Object Oriented Parallel Architecture Simulator

Sang Gue Oh
School of Computer and Information Science
Syracuse University
sgoh@top.cis.syr.edu

Cheol Min Hwang
School of Computer and Information Science
Syracuse University
cmhwang@top.cis.syr.edu

Alok Choudhary
Electrical and Computer Engineering
Northwestern University
choudhar@ece.nwu.edu

Abstract

In this paper, we introduce a general purpose **Object Oriented Simulation Environment** *which enables the study of different architectural configurations for servers. The simulation environment has been built on the top of AWESIME[1] which provides a process oriented discrete event simulation environment. On the top of AWESIME, the simulator provides the base class definitions for major components required to build large scale high-performance computer systems, and more advanced model of any particular component can be derived from these base classes and easily replaced with old one. This feature enables rapid evaluation of alternatives in simulating different architectures. In addition, this design allows studying a component of interest in various degrees of detail while keeping the other components at a coarse-level for a faster simulation time.*

1. Introduction

Over the last few years, scalability of high-performance system has been demonstrated in terms of processing power and communication technology (both in terms of bandwidth and latency). The debate on cost-effectiveness is still underway. The trend seems to be to use off-the-shelf technology for processing nodes to leverage their availability at low cost. Similar debate is ongoing for using off-the-shelf interconnect technology (e.g., ATM) for high-performance server systems.

In this paper, we introduce a general purpose **Object Oriented Simulation Environment** which enables the study of different architectural configurations for servers. The simulator has been built on the top of AWESIME which provides a process oriented discrete event simulation environment. On the top of AWESIME, the simulator provides the base class definitions for major components required to build large scale high-performance computer systems, and more advanced model of any particular component can be derived from these base classes and easily replaced with old one. This feature enables rapid evaluation of alternatives in simulating different architectures. In addition, This design allows studying a component of interest in various degrees of detail while keeping the other components at a coarse-level for a faster simulation time.

The section 2 discusses the design and infrastructure of the simulator, and we try to reason the validity of the simulator in the section 3. Then, we summarize the current status of the simulator and discuss the future work in the last section.

2 A General Purpose Object Oriented Architecture Simulator

The main objectives of the simulator is to design and develop an *object oriented simulation environment* which is general enough to study different architectural configurations for server systems. Another objectives of the simulator is to develop and provide detailed workload models for various application domains such as frequently used *scientific computing*, *on-line transaction processing*, *video-on-demand*, *decision support system*, and *world wide web*. The simulation environment has been chosen to be *object oriented paradigm*, and the nature of this paradigm allows easy replacement of one module with another without affecting the other part of simulation environment enabling easy and rapid evaluation of alternatives. Another design principle is

[1] A Widely Extensible SIMulation Environment by University of Colorado

Proceedings of the High-Performance Computing on the Information Superhighway, HPC-Asia '97
0-8186-7901-8/97 $10.00 © 1997 IEEE

to make it *general* enough to study any single component of the system independently.

2.1 AWESIME Library

AWESIME is a library of C++ classes. Objects provided by AWESIME are building blocks for constructing process oriented discrete event simulation. AWESIME is intended to be extensible and its class hierarchy provides a set of functions common to all classes. Furthermore, the class hierarchy is broken into *conceptual subclasses* or *container classes* that provide a common set of functions for a particular concept in a simulation system. AWESIME is designed for architectures with a shared address space running some variant of the Unix operating system. A prime goal in the design of AWESIME is portability, necessitating certain assumptions in the underlying memory architecture of the host machine. The AWESIME library creates multiple Unix processes that are ideally mapped into individual processors. The child processors inherit the memory and file configuration of the initial process. Within AWESIME, each Unix process is termed a **CPU**.[3] As a whole, AWESIME library provides a pseudo parallel programming environment on a single processor system. For more information on AWESIME, readers may refer to *A User Guide to AWESIME* found at the web site *http://esl.cs.colorado.edu/AWEDUDE/awedude.html*.

2.2 Parallel Architecture Simulator

The simulator is a group of C++ class definitions provided for general architecture simulation on the top of AWESIME library. Due to the nature of object oriented paradigm, it is relatively easy to evaluate different architectural configurations. One such example is evaluating different network topologies. While leaving nodes configurations as they are, users can simulate different topologies such as mesh, ring, hypercube, and so on simply by replacing different network models. This feature also allows users to study a component of interest in various degrees of details while keeping the other components at a coarse-level for a faster simulation time. For instance, users can choose to simulate mesh network topology using *hop-by-hop* transfer methodology which naturally adopts the current traffic congestion and therefore produces more accurate results. This method may, then, take long simulation time since each packet of network traffic is an active thread which requires expensive context switching to simulate. Alternatively, users can choose to simulate the same network topology using analytic model which computes network latency from the number of hops and the given congestion parameter.[1]. This method gives faster simulation time, and therefore it might not be a bad choice to use this model when network is not the main component of interest. In the following, the major objects defined in the simulator are briefly introduced. For more details, refer to *http://www.ece.nwu.edu/~goh*.

- **System Object**
  System object is basically a container class definition. Its structure imitates a frame of typical MPP system with a number of different slots for CPUs, disks, buses, memories, and network ports. Due to the nature of software, then, the number of slots for each component is not limited to a certain fixed number making this object scalable. With this infrastructure, it is possible to build almost any type of existing stand-alone MPP systems onto it.

- **Node Object**
  Node object is another container class which is like a small version of System object in a sense that Node object itself can be a multiprocessor system. The basic difference between System and Node objects is that Node object has been fixed to have its own bus system for interconnection method while System object can have a variety of different interconnection network topologies. In both objects, local memory, shared memory, and I/O subsystem are all optional depending on what kind of system to be built.

- **IOSys Object**
  As the role of I/O subsystem becomes more and more important, its architecture also becomes more and more complicated everyday, and so it is almost like a computing node with its own powerful processor and large buffer space or even big cache memory. In these days, it is common to dedicate a conventional computing processor as a separate I/O processor to handle incoming I/O requests for optimum performance. This object is designed to be able to model such a complicated I/O subsystem.

- **Disk Object**
  Disk object itself becomes very intelligent in a sense that it can perform various optimization tasks such as disk scheduling for minimum seek latency. The new trend in computer technology even allows peripheral devices such as disks to be directly connected to interconnection network without having controlled by a certain host node.[2] This disk object is designed so that it accommodates even this new technology. Raw disk system may be single disk environment or disk array. One of our disk mechanism objects is based on an analytic model described in [6, 7]. The simulator also provide another disk mechanism object which is based on a set of empirical data obtained by tracing

Proceedings of the High-Performance Computing on the Information Superhighway, HPC-Asia '97
0-8186-7901-8/97 $10.00 © 1997 IEEE

actual system's I/O request pattern for a long period of time.

- **Network Object**
  Network object consists of a number of interface units or switches and connections between them. The connection topology depends on each different models. Every well-known network topologies are to be modeled within the simulator. The simulator currently supports topologies such as mesh, ring, hypercube, and Ethernet connections.

- **IU Object**
  Interface Unit includes network interface units (or ports) in each node or disk and switches in network such as MRC. Each different subclass of IU object employs different communication protocols.

- **OSKernel Object**
  Due to the difficulty in modeling software behavior, the functionality of OSKernel object has been kept at minimum level; it currently handles the basic interprocessor communication such as *send* and *recv*. Collective interprocessor communications such as *broadcast* and *accumulate* are separately provided by **Library** object.

- **FileSys Object**
  FileSys Object handles file system related calls such as *read* and *write*. It currently implements a basic striping parallel file system and distributed file system.

- **Message Object**
  Message object carries actual data through the network and each smallest communication unit is represented as an active thread. This thread-based simulation is the main feature that enables the simulator to achieve pseudo-parallelism on a single processor environment.

- **Workload Object**
  Different application domains have different characteristics of workload from each others, and these differences may have a great deal of impact on server machine design in terms of both hardware and software architecture. The simulator provides workload models for each major application area which demonstrates a unique characteristics such as *scientific application*, *OLTP*, *VOD*, and *WWW*.

- **Processor and Memory Object**
  Both Processor and Memory objects are currently primitive in terms of their functionalities; they just consume the given amount of simulation time. Their major influence over simulation is implementing mutually exclusive access on shared critical sections.

Routing Technique	Analytic Model	Simulator
Basic Switching	4543	4380.86
Store-and-forward	2003	1958.53
Cut-though	1625	1621.76

Table 1. Basic Communication Operation
(Time Unit : clock tick, Message Size : 500 bytes)

(Note: These experiments were done to validate the simulator using a single message without any interference from any other messages.)

Data Size	Analytic Model	Simulator
1 KB	1148269.70	1149119.18
2 KB	1172164.62	1173548.27
4 KB	1219954.47	1221408.40
8 KB	1169773.70	1168452.97

Table 2. Local I/O Operation
(Time Unit : clock tick)

(Note: Like basic communication experiments, these experiments were also performed within an isolated environment for the purpose of validation.)

3 Validation

In this section, we try to reason out the validity of the simulator. First, we isolated and analyzed two major lines of operations: basic interprocessor communication and local I/O. Then, we simulated these operations on our modeled system which consists of 16 nodes connected via 4×4 mesh interconnection network. Due to the space restriction, however, we omit these sections and presents only their results in the table 1 and 2. The analysis of these operations basically adopted techniques described in [5]. Assuming the soundness of these two basic lines of operation, then, we simulated more realistic workload, matrix multiplication. We first asymptotically analyzed GAXPY algorithm for matrix multiplication and simulated matrix multiplication workload based on GAXPY algorithm. The following sections present the analysis of the algorithm and the simulation result. We will try to clarify all the assumptions made for each operation in order to avoid possible misinterpretation.

Proceedings of the High-Performance Computing on the Information Superhighway, HPC-Asia '97
0-8186-7901-8/97 $10.00 © 1997 IEEE

3.1 Simple Analysis of GAXPY Algorithm for Matrix Multiplication

Let A, B, and C be $n \times n$ matrices such that $C = A \times B$. Then, the **GAXPY** algorithm for computing $C = A \times B$ is

$$c_j = \sum_{k=1}^{n} b_{kj} a_k, j = 1 : n \quad (1)$$

The algorithm indicates that, in order to compute the j^{th} column of C, the j^{th} column of B and all columns of A are needed. Then, assuming n be a multiple of p, equation 1 can be rewritten as a sum of p partial sums as follows

$$\begin{aligned} c_j &= \sum_{k=1}^{\frac{n}{p}} b_{kj} a_k + \sum_{k=\frac{n}{p}+1}^{\frac{2n}{p}} b_{kj} a_k + ... \\ &+ \sum_{k=((p-1)\times\frac{n}{p})+1}^{n} b_{kj} a_k, j = 1 : n \end{aligned} \quad (2)$$

Each partial sum $\sum_{kj} a_k$ returns an intermediate vector, and each vector is a linear combination of $\frac{n}{p}$ columns of A and $\frac{n}{p}$ elements of a column j of B. These intermediate vectors are then added to produce the j^{th} column of matrix C. The entire matrix C can be computed by repeating this process n times. Assuming that p processors be available, then, these p partial sums can be computed in parallel. The following pseudo code represents the partial sum process on each processor:

```
do j = 1 to n
   do i = 1 to n/p
      do k = 1 to n
         temp(j,i) = temp(j,i) + a(k,i) × b(i,j)   (a)
      end do
   end do
   global sum on temp(n,j)                          (b)
end do
```

Let F be the cost for one floating point operation and W be the cost for memory access of one word. Assuming no I/O operation(That is, incore computation), we can estimate the performance of the algorithm by approximating each operation. In step (a), the computation requires one floating point multiplication, one floating point addition, and seven one-word memory accesses. Hence, through out the computation, the step (a) costs $\frac{n^3(2F+7W)}{p}$ time units. In step (b), an interprocessor communication operation takes place during which a column of size n with partial sum is all-to-all broadcasted. Let the cost of this communication be the function of m and p where m is the size of matrix in number of bytes. After the all-to-all broadcast, there will be $n \times p$ floating point additions to produce a column of resulting matrix C, and this step is repeated for each column of matrix C. Hence, this step costs $nF(m,p) + pn^2(F+W)$ time units overall.

$$\begin{aligned} T_G &= n^2 F(\frac{2n}{p} + p) + n^2 W(\frac{4n}{p} + p) + \\ &\quad nF(m,p) \end{aligned} \quad (3)$$

Now, the broadcasting algorithm depends on the underlying network topology. For this analysis, we assume $q \times r$ 2-D mesh interconnection network with end-around connection where $p = q \times r$, and a simple dimensional broadcasting algorithm described in [5]. In this algorithm, all-to-all broadcasting takes place in two phases. In the first phase, each node broadcasts the message of size m along the X dimension taking $(q-1)(t_{mp} + t_{ce} + (t_h + m/bw) \times 2 + t_i)$ time units, and then the message of size qm along the Y dimension in the second phase taking $(r-1)(t_{mp} + t_{ce} + (t_h + qm/bw) \times 2 + t_i)$ time units.[5] We now have:

$$\begin{aligned} F(m,p) &= (q+r-2)(t_{ce} + 2t_h) + \\ &\quad (qr-1)(\frac{m}{t_{mp}} + \frac{m}{t_i} + \frac{2m}{bw}) \end{aligned} \quad (4)$$

and we can rewrite the equation 3 as follows:

$$\begin{aligned} T_G &= n^2 F(\frac{2n}{p} + p) + n^2 W(\frac{4n}{p} + p) + \\ &\quad n(q+r-2)(t_{ce} + 2t_h) + \\ &\quad n(qr-1)(\frac{m}{t_{mp}} + \frac{m}{t_i} + \frac{2m}{bw}) \end{aligned} \quad (5)$$

3.2 Simulation of GAXPY Algorithm for Matrix Multiplication

We have developed a simple workload for matrix multiplication based on GAXPY algorithm described in the previous section and run simulations to see if the simulator behaves as analyzed. In this simulation, we have simplified the cost of floating point operation and memory access so that these operations have constant costs.[2]

We have varied the number of processors from 4 to 32 nodes and the sizes of matrix that we have simulated are 256 and 512×512. For matrix multiplication of size 256, the size of message broadcasted is $256 \times 4 = 1024$ bytes and $512 \times 4 = 2048$ bytes for matrix multiplication of size 512×512. The result of these simulations and analysis are

[2]Due to the space restriction, we have omitted some sections which include terms and parameter set. For the extended paper, send mail to *sgoh@top.cis.syr.edu.*

Proceedings of the High-Performance Computing on the Information Superhighway, HPC-Asia '97
0-8186-7901-8/97 $10.00 © 1997 IEEE

Size	4 nodes	8 nodes	16 nodes	32 nodes
256 × 256	1347.55	828.64	734.40	1017.78
512 × 512	10423.11	5830.61	4195.10	4698.95

Table 3. Matrix Multiplication Result
(Analyzed Estimation, Time unit = msec)

Size	4 nodes	8 nodes	16 nodes	32 nodes
256 × 256	1772.21	1038.36	833.74	1056.10
512 × 512	13799.20	7508.10	5011.63	5061.69

Table 4. Matrix Multiplication Result
(Simulation Result, Time unit = msec)

shown in table 3 and corresponding analytic estimation in table 4. The graph representations of corresponding results are depicted in figure 1 and figure 2, respectively.

There are differences between analyzed estimation and simulation result shown in the table 3 and table 4. This is mainly because the GAXPY algorithm for matrix multiplication is asymptotically analyzed considering the costs of computations within the inner most loop while other auxiliary computations are also implemented in the simulation workload. The scale of this difference then depends upon the set of parameters fed into the simulator and, since the cost of floating point operation assumed is relatively small compared to that of other operations, the difference in this simulation is relatively large.

Up to 16 processors, the turnaround time of matrix multiplication decreases as the number of processors increases; the cost of matrix multiplication with 32 processors is then

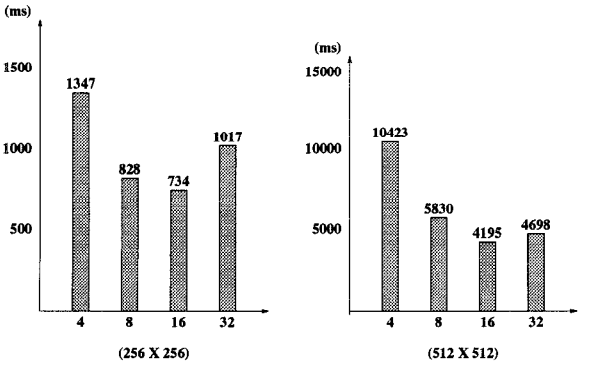

Figure 1. Matrix Multiplication Result
(Analyzed Estimation, Time unit = msec)

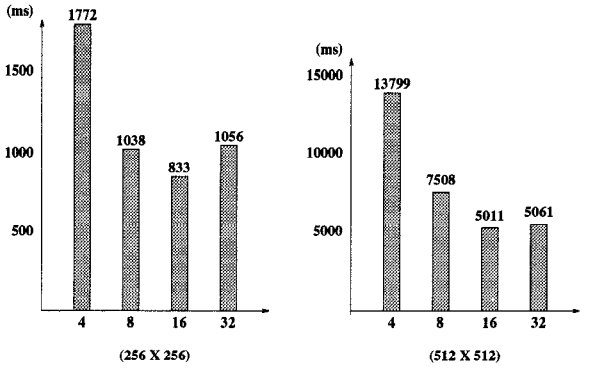

Figure 2. Matrix Multiplication Result
(Simulation Result, Time unit = msec)

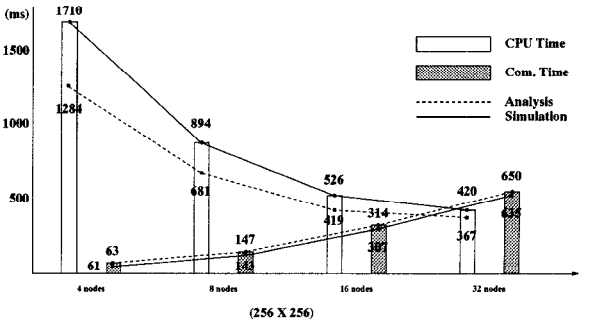

Figure 3. Comparison between Computation and Communication Time
(Matrix Size = 256 × 256, Time unit = msec)

higher than that with 16 processors. This is because the cost of interprocessor communication increases as the number of processors increases and there is a saturation point somewhere in between 16 processor configuration and 32 processor. The costs of separate computation and communication are shown in the table 5 and table 6.

Except for the differences in absolute figures, then, we can observe the matching patterns between analyzed estimation and simulation result in terms of their ratios. The graph in the figure 4 takes the computation time on 4 processor configuration as the unit time and shows the reduction ratios in computation time as the number of processors increases. According to analyzed estimation, 46.9% of reduction in computation time is expected on 8 processor configuration, 67.4% on 16 processors, and 71.4% on 32 processors for matrix of size 256 × 256, and 48.5%, 71.1%, and 79.4% for matrix of size 512 × 512. The reason why the reduction ratio is not 50% when the number of processors is doubled is of course due to the portion of computation which can not

Proceedings of the High-Performance Computing on the Information Superhighway, HPC-Asia '97
0-8186-7901-8/97 $10.00 © 1997 IEEE

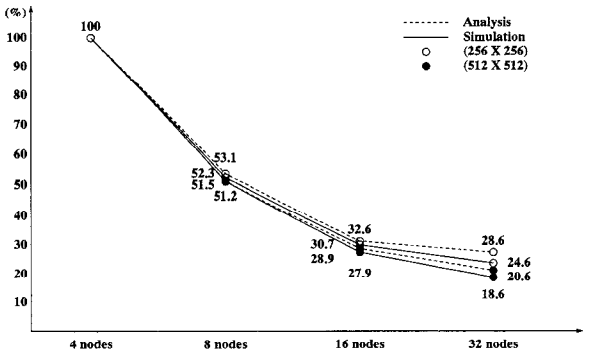

Figure 4. Reduction Ratio in Computation Time

Size	4 nodes		8 nodes	
	CPU	COM	CPU	COM
256 × 256	1284.51	63.05	681.57	147.07
512 × 512	10171.19	251.92	5242.88	587.73
	16 nodes		32 nodes	
256 × 256	419.43	314.97	367.00	650.78
512 × 512	2936.01	1259.09	2097.15	2601.80

Table 5. Matrix Multiplication Result
(Analyzed Estimation, Time unit = msec
CPU = CPU Time, COM = Communication Time)

be parallelized, and this sequential portion of computation becomes proportionally larger in its ratio as the number of processors increases. The corresponding reduction ratios in simulation are 47.7%, 69.3%, and 75.4% for the matrix of size 256 × 256 and 48.8%, 72.1%, and 81.4% for the matrix of size 512 × 512. Hence, in terms of parallelization ratio, simulation results demonstrate better performance. This is because, in simulated workload model, there is more computation which has not been considered into analytic model, and most of this additional computation has also been parallelized. The degree of parallelization of course is not the issue at this point. With comparisons shown in figure 4, then, we claims that the simulator behaves as intended and its functionality is asymptotically sound. Provided that a proper set of parameters be fed into it, therefore, we believe that the simulator produces the output as expected with an acceptable deviation.

4 Conclusion and Future Work

The framework of the simulator has been done so that the simulator is now functional. In many aspects, however,

Size	4 nodes		8 nodes	
	CPU	COM	CPU	COM
256 × 256	1710.62	61.59	894.70	143.67
512 × 512	13553.11	246.09	6933.97	547.13
	16 nodes		32 nodes	
256 × 256	526.06	307.69	420.38	635.72
512 × 512	3781.69	1229.94	2520.12	2541.57

Table 6. Matrix Multiplication Result
(Simulation Result, Time unit = msec
CPU = CPU Time, COM = Communication Time)

it still needs finer tuning. Above all, the granularity of most models is relatively coarse; finer-grain models are to be provided as the project moves onto its next phase. During the second phase of the project, then, we will focus on the study of the architectural configuration for the server machines based on the simulator that we have built while keeping on turning it. Especially, we will develop more workload models such as **OLTP**, **VOD**, and WWW and study the impact of these workloads on the server machines with different architectural configurations. In order to do these studies, collecting and providing empirical parameter sets are the most fundamental and important step. As the consequence of these studies, we do hope that we will be able to address a feasible and concrete solution for the design of the server machines.

References

[1] A. Agarwal. Limits on interconnection network performance. *IEEE Transactions on Parallel and Distributed Systems*, 2(4), October 1991.
[2] G. A. Gibson. A case for network-attached secure disks, 1996.
[3] D. C. Grunwald. *User's Guide to AWESIME-II*, 1991.
[4] K. Hwang. *Advanced Computer Architecture*. McGraw-Hill, Inc., 1993.
[5] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. The Benjamin/Cummings Publishing Company, Inc., 1994.
[6] E. K. Lee. *Performance modeling and analysis of disk arrays*. PhD thesis, Univ. of California at Berkeley, 1993.
[7] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *IEEE Computer*, March 1994.
[8] H. J. Watson and J. H. Blackstonee, Jr. *Computer Simulation*. John Wiley & Sons, Inc., 1989.

Proceedings of the High-Performance Computing on the Information Superhighway, HPC-Asia '97
0-8186-7901-8/97 $10.00 © 1997 IEEE