Experiments with the CHIME Parallel Processing System

Anjaneya R. Chagam[1], Partha Dasgupta[2], Rajkumar Khandelwal[1],
Shashi P. Reddy[1] and Shantanu Sardesai[3]

[1]Intel Corporation, Chandler, AZ, USA and Penang, Malaysia
[2]Arizona State University, Tempe, AZ, USA. http://cactus.eas.asu.edu/partha
[3]Microsoft Corporation Redmond, WA, USA

Abstract: This paper presents the results from running five experiments with the Chime Parallel Processing System. The Chime System is an implementation of the CC++ programming language (parallel part) on a network of computers. Chime offers ease of programming, shared memory, fault tolerance, load balancing and the ability to nest parallel computations. The system has performance comparable with most parallel processing environments. The experiments include a performance experiment (to measure Chime overhead), a load balancing experiment (to show even balancing of work between slow and fast machines), a fault tolerance experiment (to show the effects of multiple machine failures), a recursion experiment (to show how programs can use nesting and recursion) and a fine-grain experiment (to show the viability of executions with fine grain computations.

1. Introduction

This paper describes a series of experiments to test the implementation, features and performance of a parallel processing system called Chime. The experiments include runs of various scientific applications. Chime is a system developed at Arizona State University [1, 2] for running parallel processing applications on a Network Of Workstations (the NOW approach).

Chime is a full implementation of the parallel part of Compositional C++ (or CC++) [3], running on Windows NT. CC++ is a language developed at Caltech and is essentially two languages in one. It has two distinct subparts – a distributed programming language designed for NOW environments and a parallel programming language designed for shared memory multiprocessor environments. While shared memory multiprocessors are very good platforms for parallel processing, they are significantly costlier than systems composed of multiple separate computers. Parallel CC++ is an exceptionally good language, but it was not designed to run on NOWs. Chime solves this problem, by implementing parallel CC++ on the NOW architecture.

Chime uses a set of innovative techniques called “two-phase idempotent execution strategy” [4], “distributed cactus stacks” [1], “eager scheduling”[4], “dependency preserving execution” [2] and the well-known technique called “distributed shared memory” [5] to implement parallel C++. In addition, it has the extra features of load

balancing, fault tolerance and high performance. Chime is the first (and of this writing, the only) parallel processing system that provides the above features, coupled with nested parallelism, recursive parallelism and synchronization (these are features of parallel CC++). This paper described the implementation of Chime, in brief and presents details on the experiments with Chime.

2. Related Work

Shared memory parallel processing in distributed systems is limited to a handful of Distributed Shared Memory (DSM) systems that provide quite similar functions (Munin [6], Midway [7], Quarks [8], TreadMarks [9]). DSM systems care categorized by the type of memory consistency they provide. DSM systems do not provide a uniform view of memory i.e. some global memory is shared and some are not. In addition, the parallel tasks execute in an isolated context; i.e. they do not have access to variables defined in the parent's context. In addition, a parallel task cannot call a function that has an embedded parallel step (nesting of parallelism is not allowed).

The Calypso system [4, 17, 18] adds fault tolerance and load balancing to the DSM concept, but suffers from the lack of nesting and synchronization (except barrier synchronization). Chime is an extension to Calypso and absolves these shortcomings.

A plethora of programming systems for NOW based systems exist, that uses the message-passing technique. Two well-known systems are PVM [10] and MPI [11]. Fault tolerance and load balancing has been addressed, in the context of parallel processing by many researchers, a few examples are Persistent Linda [12], MPVM [13], Dynamic PVM [14] Piranha [15] and Dome [16]. The techniques used in most of these systems are quite different from ours and often add significant overhead for the facilities such as fault tolerance.

Most working implementations are built for the Unix platform (including Linux). Some have Windows NT implementations, but they are buggy at best. In our experience, we have not been able to make any non-trivial applications work correctly with these systems on the Windows platform. For this reason, we are unable to provide comparative performance tests.

3. Chime Features

Shared memory multiprocessors are the best platform for writing parallel programs, from a programmer’s point of view. These platforms support a variety of parallel processing languages (including CC++) which provide programmer-friendly constructs for expressing shared data, parallelism, synchronization and so on. However the cost and lack of scalability and upgradability of shared memory multiprocessor machines make them a less than perfect platform. Distributed Shared Memory (DSM) has been promoted as the solution that makes a network of computers look like a shared memory machine. This approach is supposedly more natural than the message passing method used in PVM and MPI. However, most programmers find this is not the case. The shared memory in DSM systems does not have the same access and

sharing semantics as shared memory in shared memory multi-processors. For example, only a designated part of the process address space is shared, linguistic notions of global and local variables do not work intuitively, parallel functions cannot be nested and so on.

As stated before, Chime provides a multiprocessor-like shared memory programming model on network of workstations, along with automatic fault-tolerance and load balancing. Some of the salient features of the Chime system are:

1. Complete implementation of the shared memory part of the CC++ language. Hence programming with Chime is easy, elegant and highly readable.
2. Support for nested parallelism (i.e. nested barriers including recursion) and synchronization. For example, a parallel task can spawn more parallel tasks and tasks can synchronize amongst each other.
3. Consistent memory model, i.e. the global memory is shared and all descendants share the local memory of a parent task (the descendants execute in parallel).
4. Machines may join the computation at any point in time (speeding up the computation) or leave or crash at any point (slowdowns will occur).
5. Faster machines do more work than slower machines, and the load of the machines can be varied dynamically (load balancing).

In fact, there is very little overhead associated with these features, over the cost of providing DSM. This is a documented feature (see section 6.1) that Chime shares with its predecessor Calypso [4]. Chime runs on Windows NT and the released version can be downloaded from http://milan.eas.asu.edu.

3.1 Chime Programming Example

Chime provides a programming interface that is identical to the parallel part of Compositional C++ (or CC++) language Consider the following parallel CC++ program:

```
#include <iostream.h>
#include "chime.h"
#define N 1024

int GlobalArray[N];
void AssignArray(int from, to){
        if (from != to)
             par {
                   AssignArray(from, (from+to)/2);
                   AssignArray((from+to)/2 + 1, to);
             }
       else GlobalArray[from] = 0;
}
int main(int argc, char *argv[]) {
       AssignArray (0, N-1)
}
```

The above program defines a global (shared) array called GlobalArray, containing 1024 integers. Then it assigns the global array using a recursive parallel function called AssignArray. The AssignArray function uses a "par" statement. The par statement executes the list of statements within its scope in parallel, thus calling two in-

stances of AssignArray in parallel. Each instance calls two more instances, and this recursion stops when 1024 leaf instances are running.

4. Chime Technologies

The implementation of Chime borrows some techniques used in an earlier system called Calypso and adds a number of newer mechanisms. The primary mechanisms used in Chime are:

Eager Scheduling: In Chime, the number of parallel threads running a parallel application can change dynamically and is larger than the number of processors used. Each processor runs a "worker" process, and one designated processor runs the manager. A worker contacts the manager and picks up one thread and when it finishes, it requests the next thread. Threads are assigned from the pool of uncompleted jobs. This technology provides load balancing (faster workers do more work) and fault tolerance (failed workers do not tie up the system) using the same technique.

Two Phase Idempotent Execution Strategy (TIES): Since there is the possibility of multiple workers running the same thread, the execution of each thread must be idempotent. The idempotence is achieved by coupling eager scheduling with an atomic memory update facility implemented by Calypso DSM (see below).

Calypso DSM: This is a variant of the well-known RC-DSM (Release Consistent Distributed Shared Memory) technique. RC-DSM is modified so that the return of pages are postponed to the end of the thread, and the manager buffers all the returned pages and updates them in an atomic fashion and then marks the thread as completed. This ensures correct execution even when threads fail at arbitrary points [17].

Dependency Preserving Execution: Threads can create threads; threads can synchronize with other threads. This can cause unmanageable problems when multiple workers are executing the same thread or when threads fail after creating new threads (or fail after reaching a synch point). Dependency Preserving Execution solves this problem. Each time a thread created nested threads, it informs the manger of the new threads and the old thread mutates into a new thread itself. Similarly at synchronization points, the manger is informed, and a mutation step is performed. The complete description of this mechanism is beyond the scope of this paper and is described in [2].

Distributed Cactus Stack: A data structure that replicates the application stack amongst all machines to ensure correct nesting or parallel threads and scooping of variable local to functions [1].

5. Experiments with Chime

We now describe a set of five experiments using various scientific applications to determine its performance and behaviors on a range of features. The five experiments shown below are the performance experiment, the load balancing experiment, the fault

tolerance experiment, the recursion experiment and the fine grain execution experiment. Experiments were conducted at different points in time, at different locations by different people, hence all the equipment used are not the same (except that Intel machines with *Windows NT 4.0*, connected by a *100Mbps Ethernet* was used for all experiments). The systems used are stated along with the discussion of each experiment.

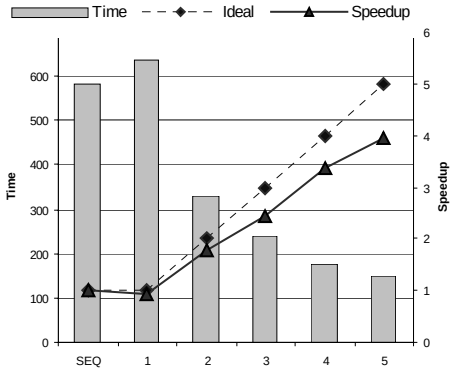

Fig. 1: Performance Experiment

5.1 Performance Experiment

The performance experiments used several matrix multiply and ray-tracing programs, and both yielded similar results. We show the results of a ray-tracing program below using Pentium Pro 200 processors. The first step is to write the program in sequential C++ and measure its execution time. Then a parallel version is written in CC++ and run with Chime on a variable number of processors (from 1 to 5) and the speedups are calculated in respect to the sequential program. The results are shown in Figure 1.

Note that the single processor execution under chime is about 9% poorer than the sequential program and the execution speed scales with addition of processors – the degradation in performance is at most 21%. This makes Chime competitive with most parallel processing systems for NOWs even though Chime has significantly better features. This experiment shows that the overhead of Chime is quite small, in spite of its rich set of features. We have been unable to run (in spite of extensive attempts) any complicated programs with Windows NT implementations of systems such as PVM and MPI and hence cannot provide comparative performance numbers.

5.2 Load Balancing Experiment

The load balancing experiment involves the same ray-tracing program as above, but using machines of different speeds to run the parallel application. In many parallel-processing systems, the slowest machine(s) dictate performance; that is, fast machines are held up for the slow machines to finish the work allocated to them. Chime, does it differently. The application was executed on 4 slow machines (P-133, 64MB) and then a fast machine (P-200, 64MB) replaced one slow machine. This caused an increase in speed. Replacing more slow machines with fast machines kept the trend.

We calculate an “ideal” speedup of the program, as follows. Suppose a computation runs for T seconds on n machines, M_1, M_2, …, M_n. Machine M_i has a performance index of p_i and is available for t_i seconds. (Performance index is the relative speed of a

machine normalized to some reference.) Then the maximum theoretical speedup that may be achieved is:

Ideal speedup
= number of equivalent machines

$= \sum_{i=1:n}(p_i * t_i) / T$

The performance index of a P-133 was set to 1 and the P-200 was measured to be 1.96. Note that the load balancing experiment shows that the actual speedup is close to the ideal speedup (within 22%) and the load is balanced well among slow and fast machines.

5.3 Fault Tolerance Experiment

The "fault tolerance" experiment, using the ray tracing program, shows the ability of Chime to dynamically handle failures as well as allowing new machines to join the computation. For this test up to four P-200 machines were used. Of these machines, one was a stable machine, and the rest were transient machines. The transient machines worked as follows:

> *Transient machine: After 120 seconds into the computation, the transient machine joins the computation and then, after another 120 seconds, fails (without warning, i.e. crashes).*

Figure 3 shows the effect of transient machines. The actual speedups and ideal speedups were computed according to the formula described earlier. Note that the ideal speedup measure takes into account the full power of the transient machines during the time they are up whether they are useful to the computation or not. The experimental results show that the transient machines do contribute to the computation. Note that the transient machines end their participation by crashing. Hence whatever they were running at that point is lost. Such

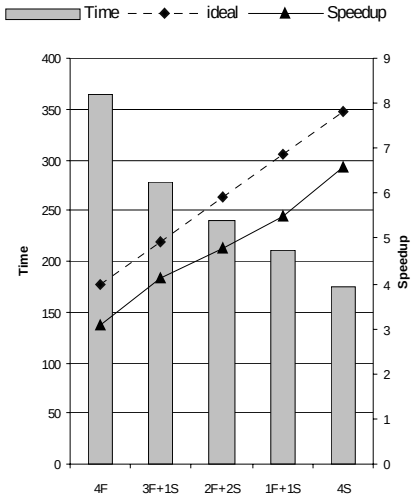

Fig. 2: Load Balancing Experiment

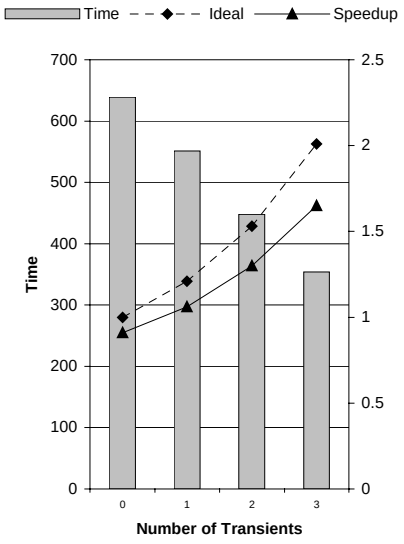

Fig. 3: Fault Tolerance Experiment

crashes do not affect the correct execution of the program under Chime.

In fact this experiment shows the real power of Chime. The system handles load balancing and fault tolerance, with no additional overhead. The real speedups are close to the ideal speedups. In cases where machines come and go, the failure tolerance features of Chime actually provide more performance than an equivalent, non-fault-tolerant system.

5.4 Nesting and Recursion Experiment

The nesting and recursion experiment is a test of Chime's ability to handle nested parallelism, especially in the case the program is recursive. We use a variant of the Fast Fourier Transform (FFT) algorithm for this experiment. This variant is called the Iterative FFT and has a significant computational complexity and hence scope for parallization. To calculate the Fourier Transform of an N-vector, we first compute the Fourier transform of two halves of the vector and then combine the results. The exact details of the FFT algorithm and its complexity analysis are omitted due to the space constraints.

This recursive program is written by writing a subroutine called ComputeFFT( ). This subroutine accepts the input vector size and the vector and then splits it into two parts and calls itself recursively and does it twice. Each invocation of the recursive call runs in parallel. Thus the program starts as one thread and then splits into two and then splits to 4 and so on, till the leaf nodes compute the FFT of 2 elements. For a data size of 32K (215) elements the recursion tree is 15 deep, the number of logical threads generated, is about 32,000.

The following pseudo code illustrates the core of the parallel algorithm.

```
ComputeFFT ( n, vector [size 2n ]){
     if (n==1) compute the FFT
     else  {
                Divide vector into odd and even halves;
                par {   // ** parallel step  **
                   ComputeFFT(n-1, odd-half-of-vector);
                   ComputeFFT(n-1, even-half-of-vector);
              }
            }
     assimilate results;
     return results to caller;
}
```

Note that the resulting parallel program is easy to write, readable and very elegant. This is one of the main appeal of Chime. The above program is run on data sets ranging from size 32K elements (215) to 265K (218) elements. The execution environment was three IBM Intellistation machines with Pentium-III 400 machines with 128MB of memory. The results are summarized below.

Input Data Size	Execution Time (seconds)		Percent speedup on three nodes
	1 node	3 nodes	
2^{15}	17	6	183 %
2^{16}	30	8	275 %
2^{17}	57	13	338 %
2^{18}	108	20	440 %

This experiment shows the ability of Chime to handle nesting and recursion, a feature that makes writing parallel programs simple and is not available on any NOW platform. Note the super linear speedup for the last two executions. This is due to the availability of large memory buffers when using three machines. While one-machine executions do page swapping, the buffering scheme built into Chime allows three machines to buffer the data and avoid having to use the paging disk. This causes a better than expected speedup, on some applications.

5.5 Fine Grain Execution Test

The fine grain test was run using two scientific applications, LU decomposition and computing Eigenvalues. We present the results from the LU Decomposition program. LU decomposition consists of transforming a matrix for a solution of a set of linear equations. The matrix transform yields a matrix whose lower triangle consists of zero. During forward elimination phase to reduce matrix A into L (Lower) and U (Upper) triangular matrices, each worker node works on subset of rows below the current pivot row to reduce them to row echelon form. The code that does this transformation is shown below:

```
for (i = 1 to N) {// N is the size of the array
    max = abs(a[i, i]);  pivot = i;
    for (j = (i+1) to N)  {
         if abs(a[j, i]) > max {
             max = abs(a[j,i];
             swap rows i and j;
         }
   }
   // at this point a[i, i] is the largest
   // element in the column I
   parfor (p = 1 to maxworker) { // ** parallel step **
      find start and end row from values of p and i;
      for j = start to end {
           use the value in a[i,i] and a[j,i]
           to set the element a[j,i] to zero
      }
      // now all elements in column i from
      // row i+1 down, is zero
}
```

As before, the code for the program is simple and readable and the structure of the parallelism is obvious. The program creates a set of maxworker threads on each iteration through the matrix. Depending on the value of N, these threads do varying

amount of work, but the maximum work each thread does is about N simple arithmetic statements. Hence the program dynamically creates a lot of threads, in a sequential fashion and each thread is rather lightweight. Hence it is a test of Chime's ability to do fine-grain processing.

The above program was executed on three Pentium-133 machines with 64MB memory. As shown in the following table, the time to run on three machines, under Chime ranges between 30% and 150% faster. But the gap between single node and three nodes has been gradually reducing as the matrix sizes are increasing. This may be due to the fact that the communication overhead is more than the computational power desired on the virtual nodes.

Input Data Size	Execution Time (seconds)		Percent speedup on three nodes
	1 node	3 nodes	
100x100	46	19	142 %
200x200	94	45	109 %
300x300	156	89	75 %
400x400	242	172	41 %
500x500	349	272	28 %

6. Conclusions

Chime is a highly usable parallel processing platform for running computations of a set of non-dedicated computers on a network. The programming method used in Chime is the CC++ language and hence has all the desirable features of CC++. This papers shows, through a set of experiments how the Chime system can be used for a variety of different types of computations over a set of diverse scientific applications.

7. References

1. D. McLaughlin, S Sardesai and P. Dasgupta, Distributed Cactus Stacks: Runtime Stack-Sharing Support for Distributed Parallel Programs, 1998 Intl. Conf. on Parallel and Distributed Processing Technique and Applications (PDPTA'98), July 13-16, 1998,
2. Shantanu Sardesai, CHIME: A Versatile Distributed Parallel Processing System, Doctoral Dissertation, Arizona State University, Tempe, May 1997.
3. K. M. Chandy and C. Kesselman, CC++: A Declarative Concurrent, Object Oriented Programming Notation, Technical Report, CS-92-01, California Institute of Technology, 1992.
4. A. Baratloo, P. Dasgupta, and Z. M. Kedem. A Novel Software System for Fault Tolerant Parallel Processing on Distributed Platforms. In Proceedings of the 4th IEEE International Symposium on High Performance Distributed Computing, 1995.
5. K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. ACM Transactions on Computer Systems, 7(4):321-359, November 1989.

6. J. B. Carter, Design of the Munin Distributed Shared Memory System, Journal of Parallel and Distributed Computing, 29(2), pp. 219-227, September 1995.
7. B. N. Bershad and M. J. Zekauskas and W. A. SawdonThe Midway Distributed Shared Memory System, Proc. of the 38th IEEE Int'l Computer Conf. (COMPCON Spring'93), pp. 528-537, February 1993.
8. Dilip R. Khandekar. Quarks: Distributed Shared Memory as a Basic Building Block for Complex Parallel and Distributed Systems. Master's Thesis. University of Utah. March 1996
9. C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations, IEEE Computer, December 1995.
10. G. A. Geist and V. S. Sunderam. Network-Based Concurrent Computing on the PVM System. Concurrency: Practice and experience, 4(4):293-311, 1992.
11. W. Gropp, E. Lusk, A. Skjellum. Using MPI Portable Parallel Programming with the Message Passing Interface. MIT Press, 1994, ISBN 0-262-57104-8.
12. Brian Anderson and Dennis Shasha. Persistent Linda: Linda + Transactions + Query Processing. Workshop on Research Directions in High-Level Parallel Programming Languages, Mont Saint-Michel, France June 1991.
13. J. Casas, D. Clark, R. Konuru, S. Otto, R. Prouty, and J. Walpole. MPVM: A Migration Transparent Version of PVM, USENIX, 8(2): pages 171-616, Spring 1995.
14. L. Dikken, F. van der Linden, J. Vesseur, and P. Sloot, Dynamic PVM -- Dynamic Load Balancing on Parallel Systems, Proceedings Volume II: Networking and Tools, pages 273-277. Springer-Verlag, Munich, Germany, 1994.
15. David Gelernter, Marc Jourdenais, and David Kaminsky. Piranha Scheduling: Strategies and Their Implementation. Technical Report 983, Yale University Department of Computer Science, Sept. 1993.
16. E. Seligman and A. Beguelin, High-Level Fault Tolerance in Distributed Programs, Technical Report CMU-CS-94-223, School of Computer, Science, Carnegie Mellon University, December 1994.
17. A. Baratloo, Metacomputing on Commodity Computers, Ph.D. Thesis, Department of Computer Science, New York University, May 1999.
18. P. Dasgupta, Z. M. Kedem, and M. O. Rabin. Parallel Processing on Networks of Workstations: A Fault-Tolerant, High Performance Approach. In Proceedings of the 15th IEEE International Conference on Distributed Computing Systems, 1995.