EC/DSIM: A Frontend and Simulator for Huge Parallel Systems*

Gudjon Hermannsson
Computer Science, SUNY
Stony Brook NY 11794

Ai Li
Computer Science, U. of Victoria
Victoria, B.C., Canada V8W 3P6

Larry Wittie
Computer Science, SUNY
Stony Brook NY 11794

Abstract

This paper presents a new fast way to simulate large networks of computers. The method uses a frontend EC, which accepts a parallel C program and translates it into a program in an intermediate language for parallel system simulations. An event driven simulator for distributed shared memory systems, DSIM, uses the intermediate language to simulate and obtain efficiency results in networks of thousands of processors.

In simulations of large parallel systems, enormous memory needs and long execution times are major difficulties. To minimize these problems, the new method analyzes system performance by using extracted execution step information, e.g. the number of shared writes in a code segment, to predict task completion times rather than executing the program step by step. Workstation evaluations of parallel codes running on several thousand processors are feasible using EC/DSIM.

1 Introduction

Computer simulations allow exploration, evaluation, and analysis of systems that are new, not physically available, or too big to study. Simulating a new computer system before actually building it gives answers to critical questions like whether the system design is feasible and how to arrange different components to maximize performance. It is especially desirable to evaluate systems with non-traditional designs, to avoid the waste of time and money. Design issues that appear during the simulation process itself may indicate important control methods for the system under study. Simulation of computer hardware can lead to software strategies for efficient use of that and similar systems. The EC frontend translator described in this paper is a by-product of the process of simulating parallel systems.

As sequential computer technology reaches physical limits, parallel computing becomes much more attractive to meet demands for greater computational speed. Parallel computers are very complicated. Before building a prototype, extensive simulations are used to determine what aspects of the design will most affect performance. Such simulations can be either trace-driven or execution-driven. Trace-driven simulation is not a good method for evaluating massively parallel processors[4]. Interprocessor communication patterns may not be preserved when changing architecture features or just moving to larger systems. However, simulating large parallel systems using execution-driven simulations has two major difficulties:

- The large sized actively used memory needed for program and computation system state; and
- The unacceptably long execution times needed to change state variables for each component.

Trying to hold state information on disk fails because it increases simulator execution times by ten or more. These twin constraints have effectively prevented detailed simulation on a single computer of parallel systems with more than one hundred processors.

The behavior of a parallel systems depends on many factors: the application program, processor speed, processor count, communication speed, and network topology. Most program computations do not affect system behavior. During simulations just to evaluate execution efficiency, it is not necessary to expend memory to store the results of most calculations, nor time to evaluate them. What is needed is a record of performance information and the few calculation results that do affect performance. For example, when executing the statement $i = 3$, the time to store the

*This research has been supported in part by Department of Energy/Superconducting Super Collider contract SSC-91W09964; by National Science Foundation grants for equipment (CDA88-22721, CDA90-22388, and CDA93-03181) and research MIP89-22353; by National Aeronautics and Space Administration grant NAG-1-249; and by Office of Naval Research grant N00014-88-K-0383.

1060-3425/94 $3.00 © 1994 IEEE

new value does affect system performance, but the value that is assigned does not, unless it controls program execution steps.

Performance evaluation of a parallel system running a given program requires three classes of information: the program computational structure that controls execution; the values of the few variables that affect program control flow; and the impact of each program execution step on performance. The times needed for computation operations, memory accesses, and communications affect system behavior; the precise result that a program computes is usually not important to processor performance. Compared to detailed simulations of program execution, simulations that retain only performance information require much less memory space, take less time to evaluate system efficiency, and can simulate much larger parallel systems, ones with hundreds or thousands of processors.

Some researchers[14, 2] propose statistical methods to predict system behavior, for example whether a loop should be executed. However, statistics often cannot accurately predict the chaotic interactions of multiple processors, for example, synchronizations that can block execution. The method proposed in this paper precisely simulates control flow and synchronization exchanges.

The parallel system performance simulator, DSIM, eliminates non-critical calculations. Much recent research in parallel computers has helped design distributed shared memory (DSM) systems which simplify programming, but allow hundreds of processors to work in concert. DSIM evaluates the performance of DSM architectures. Since it needs much less memory space than a detailed step by step simulator, DSIM can simulate the execution of programs on networks of several thousand processors and evaluate the potential for scaling to massively parallel systems.

Using the EC frontend, details of execution structures and computation times are automatically extracted from a given application program. EC differs from a normal compiler, where the goal is to break program statements into the details needed to calculate all results. The EC frontend must determine the execution structure of a given program, discover all variables that control the selection of execution steps, and collect the cost information for all execution steps. It must translate structure and cost information into the internal code understood by the simulator which executes this new program image. In the statement

```
a=b*c+d;
```

a performance simulator needs know only that there

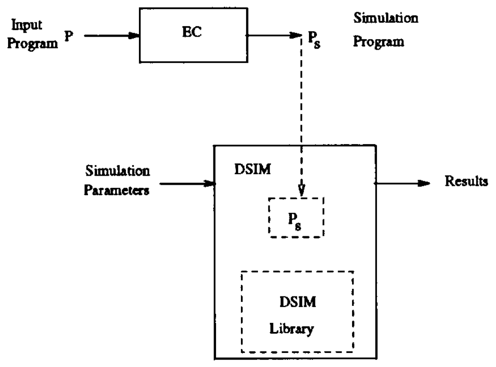

Figure 1: EC Frontend to DSIM Simulator

are two arithmetic operations, three reads of variable values and one write of a value. The operations may be classified in more detail, e.g. floating or non-floating point operations and multiplication or plus operations. The EC frontend performs the extractions and translations for simulation of C programs written for parallel architectures, as shown in Figure 1. EC has been implemented for the simulator DSIM, but can be extended to match other architectural simulators.

For their statistical simulator, Qin et al[14] have a very different frontend which accepts a Pascal-like language and generates graph structures for step-by-step simulation. EC translates C-programs into function calls which the simulator can use to build the desired execution structure.

Tango[8] is a simulator environment created at Stanford to evaluate multiprocessor systems. Tango accepts either C or Fortran and can be either trace-driven or execution-driven. However, when execution-driven, Tango simulates a program step by step.

In contrast, EC can combine information extracted from several statements into one function call. Details of EC functions are given in later sections.

The next section contains general background on DSM systems, since simulating their performance was the main motivation for creating DSIM. DSIM is described in Section 3. Section 4 discusses design and implementation issues for EC. Section 5 shows EC/DSIM results. Appendix A gives a sample of program translation.

2 Distributed Shared Memory

Researchers have recently invented DSM to try to keep the programming advantages of shared memory while scaling to high degrees of parallelism. DSM systems pass messages, but hide the underlying mechanism to provide a shared memory programming paradigm. DSM solves many contention and scaling problems of parallel systems.

One general classification of DSM systems considers how each handles remote memory accesses in the logically shared data space. At one end of the spectrum are demand driven mechanisms, which halt a processor to fetch needed data across the network, introducing long latency on every remote access. Network traffic is minimized by passing only needed data and data requests. At the other extreme are eagersharing mechanisms, built on the principle that whenever shared data changes, it is immediately sent to all processors that may need it.

There are many demand-driven DSM systems[11, 19, 9, 5, 6, 1]. Notable is the Stanford Directory Architecture for SHared memory (DASH) [11]. DASH is a scalable two level on-demand shared-memory network. Each processing node is a bus-based multiprocessor with a snooping protocol for cache coherence. Nodes are connected through an interconnection network with distributed directory coherence protocol. To reduce delay time caused by blocking, DASH researchers are considering compiler generated prefetch instructions and update-delivery for variables.

There are a few eager DSM systems: Princeton PRAM[12], Stony Brook SESAME-MERLIN[16, 17, 13, 18], and Carnegie-Mellon PLUS[3]. The main goal of eagersharing is to hide access latency by having data already local to each sharing processor when needed.

Eagersharing can be implemented as a part of a cache coherence protocol for multiprocessors or more simply as a system of interfaces between workstations. In SESAME, each network workstation has an interface snooping on its memory system and linked to a plexus of gigabit per second optical fibers. Whenever a variable value is written to local memory, its address is checked against a preset directory of shared variable regions and a copy of the new value is immediately, 'eagerly' multicast to all other processors that share the variable. Together, the interfaces handle all routing of copies of each shared variable. There is no degradation of performance on the intervening host workstations. There is cycle stealing to store a local copy on each system sharing the variable.

The SESAME[Scalable Eager ShAring MEmory] project [16, 17] at Stony Brook grew from work in 1987 and 1988 on MERLIN [13, 18]. Readers wanting more DSM details should refer to the cited papers.

3 DSIM Simulator for DSM Systems

DSIM is an event-driven simulator for DSM systems. It was initially designed to evaluate and predict system performance for the SESAME eager DSM system. Later, demand-driven DSM features such as fetch and prefetch were added for comparison with other architectures.

The primary DSIM events are estimated completion times of lengthy calculations, propagation of messages in the network, synchronization accesses, and memory accesses through bus and network interfaces. Events signal possible changes in system behavior, for example, an increase in memory bus traffic that may slow processor speed.

The computation model for DSIM assumes that the instantaneous computation rate of each processor is determined by the minimum of three constraints: (1) its *cpu_to_memory* bandwidth versus the number of operands that must be fetched from or stored into local memory for each calculation step; (2) its sharing interface acceptance rate versus the number, if any, of shared variables written during each calculation; and (3) the peak processor floating point operation rate. Bandwidth is the most important limit for most codes. Delays on locks and barriers are recorded as intervals of zero computation rate. The model is justified by observing that computation speeds for most pipelined processors are limited mainly by the CPU-memory bandwidth needed to access data[10].

The parallel C language application codes used for the simulation are SPMD (Single-Program, Multiple-Data) programs. Every processor independently executes the same program, i.e., not in lock step. Program flow can depend on a CPU variable which encodes the identification number of the executing processor. At any time different processors may be executing different parts of the program and can evaluate specified data items. At explicit "barrier" synchronization points, each CPU must wait until all CPUs have reached the same point. Since hand translation is very tedious, EC is normally used to compile each SPMD program into the internal code structures which DSIM interprets.

For each processor in the simulated network, DSIM allocates space for its program counter, its CPU identification number, special variables including the total count of CPUs in the system, local variables from the code that help control program execution, and all locks and variables shared with other processors.

One central copy of the internal representation of the program is used by all processors. It is a linked list of execution control modules that is interpreted by DSIM to predict system performance. Different types of modules correspond to different source program structures and commands:

- A task module represents a single vector or scalar computation plus its assignment. It contains information about the operations involved and the number and types of memory accesses.
- An iteration module represents a program loop structure. It contains information about loop conditions plus execution control pointers to code modules within and after the loop.
- A condition module represents an if-then or if-then-else program structure. It contains parameters to evaluate the conditional expression plus execution control pointers to the start and end of the module executed if the expression is true.
- A value_set module evaluates a variable, which may be explicit in the code or internal to the simulator. Program variables which make some contribution to execution control, for example, the index of a *for*-loop, have to be evaluated. Each value_set variable is assumed to be a scalar value kept in a register. Its evaluation costs are negligible compared to floating point evaluations of large vectors and are treated as zero by DSIM.
- A lock module changes or tests a lock value used for synchronization. Since lock values determine whether individual processors are waiting idly or executing as rapidly as other resource constraints allow, they can critically determine program behavior. Locks may be treated very differently from ordinary variables in some DSM systems.
- A barrier module marks a global synchronization point that all processors must reach before any of them are allowed to continue.
- An enable/disable sharing module deals with a special SESAME dynamic sharing control mechanism that allows greater network speedup and program scalability. By dynamic enabling and disabling of sharing for selected variables, users can prevent sharing of intermediate data changes not needed by other CPUs.
- An end module indicates the end of an iteration segment, condition segment, or the program.

Figure 2 shows the linked list for a small DSIM program consisting of a value_set module followed by an if-then-else conditional module. Program information is extracted by the EC frontend, which is discussed in detail in the following section.

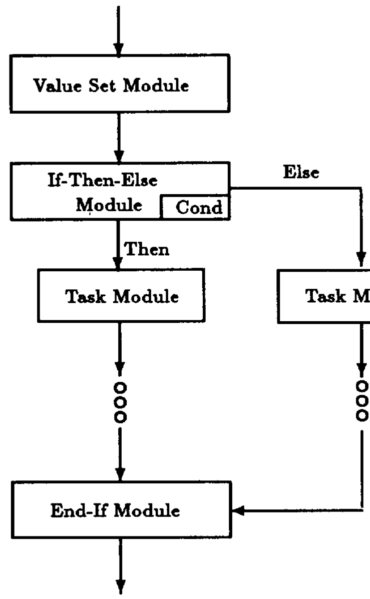

Figure 2: Sample DSIM Program During Simulation

4 Design and Implementation of EC

EC is the compiler frontend that translates parallel programs into code to generate the simulation modules for DSIM.

4.1 The Input Language

Input to EC is a C-like program for parallel architectures. The input language differs from standard C in two ways. EC accepts a slightly simplified subset of C sufficient for most simulation purposes. The end of this subsection lists features not currently in EC. In addition, several keywords have been introduced for parallel programming and simulation purposes. The new parallel execution keywords that control memory sharing and processor synchronization are:

- **shared** declares a variable or a memory region accessible by multiple processors. A memory "region" is a set of memory locations that are somehow related. It does not need to be continuous. Although most are neither calculated nor kept by the simulator, all shared values must be specified to account correctly for the messages carrying their changes on the memory busses and sharing network. In the current version of EC, each variable with the keyword shared must be an array.

- **lock** declares a region of flags (valued 0 or 1) used for synchronization. Each CPU has DSIM space reserved for its own copy of all lock flags.
- **counter** differs from lock only by declaring a memory region of words instead of binary flags. Counters are treated like locks. DSIM implements a lock region compactly as a bit-vector and a counter region as an array of words.
- **barrier** sets up a synchronization point. No CPU can advance beyond a barrier until all reach it.
- **wait** enables a processor to sleep until a lock or counter condition is true. Whenever the lock is updated, the sleeping processor is awakened to re-check the condition.
- **enable_sh** and **disable_sh** allow the programmer to activate or inactivate the sharing of a region. This technique, originated for SESAME, can be used to improve performance by eliminating unnecessary traffic in eagersharing systems.

The other new keywords supply program scaling parameters that can be changed as simulator options without program recompilation:

- **vardatasize** specifies a read-only variable which contains the size of the data set to be processed. It is the rough size of the application task.
- **varcpu** declares a read-only variable set by DSIM to give the unique identifier of the CPU executing the code. Each user program can interrogate this variable to determine which CPU is processing each section of data and code.
- **varcpunum** declares a read-only variable representing the total number of processors currently attempting to run the parallel program.
- **externalargument** declares a read-only variable representing an arbitrary argument for the code.
- **if_init_needed** declares a read-only runtime parameter that can be used to include initialization effects in efficiency measurements. Any program section which does initialization should be conditional on this value.

The current version of EC has several restrictions.

1. At least one *main* function has to be present, otherwise no translation will take place.
2. The maximum number of array dimensions is four.
3. The maximum nesting of parenthesized expressions is nine.
4. Shared, counter and lock variables must be arrays.
5. Other things that are not yet supported, but are being considered: pointers, structures, and the statements *break* and *continue*.

A planned project will modify EC to translate Fortran language programs as well.

4.2 Output From EC

For a given input program, parameterized EC generates a program consisting of a set of function calls. Each call builds and links one module in the internal representation of the program. The DSIM simulator interprets the modules to determine application program behavior on each CPU. For example, a set of function calls:

CodeFor(...);
CodeTask(...);
...
CodeEnd();

is generated to represent a *for*-loop. *CodeFor*(...) corresponds to an iteration module and is passed the index variable, increment expression, termination condition, and necessary pointers to other modules. *CodeFor*(...) is general enough to encode all standard iteration loops: for, while, and do. *CodeTask*(...) corresponds to a vector task module, which contains the number of operations (floating/non-floating) and the number of memory accesses (write/read/shared write) in a block of assignment sentences inside the loop. For a shared write, the name of the shared region will also be passed by one extra parameter. Shared writes to different regions cannot be grouped into a single task module. Instead, multiple CodeTask function calls must be generated. *CodeEnd*() ends the loop.

4.3 Control Variables and Expressions

In a given program, variables can be divided into two classes: *control variables* that direct execution into different program parts and ***non-control variables*** that do not. For example, in the following program loop,

$for(i = 0; i < 10; i++)$
$a[i] = i * i;$

the variable i is a control variable and array a is a non-control variable. Since values of control variables affect program instruction flow, they must be calculated and stored during simulation. Computations of non-control variables are not important so long as their operations and memory accesses are counted.

For a given parallel program P, a variable v is a control variable if and only if

(1) v is declared as a lock or counter in P, or

(2) v appears in $expr$ for a control statement in P: **if**($expr$)...; **while**($expr$)...; **do**...**while**($expr$); **for**($expr$)...; **wait**($expr$), or (2)

(3) v appears in the right hand side of an assignment in P to an existing control variable.

Conditions (1)-(2) define *direct control variables. Indirect control variables* are defined only by (3). A *control expression* is an expression (expr) in a control statement listed in (2), or an assignment with a control variable as the left hand side. Parameters of functions in control expressions are also control variables.

Since most parallel programs iteratively process large data sets, there are many more non-control variables, including most arrays, than control variables. For example, in Gaussian elimination programs, control variables include matrix size, loop indices, and synchronization variables. The values in the coefficient matrix do not significantly affect parallel system performance. Classifying them as non-control variables reduces valuable space and time during simulation.

For a given program P, the simulation program produced by EC has the same computation structure as P. DSIM uses the control variables to follow the computation structure of P during the simulation. For example, DSIM will loop ten times to evaluate the performance of the program segment

$for(i = 0; i < 10; i++)$
{loop body}

4.4 Expression Translation

EC handles each expression in one or two ways:

- The information embodiment section collects information on the number of operations, memory accesses, and network messages from each set of expressions in the program. When a sequential block of assignments does not contain control expressions, information is grouped into one task module. DSIM determines the execution time for the task module according to the hardware configuration and dynamic computation environment.
- The operation assembly section processes control expressions. Besides collecting information from control expressions, EC generates a sequence of function calls to evaluate each expression that controls program execution or changes the value of a control variable. Each function call performs a primitive operation on two operands.

For example, if a is a control variable, the assignment $a = b * c + d$; is interpreted as two quadruples: $(tmp, *, b, c)$ $(a, +, tmp, d)$, where tmp is a temporary variable. Each quadruple generates a value_set module for the DSIM simulator.

4.5 Implementation of EC

EC is implemented as a compiler with three passes:

1. The lexical analyzer and parser generate a syntax tree and extract information describing each node, for example, name, type, and pointers. Direct control variables are marked in the symbol table. The parser marks tree nodes representing direct control variables and control expressions which can be identified without backtracking.
2. The intermediate process marks all remaining control expressions in the syntax tree and all indirect control variables in the symbol table.
3. The code generator counts operations and memory accesses, and outputs the simulation program dictated by the marked syntax tree.

The first pass is similar to that of a standard compiler. The second follows the definition of control variables and expressions. The rest of this section briefly discusses code generation.

4.6 Code Generation by EC

The major concerns during EC code generation are:

- Declarations. Synchronization variables (locks, counters), shared variables, control variables, and compiler defined variables need to be declared by generating declaration function calls. Compiler variables are temporary variables used to evaluate some control expression during simulation. For example, the variable *tmp* at the end of section 4.4 is compiler defined. Markings in the symbol table control the generation of declaration calls.

The declaration function call for each control variable must request allocation of memory space for the value calculated during simulation. Since the values of non-synchronization shared variables are not needed during simulation, only a bit is reserved to reflect whether a valid copy of the variable is in the CPU's local memory at any given time during simulation. Only the addresses (names) of shared variables are needed to generate network sharing events; values are not needed. However, globally shared locks need local space to keep the value and information about the lock owner.

- Module Generation. The syntax tree represents the program structure. Function calls are generated by traveling the syntax tree and interpreting the details for each statement. During simulator initialization, each call uses its parameters to construct a simulator code module.

A different module function call is generated for each type of control structure statement. For example, an iteration module call is generated for a *for* or *while* statement and a conditional module call for an *if* statement. The body of each statement is handled recursively. An end module is generated after each control structure.

Unlike a standard compiler, EC generates programs that produce actual values only for *control value* computations. The second pass has decided where value_set function calls must be generated to evaluate control expressions.

While checking each expression of a *timed task*, its memory access and operation counts are used by EC to generate a CodeTask function call. The information for a sequential block of statements can be grouped into one task module until a task terminator statement is encountered. These statements terminate a task: end, any control structure (since resource usage inside must be counted separately), barrier (since it may delay execution and all resource usage), assignment within a different shared region (since there is at most one shared region name per task function call), and disable/enable sharing (hardware-related statements can change sharing status and performance).

All other statements generate a single function call, straightforwardly. The code generation phase of EC produces a simulation program consisting of a set of function calls. This program, combined with the DSIM library, drives the DSIM simulation of the original EC program. Appendix A gives a more complete example of EC output.

5 Results from Using EC and DSIM

This section shows EC and DSIM simulation results for a parallel Gaussian elimination code running on SESAME eagerly shared distributed memory systems[17] with large numbers of processors. Gaussian elimination is a challenging benchmark for parallel computer performance because the workload per processing step varies from a whole row to one element during reduction of the matrix to triangular form, all processors must rapidly share the results of occasional steps that complete a row reduction, and global sharing all array changes all the time does not work since it overloads memory busses and networks. A well-designed parallel algorithm already exists. This Livermore National Labs code [7] runs rapidly on Sequent shared memory multiprocessors. The algorithm has been modified for EC [15]: only the writes for the final reduction of each row i, when a multiple of row $i-1$ is being subtracted, are shared with other processors.

In the simulation, each processor is assumed to have a peak computation speed of 33 MFLOPS for 64-bit data and a fast 400 MB/sec local memory bus. Each data sharing hop takes $200ns$, the delay for eagersharing interfaces using 1 gigabit/sec serial links.

Figure 3 shows system performance for Gaussian elimination of a 2800 by 2800 matrix representing 2800 equations as run on SESAME systems with 1 to 2800 computers. It demonstrates that SESAME eagersharing interfaces can efficiently support difficult problems running on large networks, and that DSIM can produce simulation results for large parallel systems. The curves show both average processor efficiency as a percentage of peak processing power, and sustained network speedup compared to one processor at the peak power. Network performance peaks at a cumulative sustained speedup of 690 from about 1400 processors, for an effective computing power of 23 GigaFLOPS. Speedup is nearly constant at 625 to 690 for any network of 900 processors or more. Further analysis indicates that sustained efficiencies of nearly 50% are possible for 250,000 CPUs of 64MBytes each using Gaus-

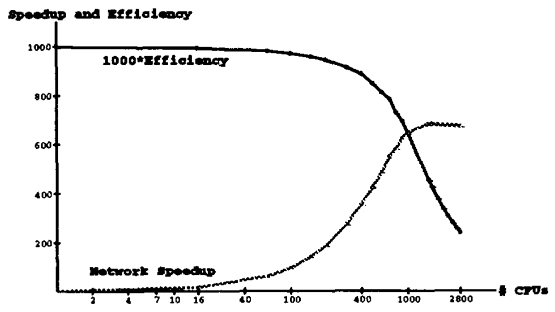

Figure 3: Gaussian Elimination of 2800 Equations

sian elimination on systems of 1,000,000 equations, for network speedup of 120,000 (4 TeraFLOPS).

EC and DSIM save significant time and memory, allowing simulation of large parallel systems. Computer system simulations usually take hundreds of times longer than reality. DSIM takes only about five times reality. For Figure 3, the simulations of Gaussian elimination for 2800 equations with fifty different network sizes took DSIM only ten days total on a 30 MIPS computer with 64 MBytes of memory. A DSIM simulator modified to calculate all numeric results would have taken twelve days and at least 256 MBytes of memory on the same computer.

6 Conclusions

This paper explains a new method of using abstract execution information to predict and to analyze parallel system performance, instead of simulating programs in full detail. The method provides faster execution and requires much less memory space than step by step simulation. The method has been implemented as two components: EC, a translator for parallel C programs; and DSIM, a simulator for distributed shared memory systems.

EC not only generates executable code for a program, but also extracts hardware resource usage information to drive a model of execution efficiency in large parallel computing systems. Although the current version of EC is designed as a frontend for the DSIM simulator, it can be extended for use with other parallel system simulators or to study extensions to the extraction method.

To support a simulator other than DSIM, the function call generator in EC and the function definitions in the simulator initializer would need to be modified. Only the first EC pass, the lexical analyzer and parser, needs to be changed to accept a different programming language, such as Fortran. EC can easily generate code for all computations to guarantee correctness of the simulated algorithm or to provide detailed step by step simulations as long as simulator speed and memory space are not limiting factors. EC could also generate task module function calls for different levels of approximation, i.e., for whole loops or even loops of loops. CodeTasks of different grain sizes would allow very precise small grained simulations of small parallel systems, and faster large grained evaluations of huge systems.

The implementation of EC and DSIM is a new attempt to help shorten the simulation process for large parallel systems. It allows rapid simulations that consider only time costs of operations and memory accesses. Using EC, a simulator can automatically compile any given program and simulate its behavior without carrying out unnecessary numerical computations. Integration of many tasks into large grained CodeTask modules can greatly reduce simulator run times. Our experiments have shown that the combination of EC and DSIM can simulate parallel programs running on networks with thousands of processors, roughly 100 times larger than possible using detailed step by step simulators.

References

[1] A. Agarwal, B. Lim, D. Kranz, and J. Kubiatowicz. APRIL: A Processor Architecture for Multiprocessing. *17th Int. Symp. on Computer Arch.*, pages 104–114, May 1990.

[2] J. Archibald and J. Baer. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. *ACM Trans. on Computer Systems*, 4(4):273–298, November 1986.

[3] R. Bisiani and M. Ravishankar. PLUS: A Distributed Shared-Memory System. *17th Int. Symp. on Computer Arch.*, pages 115–124, May 1990.

[4] P. Bitar. A Critique of Trace-Driven Simulation for Shared-Memory Multiprocessors. *Workshop on Cache and Interconnect Architectures in Multiprocessors*, pages 37–52, 1990.

[5] M. Carlton and A. Despain. Multiple-Bus Shared-Memory System: Aquarius Project. *IEEE Computer*, 23(6):80–83, June 1990.

[6] D.R. Cheriton, H.A. Goosen, and P.D. Boyle. Multi-Level Shared Caching Techniques for Scala-

bility in VMP-MC. *16th Int. Symp. on Computer Arch.*, pages 16–24, May 1989.

[7] G.A. Darmohray and E.D. Brooks. A Parallel Gauss Elimination Algorithm with Minimized Barrier Synchronization. TR UCRL-101587, Lawrence Livermore NL, 1989.

[8] H. Davis, S.R. Goldschmidt, and J. Hennessy. Multiprocessor Simulation and Tracing Using Tango. *Int. Conf. on Parallel Processing*, II:99–107, August 1991.

[9] J.R. Goodman and P.J. Woest. The Wisconsin Multicube: A New Large-Scale Cache-Coherent Multiprocessor. *15th Int. Symp. on Computer Arch.*, pages 422–431, May 1988.

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

[11] D. Lenoski, K. Gharachorloo, J. Laudon, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. Design of Scalable Shared-Memory Multiprocessors: The DASH Approach. *Spring COMPCON 90*, 62–67, February 1990.

[12] R.J. Lipton and D.N. Serpanos. Uniform-Cost Communication in Scalable Multiprocessors. *Int. Conf. on Parallel Processing*, 1:429–432, August 1990.

[13] C. Maples and L.D. Wittie. MERLIN: A Superglue for Multicomputer Systems. *Spring COMPCON 90*, pages 73–81, February 1990.

[14] B. Qin, H.A. Sholl, and R.A. Ammar. RTS: A System to Simulate the Real Time Cost Behaviour of Parallel Computations. *Software - Practice and Experience*, 18(10):967–985, October 1988.

[15] L.D. Wittie, G. Hermannsson, and A. Li. Scalable Eagerly Shared Distributed Memory. TR-90/48, SUNY Stony Brook, November 1990.

[16] L.D. Wittie, G. Hermannsson, and A. Li. Eager Sharing for Efficient Massive Parallelism. *Int. Conf. on Parallel Processing*, II:251–255, August 1992.

[17] L.D. Wittie, G. Hermannsson, and A. Li. Evaluation of Distributed Memory Systems for Parallel Numerical Applications. *6th SIAM Conf. on Parallel Processing for Scientific Computing*, 561–568, March 1993.

[18] L.D. Wittie and C. Maples. Merlin: Massively Parallel Heterogenous Computing. *Int. Conf. on Parallel Processing*, 1:142–150, August 1989.

[19] P. Woodbury, A. Wilson, B. Shein, I. Gertner, P.Y. Chen, J. Barttlet, and Z. Aral. Shared Memory Multiprocessors: The Right Approach to Parallel Processing. *Spring COMPCON 89*, 72–80, February 1989.

A An Example of EC Program Output

In Section 4.6, EC code generation issues were discussed briefly. This example illustrates how EC works in more detail The following tiny program adds i to the i-th entry of a shared array.

```
shared float a[100];
varcpu int cpu_id;
varcpunum int  num_cpu;
vardatasize int dim;
main()
{       int i;
        for (i = cpu_id; i< dim; i+=num_cpu)
          a[i]= a[i]+i;
        barrier;
}
```

Command line arguments are used to set network size (*num_cpu*) and data size (*dim*) without re-compiling. Each processor that executes the code has its own unique CPU index as the value of *cpu_id*. The variable i is a control variable, but the array assignment, $a[i] = a[i] + i$, does not need to be evaluated since it is not used to control execution flow. However, both memory references and the plus operation of the assignment must be counted to evaluate performance. This program is translated by EC into:

```
main(argc, argv)
int argc; char *argv[];
{ /*  List of Global variables  */
VarId PRG_cpu_id, PRG_num_cpu, PRG_dim;
ShrRegT PRG_a;
/* Temp Var. List skipped to shorten */
...
/* List of Local Variables in main */
VarId PRG_i;

DSIMInit(argc,argv); /* For parameters */
/*  Global Variable Initialization */
PRG_cpu_id = VDecl("cpu_id",VCpu, 4, 0);
PRG_num_cpu = VDecl("num_cpu",VNcpus,4,0);
```

```
PRG_dim = VDecl("dim",VData, 4, 0);
PRG_a = ShrRegDecl("a", 100);
/*  Temp Var Initialization  skipped */
...
/*  Local Variable Initialization */
PRG_i = VDecl("i",VLocal, 4, 0);
/*   End of Variable Initialization */

/* For-loop start:
for (  Initialize: i=cpu_id;
       Condition: i<dim;
       Body: a[i]=a[i]+i;
       Increment: i+=num_cpu;
For-loop end */
CodeSet("Var",PRG_i,PRG__i_n_d_e_x_0,
  VarXCalcEqual,PRG_i,PRG__i_n_d_e_x_0,
  PRG_cpu_id,PRG__i_n_d_e_x_0,0); /*STM 0*/
CodeSet("Var",
  PRG_t_m_p_v_1,PRG__i_n_d_e_x_0,
  VarVarXCalcLT,PRG_i,PRG__i_n_d_e_x_0,
  PRG_dim,PRG__i_n_d_e_x_0,0);    /*STM 1*/
CodeForF("",                      /*STM 2*/
  PRG__d_u_m_m_y,PRG__i_n_d_e_x_0,
  XCalcNumber,
  PRG__d_u_m_m_y,PRG__i_n_d_e_x_0,
  PRG__d_u_m_m_y,PRG__i_n_d_e_x_0,1,
  VarVarXCalcEQ,
  PRG__d_u_m_m_y,PRG__i_n_d_e_x_0,
  PRG_t_m_p_v_1,PRG__i_n_d_e_x_0,0,
  XCalcNumber,
  PRG__d_u_m_m_y,PRG__i_n_d_e_x_0,
  PRG__d_u_m_m_y,PRG__i_n_d_e_x_0,0);
CodeCheckBusyWait("");           /*STM 3*/
CodeTask("","", shcopy, XCalcNumber,
  PRG__d_u_m_m_y,PRG__i_n_d_e_x_0,
  PRG__d_u_m_m_y,PRG__i_n_d_e_x_0,1,
  1,  /* Floating point ops */
  0,  /* Non floating point ops */
  0,  /* Read only operands */
  1,  /* Read write operands */
  0,  /* Write only operands */
  1,  /* Shared writes */
  PRG_a);                         /*STM 4*/
CodeSet("Var",                    /*STM 5*/
  PRG_i,PRG__i_n_d_e_x_0,
  VarVarXCalcPlus,
  PRG_i,PRG__i_n_d_e_x_0,
  PRG_num_cpu,PRG__i_n_d_e_x_0,0);
CodeSet("Var",                    /*STM 6*/
  PRG_t_m_p_v_2,PRG__i_n_d_e_x_0,
  VarVarXCalcLT,
  PRG_i,PRG__i_n_d_e_x_0,
  PRG_dim,PRG__i_n_d_e_x_0,0);
CodeSet("Var",                     /*STM 7*/
  PRG_t_m_p_v_1,PRG__i_n_d_e_x_0,
  VarXCalcEqual,
  PRG_t_m_p_v_2,PRG__i_n_d_e_x_0,
  PRG_t_m_p_v_2,PRG__i_n_d_e_x_0,0);
CodeForEnd();                      /*STM 8*/
/* barrier; */
CodeBarrier("");                   /*STM 9*/
/*   End Function main    */
CodeEnd("The End");                /*STM 10*/
DSIMMainLoop();  /* Start simulation */
}  /*  end main  */
```

The first part contains function calls that declare and initialize the variables used to follow execution flow during simulation. Some of the function calls have unused portions, for example, *PRG__i_n_d_e_x_0* repeatedly is a dummy variable for an unneeded parameter used for an array index in array variables. In this example, counted references are only to arrays, and all other variables are assumed to be in registers.

Statements 0 (STM 0) and 1 (STM 1) are control computation modules that set variables for the *for*-loop that starts with STM 2 in the code generated by EC. Both help change i, which affects execution flow. STM 0 starts i at *cpu_id*. STM 1 assigns the result of $i < dim$ to a temporary variable, used to determine whether to execute the statements in the body of the *for*-loop.

STM 3 allows the simulator to process instructions for another CPU whenever the loop code is waiting for an external event. The array assignment(STM 4) in the loop body does not affect program execution flow. The CodeTask statement(STM 4) counts all resources used for the assignment: 1 read, 1 shared write, and 1 floating point operation. STMs 5-8 increment loop index i and test for loop termination.