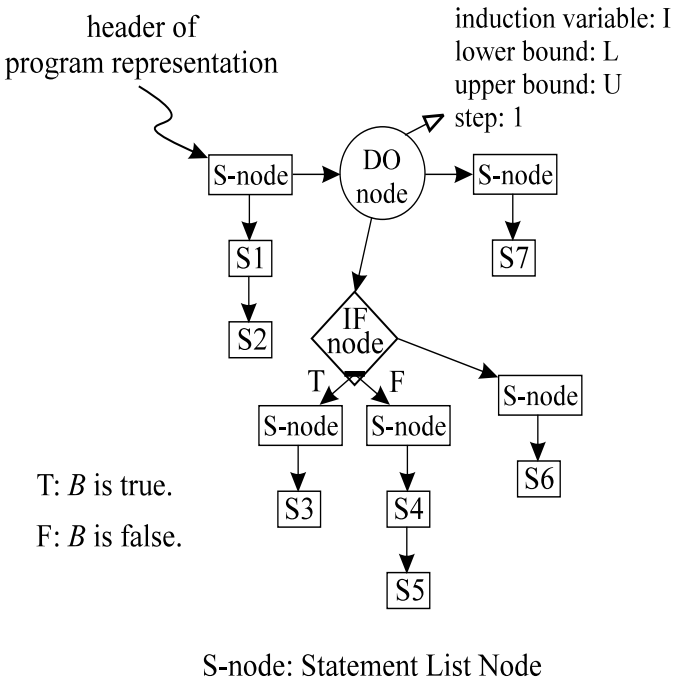



Figure 8: The program representation of the given segmentation code.

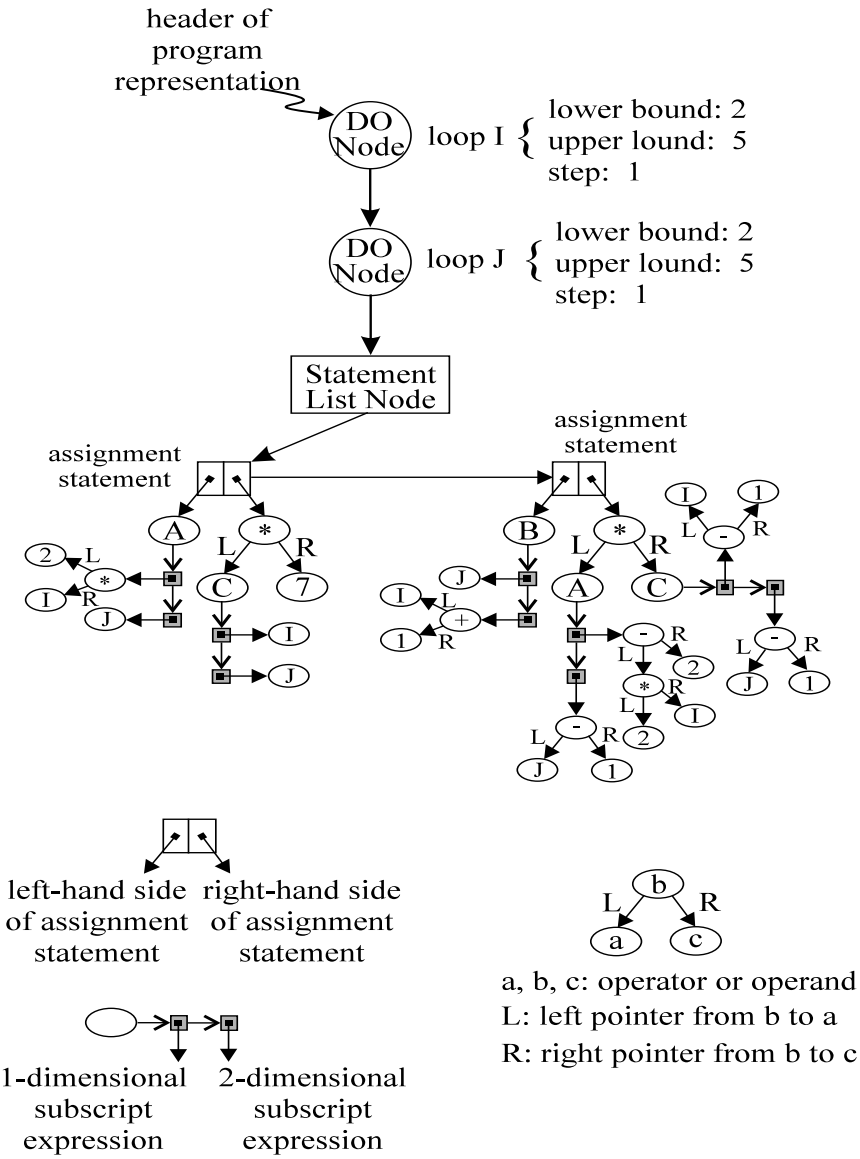



Figure 9: The graph view of the program representation of the TEST program.

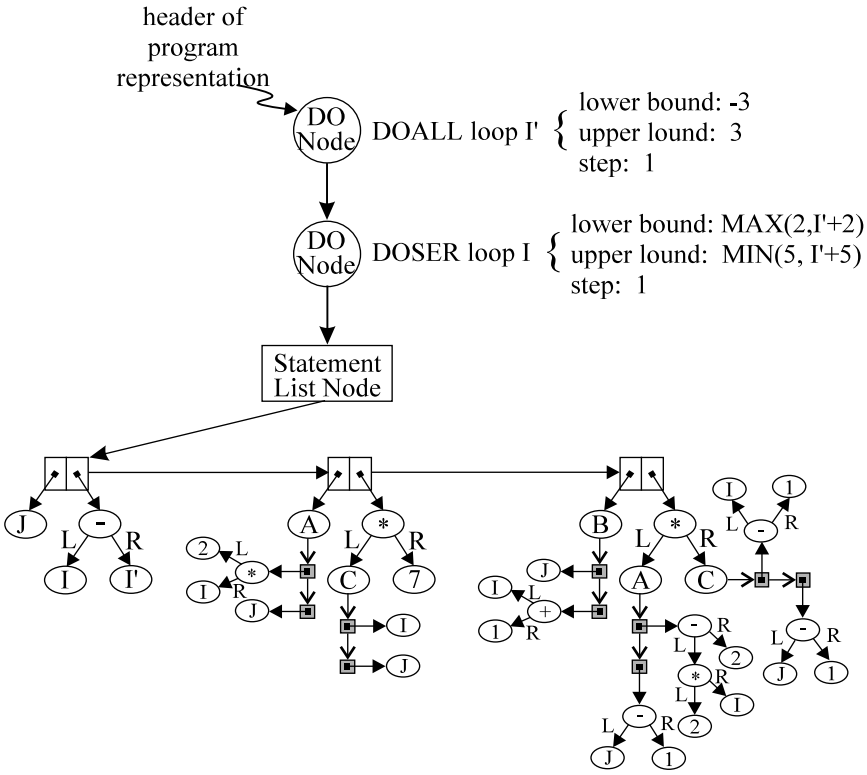



Figure 10: The graph view of the program representation of the transformed program.

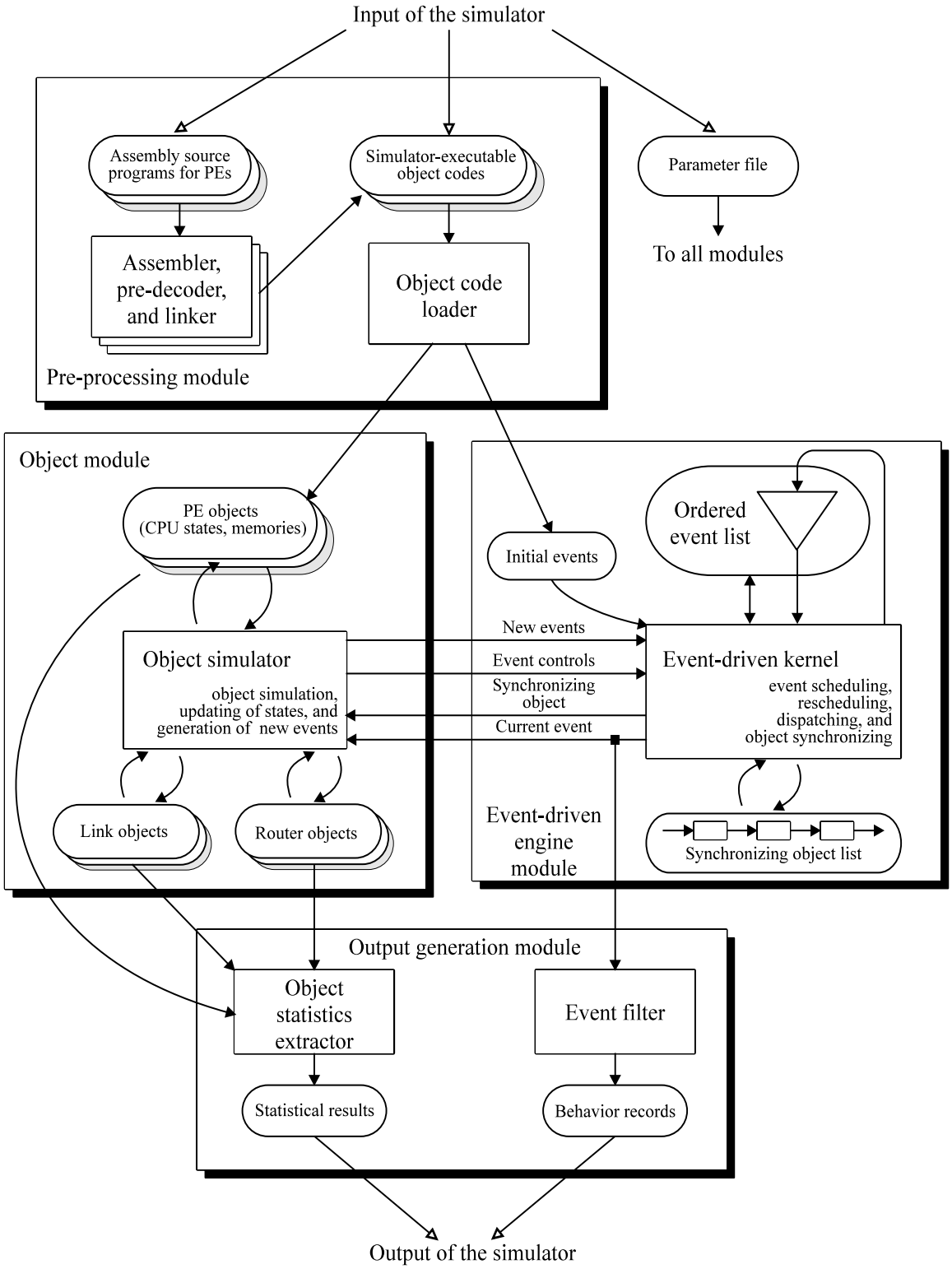



Figure 11: The overall schema of the simulator of distributed memory multicomputers.