1998 International Computer Symposium
Workshop on Computer Architecture
December 17-19, 1998, N.C.K.U., Tainan, Taiwan, R.O.C.

CPSS: AN INTEGRATED SIMULATOR FOR PARALLEL SYSTEMS

Lixin Tao
Computer Science Department
Concordia University
1455 De Maisonneuve Blvd. West
Montreal, Quebec, Canada H3G 1M8
lixin@cs.concordia.ca

Abstract

In this paper, we present the design and implementation of the Concordia Parallel Systems Simulator (CPSS). The ultimate purpose of the CPSS is to provide a parallel programming environment which allows users to study impacts of system and software factors on program performance and to locate performance bottlenecks in parallel programs.

CPSS uses functional simulation technique, and can accurately simulate most multicomputers and multiprocessors based on various communication networks and routing techniques. It also supports reconfigurable and partionable systems. Its unique features include an integrated network simulator, run-time adjustment of simulator parameters, support of virtual-architecture programming, load-time specification of program mapping, and accurate performance information for various components of a parallel system during the execution of a user application.

key words: parallel system simulation, functional simulation, performance evaluation

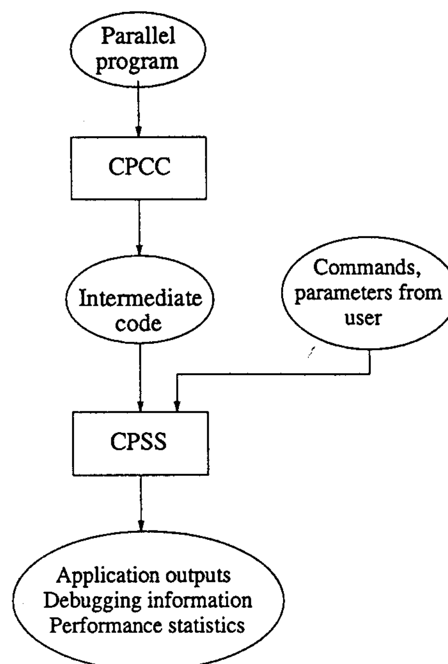

Figure 1: General structure of the CPPE

1 Introduction

The performance of a parallel program depends on various components including hardware architecture, algorithm, programming model, compiling technique, data and task mapping, data routing technique, operating system, and run-time support. Experience shows the interaction among these components also plays an important role in the performance of a program. It is very helpful for programmers to have an efficient performance debugger with which they can collect execution statistics of their programs, identify the performance bottlenecks, fine-tune their source code, and play with system parameters to observe the impact of different system components on their programs.

Since 1994 we focused on the design and implementation of a compact performance debugger prototype, called *Concordia Parallel Programming Environment (CPPE)* (see Figure 1), which includes a compiler front-end transforming code in our *Concordia Parallel C (CPC)* to an abstract syntax tree; a code generator to generate intermediate code for our *Concordia Parallel Systems Simulator (CPSS)*; and CPSS which performs simulation of various parallel systems and their communications subsystems. In addition to being a useful tool to identify performance bottlenecks of parallel code and study the interaction of various system components of a parallel system, CPPE is also a self-contained programming environment in which students can learn and practice parallel programming. CPPE contains about 500K well documented C source code and runs on most platforms including PCs and workstations. On Unix and NT it can simulate more than 4000 parallel processors.

2 Current Status

Several simulation systems for parallel computers have been developed [1, 2, 5, 3, 4]. Existing simulation techniques can be classified into three categories.

1. Direct execution. A parallel program is first compiled into object code which is in the assembly language of the host. During compilation, the compiler identifies two kinds of instruction for the purpose of simulation: local instructions and non-local instructions. An instruction is local if it has effects on only the local processor. Examples of local instructions are register-to-register instructions or memory accesses to a local variable residing in the local memory. Non-local instructions, in contrast, impact another part of the system such as a remote processor or the network. In particular, non-local instructions perform parallel tasks such as process creation/termination, message sends/receives or process synchronization. Each non-local instruction will be simulated via a procedure call which interprets the instruction at the functional level. Local instructions, on the other hand, are executed directly by host processes and timed with the host's clock. This simulation technique is fast but not accurate since the simulation is timed with the host's clock and not the clock of the target architecture.

2. Direct execution with code augmentation. This approach enhances the pure direct execution technique by adding cycle counts of local instructions to the object code during the compilation phase. The cycle count of an instruction is the time it would take to execute this instruction on the real machine. The simulation of local instructions is no longer timed with the host's clock but accumulated using cycle counts added to the object code. This approach thus results in a more accurate simulation than the pure direct execution technique.

3. Functional simulation. A parallel program is first translated into intermediate code of a virtual parallel machine. The set of intermediate code instructions is definable and can be different from the host's assembly language. At run time, the intermediate code instructions are interpreted at the functional level as if they were being executed on the target machine. Functional simulation in general takes more simulating time than the other two techniques, but its simulation results are the most accurate.

2.1 Proteus

The pure direct execution technique correctly simulates the functionality of local instructions but ignores the exact calculation of the actual execution time. Proteus [1] (developed at MIT in 1991) uses code augmentation to count the cycles required by the target machine to execute local instructions.

The application program is first compiled into the host's assembly language. A code-augmenting program will then add cycle counts to local instructions of the object code. The compiled code is first divided into basic blocks of local instructions. A basic block is the smallest block of code delimited by a non-local instruction or an instruction where the execution can branch (e.g. a jump, a function call). Each instruction of a basic block is then matched with a cycle count by looking up a table. The cycle counts of all the instructions in that basic block are then summed and an instruction updating a global cycle counter is added at the end of the block. The cost of each basic block is thus a fixed number and determined at compile time.

Each application process of the target is simulated by a light-weight process (thread) of the host. Context switching on the host is required to interleave execution of threads which run local instruction blocks. Each thread context switching takes 3 microseconds. Non-local instructions are implemented by procedures and interpreted by the simulation engine as in the pure direct execution approach. During program execution, the simulation engine also manages simulating threads: when a simulating thread finishes execution of a basic block, it updates the cycle counter of its simulated processor and gives control to the engine. The engine then selects the next available block for execution and passes control to the corresponding simulating thread. The engine also handles interprocessor communications.

A specific engine must be defined for each simulated MIMD architecture. When the user chooses to simulate a specific multicomputer system, the user has to modify the parameters of the engine. The engine is then recompiled and linked with the user's application.

Proteus' debugging capability depends heavily on the use of sequential *dbx* tools. The user is also allowed to add monitoring code into the simulation engine and the application. During program execution, monitoring code produces data and event traces, and logs the traces into an output file. When the program execution is completed, an graph generator is used to interpret the trace file data and present the results of the simulation.

Although Proteus simulation is fast, it suffers from several drawbacks.

- The timing results may not be accurate because the cost of each basic block is determined at compile time and is a fixed number. In reality, the cost of an instruction depends on other run-time factors such as the operands (or cache hits if the target machine is a shared-memory architecture).

- The simulator can simulate accurately only a limited set of architectures whose instruction sets are similar to that of the host. If the instruction set of the target machine is quite different from that of

1998 International Computer Symposium
Workshop on Computer Architecture
December 17-19, 1998, N.C.K.U., Tainan, Taiwan, R.O.C.

the host, the assignment of a cycle count to every local assembly instruction is no longer accurate.

- Simulation performance may be substantially degraded if the application involves many processes. In addition to the simulation engine process, a host process is created for each application process. If the number of application processes is large, this may incur substantial overheads of context switching among the simulating threads. In practice, augmentation overhead is an insignificant part of simulation cost. Simulating non-local instructions and context switching dominate the cost of simulation [1].
- The simulator is not flexible from the user's point of view. When the architecture is changed, the engine parameters must be modified. The engine is then re-compiled and linked with the user application. This is not convenient, for example, for experimenting with program mappings. This experiment would require to run the same program on different architectures of varied sizes. The simulator must be modified, re-compiled and linked with the application code every time the topology or system size is changed.
- Debugging capability relies mainly on software instrumentation. In-session debugging facility is very limited and depends on sequential *dbx* tools.

2.2 Tango

Tango simulator [2] was built at Stanford University in 1990. Tango and Proteus were developed independently but they are quite similar. However Tango simulates only shared-memory architectures.

Application programs are written in C or Fortran. Parallel features are provided by macros. For instance, *Lock* acquires a binary lock and *Unlock* releases it. The compilation process consists of five steps: macro expansion, compilation into assembly language, code augmentation, assembly and linkage. If a parameter needs to be changed, the simulation engine must be modified and the compilation process is repeated.

Like Proteus, Tango may produce inaccurate simulation results due to fixed costs of local instruction blocks calculated at compile time. Similarly, the target system is assumed to have a basic instruction set that can be approximated by the host architecture in order to obtain accurate simulation.

Novel target machine instructions that do not exist on the host are implemented in libraries and macro packages and will be interpreted at run time. However, if the target machine instruction set differs considerably from the host instruction set, the simulation would approach functional simulation.

Tango's performance is not as good as that of Proteus. Tango uses Unix processes to simulate parallel execution while Proteus uses faster light-weight processes managed by the simulation engine. Context switching time in Tango is 180 to 250 microseconds [2]. If the application execution involves a large number of processes, context switching cost is significant.

Tango does not support any in-session debugging tools. Debugging and statistics data are provided using the instrumented software approach. Many kinds of trace file are generated. System events are recorded in trace files. Program outputs are logged in an output file. There are also process summary file and event trace file. This is not a user-friendly debugging environment for parallel applications.

Tango was implemented for studying shared-memory behaviors, shared-memory synchronization and concurrency abstractions, and for architectural evaluation [2]. It can also be used for application studies. However debugging tools are not adequately provided for code development or performance fine-tuning.

2.3 CPSS Simulation Technique

The CPSS uses the functional simulation approach to simulate execution of parallel programs on a multicomputer system. Applications are written in the CPC language which enhances the C language with parallel features to express process creation/termination and message sends/receives. Parallel operations are defined at a high level of abstraction and reuse existing syntax of the C language wherever possible to promote programmability and ease of learning.

The intermediate instruction set is designed based on an analysis of common operations of parallel systems. The objective is to simulate a wide range of message-passing multicomputers. Every intermediate code instruction is associated with a configurable cost which can be adjusted to match a specific target.

The intermediate instruction set can be extended if it is different from the target's instruction set. The implementation of the simulator is modular and decoupled. So a new intermediate instruction can be added easily to the simulator as a routine which interprets the instruction. The addition of a new intermediate instruction does not affect the simulation of another target whose instruction set does not contain the new instruction since this target would not use the added routine at all.

One of our design goals is to obtain simulation outcomes fast. We do not go into low level details but maintain the essential characteristics of instruction behaviors on target machines in order to yield program outputs within reasonable time limits. Also, the intermediate instruction set is defined at a high-level of abstraction. Application processes are run by a single host process. So there is no host context switching during simulation. This helps to simulate applications with large numbers of processes efficiently.

The CPSS supports virtual architecture programming and run-time mapping to improve programmability of message-passing applications. The burden of

program mapping is now shifted from the user to the simulator. The user writes an application using the virtual architecture most natural to the application. At run-time the virtual architecture will be mapped to the available physical architecture. Moreover, the same source program can be mapped to different physical architectures without any changes to the source code. The simulator provides a library of optimal and optimized mappings.

The CPSS contains a dynamic network simulator. The simulated network is wormhole-routed, flit-based and time-driven. Packets are routed link by link until completely received. The network simulator offers very accurate message routing and communication performance statistics.

The CPSS provides users with a rich set of debugging tools. Users can set instruction and time breakpoints, define trace variables and single-step the source code of a particular process. As a performance debugger, the simulator allows users to define system parameters, examine status of processes, processors and messages, and view computation and communication statistics.

The CPSS is also very flexible and convenient. Users can configure most computation and communication parameters. Values of the parameters can be changed within the same simulation session as often as needed. No re-compilation is required: the same intermediate code of the application and the same simulator code are always executed. This flexibility is unique to CPSS among the existing multicomputer simulators.

3 CPSS High-Level Design

This section gives a high-level description of the simulator. The design is based on the multicomputer system model and the programming model mentioned earlier, and implemented in a way to meet the proposed objectives.

3.1 General Structure of the CPSS

The CPSS (Concordia Parallel Systems Simulator) is an integrated part of the CPPE (Concordia Parallel Programming Environment). In fact, the CPPE consists of two components: the CPCC and the CPSS.

The core of the CPCC is a compiler. After reading a parallel program written in the CPC language, the CPCC builds a complete abstract syntax tree to perform syntax and semantic analysis, and produces object code for a generic virtual machine. Such object code is called *vCode* in the CPPE. The vCode instruction set is defined based on an analysis of common operations of multicomputer systems. To produce vCode, the compilation process makes use of the virtual architecture and does not call for the physical architecture. The advantage of this design is that the CPC parallel program need not be re-compiled every time the underlying target architecture is changed.

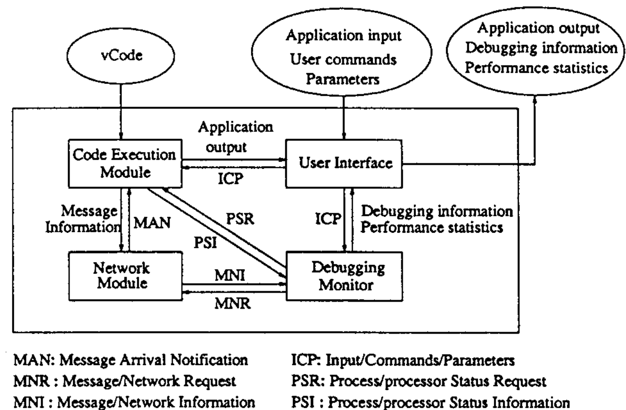

Figure 2: CPSS structure and operations

The vCode produced by the CPCC will be input to the CPSS. Other inputs to the CPSS are parameters and commands from the user. For example, the user can specify the physical topology on which the program will run and the virtual-to-physical-topology mapping. The CPSS then executes the vCode, using the parameters and commands entered by the user. The outputs from the CPSS are the application outputs, performance statistics, and debugging information (Figure 1).

The CPSS itself consists of two major components: the code execution module (CEM) and the network module. There are also two other utility modules interworking with the CEM and the network module. These utilities are the user interface and the debugging monitor. The interactions between the components of the CPSS are illustrated in Figure 2. The following subsections describe the roles and high-level design of the CPSS components and their interactions.

3.2 The Code Execution Module

The CEM plays the role of processing elements of a multicomputer system: it executes the parallel code specified by the parallel program. There is a global clock for the simulated multicomputer system which is updated periodically by the CEM. The CEM contains four main parts:

1. Mapper. The mapper maps the processors of the virtual architecture specified in the CPC program onto the processors of the physical architecture. We provide a library of optimal mappings whose objective is to minimize the maximum path contention level of the parallel program. Optimized and random mappings are also available. The mappings can be one-to-one or many-to-one.

2. Storage Manager. The job of the storage manager is to allocate simulated local memories to processes upon process creation and deallocate this space

upon process termination. The storage manager also allocates/deallocates other kinds of dynamic memory blocks such as activation records upon function calls/returns, and buffers for messages of composite types (e.g. array, structure).

3. Process Scheduler. The process scheduler schedules processes for execution and updates their process structures according to changes in process and processor status, local clocks and the global clock. In the CPSS, parallelism is simulated by time slicing: each application process is given a quantum to run and processes are scheduled in a round-robin fashion. During each quantum, the process scheduler traverses the list of processes, and schedules one process at a time for execution. The execution starts with the instruction specified by the current value of the program counter of the process. The local clock of the process is updated after each instruction. The process runs until its quantum expires or it is put to sleep by some event. The process scheduler then schedules the next process for execution. When every application process has finished its quantum, the global clock is advanced to the next quantum.

4. Instruction Interpreting Routines. These routines interpret vCode instructions. Each instruction is associated with a cost which the routine will look up in a cycle-count table to update the local clock of the current process accordingly.

The CEM contains four major data structures:

- List of parallel processes. Each parallel process created by the virtual-architecture program is associated with a structure PROCESS that contains all the information needed to run this process. Each process has a local clock that keeps track of the present time of this process. All PROCESS structures are placed on a linked list that is maintained and processed by the scheduler.
- Table of processors. This is an array where each element is a structure PROCESSOR. This array is dynamically allocated at the beginning of each run when the physical architecture is known. Each PROCESSOR structure records the status of and information related to a physical processor.
- Memory pool. This is a big array from which local memories of processors are allocated. The array accommodates local memories of all processors in use. The memory pool also provides space for activation records upon function calls, and buffer space for messages of type array or structure. Memory blocks (local memories of processes, activation records, message buffers) are allocated upon requests and returned to the common memory pool when they are no longer used. The first-fit allocation scheme is used for the management of the memory pool.
- Message buffers (channel buffers). This is an array where contents of messages are buffered, waiting to be read. A message of type array or structure is too big to be stored in this array. In this case, the actual contents of the message is stored in a block of the memory pool, and the pointer to this memory block is saved in the array of message buffers.

3.3 The Network Module

The network module is under control of the network manager. The role of the network manager is to

- allocate network resources to messages to be sent,
- route messages and deliver them to destination processors,
- detect and resolve deadlock, if any.

The main features of the wormhole-routed network manager are the reservation of channels and data paths for the messages, the pipelined flow of flits, the release of the reserved resources by tail flits, and the release of flit buffers associated with each virtual channel. The network also uses the global clock mentioned above. In each quantum, all active packets that are not blocked are advanced by one link.

When a message needs to be sent, the sender process writes the message to a message buffer, invokes routine *WH_CEM_SENDS_MSG* to pass message information to the network manager, and continues with its execution (non-blocking send). The network manager will route the message using the information received from the sender. In our design, the network simulator does not route actual contents of messages. It simulates the movement of flits by advancing their ID numbers. When a message reaches the destination, the network manager notifies the CEM of the arrival of the message. The intended reader can then read the message from the message buffer.

The main data structures of the network module are:

- List of new messages. New messages which are being initialized for routing are queued at this list. The waiting time at this list simulates message startup overheads. When the startup overhead time of a new message expires, the message will be removed from this list and appended to the list of active messages.
- List of active messages. This is a linked list of messages which are currently being routed through the network.
- List of active packets. This linked list contains packets belonging to active messages.
- Array of physical link structures. Each physical link structure stores information related to that

link such as the link-request queue, number of occupied virtual channels, an array of LANE structures (a structure for each virtual channel of the link).

- Routing table. When a packet arrives at an intermediate node, the router uses the addresses of the current node and the destination to look up the routing table for the address of the next node on the path.

3.4 The User Interface

The user interface enables the user to interactively communicate with the simulator. The user interface receives parameters and commands from the user, validates the received information, and pass valid parameters or commands to the appropriate module (the CEM, the network module, or the debugging monitor). During execution of a parallel program, the user interface interacts with the debugging monitor to display performance statistics and debugging information. Program outputs are also transfered from the CEM to the user interface for displaying.

3.5 The Debugging Monitor

The debugging monitor is responsible for handling the debugging mechanisms. During execution of the parallel program, the CEM and the network manager regularly update the debugging variables. After each breakpoint and after the completion of program execution, the debugging monitor collects and processes the values of the debugging variables to generate performance statistics and other information about program execution.

4 Process Management

In this section we discuss how parallel processes and their execution are simulated. We also describe data structures needed to manage and run simulated parallel processes.

4.1 Data Structures

4.1.1 Process Control Block

Every application process is associated with a process control block (PCB) that stores various information needed for its execution. A PCB is dynamically allocated upon the creation of a new process, and deallocated when the process terminates.

When a new process is created, a PCB is allocated and appended to the list of processes. This is a singly-linked list managed by three pointers: *actProcHead* pointing to the first entry of the list, *actProcTail* pointing to the last entry of the list and *curProc* pointing to the PCB of the process currently running.

4.2 Parallelism by Time Slicing

Parallel execution of application processes are simulated by time slicing. The execution of a parallel program is divided into quanta, each quantum lasting q clock cycles (or time units) where $q > 0$. During each quantum, the scheduler traverses the list of processes, and schedules every process in a round-robin fashion. If the process is able to run (i.e. it is not blocked or delayed), it executes until its time slice of q time units expires or is put to sleep by some event. The scheduler then gets the next process in the list and schedules this process. As the scheduler moves downward the list, the value of pointer *curProcess* is updated to identify the process currently running. When the last process in the list (i.e. the process whose entry is pointed to by *actProcTail*) finishes its time slice, the global clock (*globClock*) is advanced by q time units to the next quantum and a new quantum begins. Such a quantum simulates q time units of parallel execution of all processes on a real parallel machine.

The duration of a quantum is the time for a non-header flit to move from one node to an adjacent node (*flit latency*). For example, if $q = 3$, the CEM (code execution module) runs parallel processes for 3 clock cycles, and then the network simulator moves unblocked flits forward by one link. The computation quantum and the communication step are considered to be running in parallel. The quantum can be a fractional number. For instance, when $q = 0.5$, parallel processes run for one clock cycle every time the network simulator advances unblocked flits by two hops.

4.3 Process States and Job Scheduling

Possible states of a process p are

- *Ready*: p can be scheduled for running.
- *Running*: p is currently executing its code.
- *Delayed*: p is put to sleep and the wakeup time is known. The process will be waken up by the scheduler when the wakeup time expires.
- *Blocked*: p is put to sleep and the wakeup time is not known in advance. p will be unblocked by another process (e.g., one of its children) or an event (e.g., the arrival of a message in the wormhole-routed network).
- *Terminated*: p has completed the execution of its code.

Figure 3 shows the transitions among the states. Figure 4 shows the process scheduling algorithm.

4.4 *fork* Processes

Execution of a *fork* statement will create a new process. The parent can specify the virtual processor on which the child process will run. The parent can also

1998 International Computer Symposium
Workshop on Computer Architecture
December 17-19, 1998, N.C.K.U., Tainan, Taiwan, R.O.C.

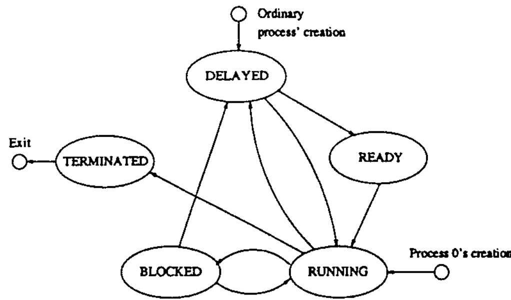

Figure 3: Process state transitions

assign channel variables to the child so that the child will use these channel variables to communicate with other processes. The child will be the owner of the assigned channel variables and only it can read from these channels.

After a parent process spawns a *fork* child, it can continue with the instruction following the *fork* right away. The parent and the child are then running in parallel.

When a parent process wants to terminate, it must wait for all of its children to finish first. The parent will be blocked until the last child terminates; this child will wake up the parent and let the parent terminate.

The algorithm for executing a *fork* is shown in Figure 5. In the algorithm, the parent process executes steps 1, 2, 3, 4, and 5b, while the new *fork* child will execute steps 5a and 6. The algorithm of *NewForkChild* is summarized in Figure 6.

5 Communication with Channel Variables

Write/read operations on channel variables in a CPC program abstract message send/receive in the real multicomputer. A channel can be considered as an infinite buffer owned by some process p, where other processes can deposit messages of the same type for p to read.

To send a message of type *msgType* to the receiver process r, the sender s identifies a channel variable v of type *msgType* owned by process r. Process s executes an assignment statement where v is on the left-hand side (LHS) of the assignment. The message content is the value of the expression on the right-hand side (RHS) of the assignment.

Process r must be aware of to which channel s has written the message (since r may have more than one channel of type *msgType*). Process r then executes a channel read on that channel to get the message from s. If the message has not arrived at r's processor, r's execution is suspended until the message is available, at which time r removes the message from the channel buffer.

Message types and thus channel types can be either

```
JobScheduler()
{ logic_deadlock = TRUE;
  done = FALSE;  count = 0;
  do {
   count++;
   if (curProc == actProcHead)
    globClock += quantumDuration;
   // completed one round, so advance
   // global clock to next quantum
   p = curProc;
   // p points to PCB of process
   // currently running
   if (p->state == Delayed)
    if (p->waketime < globClock)
    //time to wake up
     { p->state = Ready;
       if (p->waketime > p->time)
       //wakeup time exceeds local clock
         p->time = p->waketime;
         // update p's local clock
     } else  // continue to sleep
     { p->time = globClock;
       // update p's local clock
       logic_deadlock = FALSE;
     }
   if (p->state == Running)
   { logic_deadlock = FALSE;
     if (p->time < globClock)
      done = TRUE;
     // p will continue to run
   } else if (p->state == Ready)
   { logic_deadlock = FALSE;
     m = pointer to PCB of process
         currently running on p's
         processor;
     if (m == NULL)
     // there is no Running process on
     // p's processor
     { curProc = p;  p->state = Running;
       // p is scheduled to run
       done = TRUE;
     } else if (
         (m reached contextSwitchLimit) and
         (m's priority <= p's priority)
        ) // then context-switch
     { curProc = p;  p->state = Running;
       m->state = Ready;  done = TRUE;
     } else  p->time = globClock;
     // p is not allowed to run
   } else if (p->state == Blocked)
     // continue to sleep
      p->time = globClock;
      // update p's local clock
   else if (p->state == Terminated)
    // p terminated
    remove p's PCB from the list of processes;
   else if (! done)
    // if p gets here, p is not
    // allowed to run
    p = p->next;
    // consider the next process in the list
  } while (!(done or
     (logic_deadlock and
      count > total number of processes)));
  if (logic_deadlock and no messages
      in network) System_Deadlock();
}
```

Figure 4: Process scheduling algorithm

```
Step 1 :  Calculate the ID of the virtual
          processor on which the child
          will run;
Step 2 :  Assign the specified channel
          variable(s) to the child;
Step 3 :  NewForkChild;
          // create a new fork child
Step 4 :  ForkJump;
          // after creating the child,
          // the parent jumps to the next
          // instruction following the fork
          // (Step 5b)
Step 5a:  Child executes the code body of
          the fork;
Step 6 :  ForkChildEnd;
          // the fork child terminates
Step 5b:  The parent executes the instruction
          following the fork
```

Figure 5: Algorithm for *fork* process creation (vCode instructions are in bold face)

basic types (integer, char, float, pointer or enumerate) or composite types (structure, array).

6 On-Line Network Simulation

The interprocess communications generated by CEM is carried out by our on-line wormhole routing simulator. Wormhole routing is the dominant routing technique in the second generation massively parallel systems. While some simulation techniques for wormhole routing have been used before, they are mainly designed as independent off-line tools to study the performance of such networks under artificial communication constraints. The unique feature of our design is on-line simulation. The execution module of CPSS generates interprocessor communications on the fly, and our wormhole routing simulator simulates the data movement in the network in parallel and collects rich set of information on network performance. The on-line nature enforces harsher constraints on our design, but it also makes our design a valuable tool to study wormhole routing networks under realistic applications. It can also be used as a tool for designing and testing new routing strategies and designs.

```
NewForkChild()
{
 //Precondition:
 // ID of child's virtual processor is on top
 // of the parent's stack.
 // childPtr: pointer to the PCB of the
 // new fork child.
 Allocate a PCB for the new child;
 Init_PCB(); // initialize the child's PCB
 Append the new PCB to the list of processes;
 Allocate a process frame for the child {
   childPtr->base = starting address of
                    the process frame;
   childPtr->T    = childPtr->base - 1;
   childPtr->stackTopLim = childPtr->base +
               (size of process frame) - 1; }
 Update the descriptor of the child's processor
 {
   physProsorTable[childPhysProcID].status
     = Used;
   physProsorTable[childPhysProcID]
                             .nbrProcesses++;
 }
 Pop the top value off the parent's stack;
 // this value is child's virtual processor
 // ID that was used in function Init_PCB()
 parentPtr->forkCount++;
 // parent has one more fork child
 if the child begins with a function call
   Prepare_Parm_Eval();
   // prepare for parameter evaluation
 else // child's code is a single statement
   Send a birth message to the child's
   processor;
}
```

Figure 6: Algorithm of vCode instruction *NewForkChild*

References

[1] E.A. Brewer, et al, "Proteus: A high performance parallel-architecture simulator," Technical Report MIT/LCS/TR-516, Massachusetts Institute of Technology, Laboratory of Computer Science, September 1991

[2] H. Davis, S.R. Goldschmidt, and J. Hennessy, "Multiprocessor simulation and tracing using Tango", *Proceedings of the 1991 International Conference on Parallel Processing*, August 1991, vol.2, pp.99-107

[3] B.P. Lester, *The art of parallel programming*, Prentice Hall, 1993

[4] E. Olk, "PARSE: Simulation of message passing communication networks," *Proceedings of the 27th Annual Simulation Symposium*, 1994, pp.115-124

[5] E. Reiher, H.H.J. Hum, and A. Singh, "Simulating networks of superscalar processors," *Proceedings of the Supercomputing Symposium*, 1993, pp.125-133