Preemptive Scheduling for Distributed Systems[1]

Donald McLaughlin [A], *Shantanu Sardesai* [B] *and Partha Dasgupta* [B]

[A] Department of Computer Science and Engineering.
Arizona State University
Tempe AZ 85287-5406
`partha@asu.edu`

[B] Tandem Computers Inc.
19333 Vallco Parkway,
Cupertino, CA 95014-2599
`shantanu.sardesai@tandem.com`

Abstract

Preemptive scheduling is widespread in operating systems and in parallel processing on symmetric multiprocessors. However, in distributed systems it is practically unheard of. Scheduling in distributed systems is an important issue, and has performance impact on parallel processing, load balancing and metacomputing. Non-preemptive scheduling can perform well if the task lengths and processor speeds are known in advance and hence job placement is done intelligently Although obtaining optimal schedules is NP-complete, many good heuristics exist.

In most practical cases, non-preemptive scheduling leads to poor performance due to excessive idle times or due to a long job getting assigned a slow machine. We show how to use preemptive scheduling in distributed systems. Surprisingly, the benefits outweigh the increased overhead. However the implementation of preemptive scheduling is complicated by the need for process migration. This paper presents preemptive scheduling algorithms, their implementation and performance measurement.

Keywords: Scheduling, Parallel Computing, Distributed Computing.

1. Introduction

All multitasking operating systems use preemptive scheduling. Many multiprocessor systems also employ preemptive inter-task scheduling when they run parallel computations. However, preemptive scheduling in distributed systems is rare, if not non-existent.

Consider a cluster of workstations, running a *parallel* application. The application divides itself into a set of tasks. The scheduler assigns these tasks to a set of workstations. The assignment raises a set of issues:

- Is the number of tasks equal to the number of workstations?
- Are all tasks of the same length?
- Are the speeds or processing loads of the workstations identical?
- Does the execution of a task create further tasks?

The answers to most of the questions are expected to be *No*. This leads to a mismatch of tasks to workers causing idle times towards the end of the computations and non-ideal turnaround times. A particular case is when faster machines get short tasks and slower machines get larger tasks. In addition to parallel processing, good scheduling is also important in metacomputing – where a processor farm is used to run general-purpose tasks.

Our research has addressed such problems in a variety of ways [Das97]. We have developed scheduling algorithms, both non-preemptive and preemptive that provide good throughputs in managing distributed computations. This paper describes an implementation of a set of preemptive scheduling algorithms, for a parallel-processing environment, for networked machines, running Windows NT.

2. The Environment

Our work environment consists of the *Chime* [Sar97] parallel processing system. This system supports parallel processing on a network of workstations, with support for *Distributed Shared Memory (DSM), fault tolerance, adaptive parallelism* and *load balancing*. The default scheduler used in Chime is *Eager Scheduling* [BDK95]. Eager Scheduling is similar to a FIFO scheduling algorithm augmented to provide fault tolerance (by assigning uncompleted tasks repeatedly).

Chime provides a programming interface that is based on the *Compositional C++* or CC++ [CK92] language definition. CC++ is a language, developed at CalTech, defined for use in high performance parallel and distributed applications. CC++ allows a programmer to insert parallel statements in a sequential program. The two parallel statements available in CC++ are the **`par`** and **`parfor`** statements.

The **`par`** statement is a compound statement, which runs each individual statement inside the compound statement in parallel. This provides task parallelism. The **`parfor`** statement is identical to a **`for`** statement, but all the iterations of the loop run in parallel. This provides data parallelism. The parallel execution of the statements inside the **`par`** or **`parfor`** statements is run by tasks, and these tasks are the children of the task that ran the **`par`**/**`parfor`** statement (the parent task)

Since these statements can be embedded anywhere in a program, the tasks share the data that is global to the scope of the statements. Hence, all global data is shared. In addition, all sibling tasks share declared in the parents context. Also, parallel statements can be nested inside other parallel statement.

To implement the features of CC++, Chime provides support for shared memory (both global memory and stack memory), and nested parallelism. Chime provides this support when the program is run on a network of computers with no physical shared memory. In addition Chime provides the features such as fault-tolerance and load balancing and inter-thread synchronization. In this paper, we will not

```
par { // all 3 statements
      // run in parallel
      x = 1;
      y = 2;
      z = 3;
 }
```

[1] This research is partially supported by grants from DARPA/AFRL-Rome, Intel Corporation and NSF.

discuss much of the implementation details of Chime, except those that are relevant to scheduling.

3. Distributed Scheduling

The scheduling of parallel computations on a set of processors has been an area of active study [TG89]. Theoretical results show that attaining optimality in a non-preemptive scheduling environment is NP-complete, but preemptive scheduling is polynomial. In practice, this area has been intensely studied in multiprocessor systems, giving rise to policies such as co-scheduling [Ou82], open-shop scheduling [GS76], self scheduling [PK87], processor-affinity scheduling [ML92] and so on.

However, work in distributed scheduling is scanty. Most distributed parallel processing systems use simplistic static scheduling schemes. For example PVM [GBD94] and MPI [GLS94] programmers often split the computations in as many parts as there are machines, and then assign one task to each machine. Most DSM systems [ACD96, BZS93, Car95] also employ static scheduling. This is mainly due to the belief that preemption in distributed systems is too costly for any reasonable benefit.

Some load balancing algorithms use process migration. This can be thought of as preemptive scheduling (see Condor [LLM88]). However, such preemption is activated only when a disparity in load is observed, and the preemption is not used as a strategy for scheduling.

Most work in optimizing span time in scheduling assumes the knowledge of runtime of processes and the speeds of processors. In practical situations, such knowledge is either not available or not accurate. Hence, we have developed scheduling protocols where such knowledge is not needed. Preemption is necessary in situations where such knowledge is not available.

We have found that preemption is actually feasible and even attractive as a part of the scheduling policy - *even if each machine runs one task - and hence loads are always "balanced"*.

4. Preemptive Schemes

Over the last few years we have simulated and implemented a host of preemptive and non-preemptive scheduling algorithms. In this section we present three such algorithms. Some of this work was inspired by [SS96].

The first algorithm is targeted for situations where the number of tasks to be executed is slightly larger than the number of machines available. This algorithm pre-computes a schedule that is optimal in execution time as well as optimal in the number of context switches needed. However it requires that the task execution time be known in advance. Hence, it is not practical for all situations.

The second algorithm is a variation of the well-known round robin algorithm. We call this the *Distributed, Fault-tolerant Round Robin* algorithm. In this algorithm, a set of n tasks is scheduled on m machines, where n is larger than m. Initially, the first m tasks are assigned to the m machines. Then, after a specified amount of time (time quantum), all tasks are preempted and the next m tasks are assigned. This continues in a circular fashion until all tasks are completed.

This scheduler yields a fault-tolerant system as the tasks are preempted from live workers at each time quantum expiry. A task on a worker that does not respond is scheduled from the context as it existed before that task was assigned to the now non-responding worker. New machines are also assimilated at each quanta expiry point. This algorithm has high overhead, but produces good schedules when there are large grained tasks of widely varying lengths.

The third algorithm is the algorithm that performs best over a wide range of circumstances. This is the *Preemptive Task Bunching* algorithm. All n tasks are bunched into m bunches and assigned to the m machines. When a machine finishes its assigned bunch, all the tasks on all other machines are preempted and all the remaining tasks are collected and re-bunched (into m sets) and assigned again. This algorithm works well for both large-grained and fine-grained tasks even when machine speeds and task lengths vary [Mc97].

4.1 Evaluating the Algorithms

The above schemes are well suited for any distributed processing system that supports adaptive parallelism. They are also suited for a processing environment where a lot of independent tasks have to be scheduled on a set of machines on a network.

We conducted many simulations of these and other algorithms. Simulation results on synthetic workloads showed that the algorithms work. However such results are not real and can be often meaningless (due to the way the workload area generated). We are not going to present the simulation results in this paper. We wanted to implement the scheduling algorithms in a real system and test them to see how well they worked.

We implemented these algorithms for scheduling tasks in Chime. As stated before, Chime program can generate a lot of parallel tasks, due to the execution of **`par`** and **`parfor`** statements and these tasks have to be scheduled on a set of machines on the network. Complicating the issue, Chime uses Distributed Shared Memory. It turned out, that preemptive scheduling on a DSM system is not as straightforward as it may seem. We will discuss some of the problems we faced.

The implementation of preemptive scheduling was very time consuming, very tricky and led us to often seemingly insurmountable problems. An asynchronous request for preemption can happen at any time – and since we are not allowed to change a single statement in the source program, written by a programmer, we often had no mechanism to block out preemption requests at inopportune times. However we have been able to implement preemption, mainly through tricky code in the runtime system.

In the next few sections we discuss some implementation details and an overview of performance.

5. The Implementation Architecture

Central to the implementation of any preemptive scheduling protocol is the mechanism of process migration. In our design, however, we avoided actual process migration by implementing a task migration scheme (described later). In our implementation we augmented the Chime system to support preemption and hence the ability to stop an executing task (freezing it) and later restarting it on another machine. Hence preemption followed by a rescheduling is equivalent to a task migration.

The Chime system runs on a manager machine and a set of worker machines. The worker machine runs two threads in a process; one thread runs the actual code of the task and the other thread runs the Chime runtime routines. To freeze a process, we instruct the Chime runtime thread to suspend the thread executing the task and send the context of the task to the manager. Later, to restart the task, we send the context of the frozen task to a worker which then instantiates a thread to run the context.

Chime run on Windows NT 4.0. Thus Chime uses Windows NT features such as kernel threads, events, memory protection and Winsock based communication to achieve scheduling, DSM (Distributed Shared Memory) and synchronization.

5.1 The migration mechanism

Consider a worker process that is executing task T_1 on a machine M_1. Now the scheduler wants to move T_1 to machine M_2. Achieving this goal does not require process migration. If there is a worker already running on M_2, only the context of the application thread (which represents the context of the task in execution) needs to be transferred from M_1 to M_2. Thus, task migration achieves preemptive scheduling in our system.

In the example above, the scheduler (in the manager) first notifies the asynchronous thread (in the worker) on M_1 to freeze T_1. The asynchronous thread then notifies the control thread, which in turn suspends the application thread running T_1. The control thread then retrieves the context of the application thread and packages it up and sends it to the manager. The manger then sends the context of T_1 to the control thread on M_2. The control thread in M_2 sets the context of its application thread to the received context and resumes it. Now T_1 resumes executing on M_2.

The architecture may sound a little complicated, but there are some reasons:

- The need for a control thread is dictated by the fact that NT has no support for signals.
- The need for a dedicated application thread arises from the fact that this thread executes unmodified user code, and hence cannot handle any asynchronous events.
- The need for an asynchronous thread arises from the fact that the inter-thread communication is implemented using NT events. Thus the control thread is not able to receive asynchronous notifications from the manager, which arrive on a network socket.

As a result, the migration mechanism appears to be quite straightforward and implementable. However, upon actually implementation in Windows NT, we found that it barely worked.

5.2 Making Task Migration work

Our first test program revealed a very encouraging fact. Consider the following set of events:

- A thread running inside a process performs a `SuspendThread()` operation on a executing thread. Then it performs a `GetThreadContext()` operation on the suspended thread.
- The context structure returned by NT and the stack contents of the thread are then sent to a different machine, over the network.
- On the other machine, a thread executes a `SetThreadContext()` on a suspended thread with the received context. Then it overwrites the second thread's stack with the received stack contents.
- Then the thread performs a `ResumeThread()` operation on the suspended thread.

The thread that was resumed on the second machine actually continues execution at the point the first thread was suspended. Even thought the context was obtained from a different machine. This was very encouraging as it showed that task migration was possible under Windows NT using the NT-thread API.

Now implementing the full-fledged system seemed possible. Except that the resulting implementation worked intermittently! The problems noted were as follows:

- The system worked well during the debugging process, but did not work under normal execution. This was due to many timing errors and race conditions between the control, application and notification threads.
- After these glitches were fixed (painstakingly), the system often worked well most of the time but occasionally tended to fail for unknown reasons.

The problem, we found was due to the interaction of the DSM mechanisms with the migration mechanisms. It can be summed in the following scenario:

Suppose the executing application thread has begin to handle an exception in order to obtain service for the DSM. At this point, it is suspended by the operating system. If its context is passed on to another machine and resumed, the target thread has a probability of going berserk. Not always, but sometimes; since the target machine would not know about the exception raised on the previous machine. Debuggers were of little help.

The final working solution for task migration was:

- The control thread suspends the application thread.
- Then it checks to see if a flag has been raised indicating that the application thread is handling an exception. If so, then the application thread is handling an exception.
- In such a case, the control thread waits a short period and then retries the migration.
- Else, the task is migrated as described above.

In addition to the above problems, we noticed that sometimes a worker process gets a very large number of memory exceptions, and such exceptions continue to happen throughout the life of the worker. In such a case, migration of that task becomes impossible – the asynchronous thread gets starved and can never get its message to the control thread delivered. Subsequently, the migration request gets ignored and the scheduler gets confused. We fixed this problem by monitoring exceptions coming from a process at the manager level, and issuing migration requests only if the process is not thrashing.

5.3 Testing

Eventually we got the mechanism to work. To test it, we first used a long running Ray-Tracing application. This application generates a picture in parallel and displays the picture as it is computed. It turned out initially, that even if the program did not crash, the picture generated had "glitches". This was easy to spot and was the result of errors in the migration code. After we could fix all the glitches, we switched to a matrix multiply program. Matrix multiply is easier to code than Ray Trace, but a lot harder on the system – the data sets are large and the memory exceptions happen fast and furious. Many bugs that were not apparent using the Ray-Tracing application showed up when matrix multiply was used to test.

When we got correct results on matrix multiply experiments, using a high preemption protocol (Round Robin) we concluded that the scheduling mechanisms worked properly. In addition, performance results obtained (described below) showed us that:

A) The preemption mechanism works correctly.
B) The preemption algorithm is "implementable" and provides increased throughput.
C) The overhead of preemption is not too high – that is it does not overshadow the benefits.

We have also tested the algorithms for fault tolerance (where machines are made to fail) and load balancing - where machines have unequal speeds. Most of these tests were done with the Ray Tracing program. Since Ray Tracing is considered an "easy" program, and some of the tests are not easily put in chart form, we omit these results in this paper. All tests worked well, and failures and slowdowns were well tolerated. We will now elaborate on a few performance tests that show the viability of preemptive scheduling.

6. Performance of Preemptive Scheduling

For testing the performance of the system, or rather the efficacy of the preemptive protocol we used a long running, large matrix multiply program. The matrix used was 2-dimensional, with 1500x1500 elements (needs 9Mbytes per matrix) – we used three matrices to perform $C = A \times B$. We then used the four scheduling algorithms discussed in section 3 to run the application: *Eager Scheduling, Round Robin Scheduling, Optimal Scheduling, and Pre-emptive Task Bunching Algorithm*. To recap:

1. *Optimal Scheduling:* A pre-computed schedule, which provides the shortest theoretical runtime, with the least number of preemption. All machines must be identical, all tasks must be identical. Works best when the number of tasks is a little larger than the number of machines.
2. *Eager Scheduling*: The only non-preemptive scheduler in our lineup. Each of the *m* machines get one of *n* tasks. When a task is finished, the machine gets an assignment of another tasks such that this task has not been assigned yet. If all tasks have been assigned, an unfinished task is reassigned to this machine.
3. *Round Robin Scheduling*: Out of *n* tasks, *m* of them are assigned to *m* machines. After a prescribed time, all *m* tasks are preempted and the next *m* tasks (in a circular fashion) are assigned.
4. *Task Bunching*: The *n* tasks are (approximately) evenly divided (bunched) among the *n* machines. Each machine runs its task bunch sequentially. When a task bunch completes, all task bunches are preempted, the tasks are divided into machines again and the execution restarted.

We ran a set of timing experiments in order to compare the performance of these four scheduling methods. The experiments were run on three Pentium-II 266 systems with 128 Mbytes of memory connected with a 100Mb/s Ethernet. The operating system used was Windows NT 4.0. The compiler used was Visual C++ 4.0. *All timings are elapsed times, measure by a real clock* (not system reported runtime).

To ensure fair comparison of the parallel program we first ran the program as a sequential program written in C++ (not CC++). This program was compiled and run to obtain a baseline performance measure. The sequential program ran in 540 sec's on a single processor. The performance results are shown in Figure 1.

Note that the preemptive protocols such as Round Robin and Task Bunching are expected to do well in dynamic situations, where machine speeds vary, and tasks lengths are different and unpredictable. Our tests use identical machines and equal task lengths. Hence, the bias is against preemptive schedulers. Yet, the results for preemptive scheduling are good.

For each of the scheduling protocols, a set of timings was obtained (if possible). The program was run as a very coarse grained parallel program, where the matrix multiply was performance by 5 tasks. A course grain execution used 10 tasks, a medium grained execution used 20 tasks and a very fine grained execution used 1500 tasks. The round robin time quantum was set at 15 seconds. The chart below summarizes the results obtained.

When the run involved five tasks computing on three machines, there is a severe mismatch of machines to tasks. Eager scheduling performs poorly. However, the optimal and round robin performs well. The task bunching was not run, as there is nothing to bunch! The optimal algorithm pre-computes schedules and is exceptionally suited for this situation - as it assumes equal task length and equal machine speed. In fact, we do not consider the optimal algorithm to be practically viable, and hence we do not test it any further.

For coarse grain tasks, we use 10 tasks to run the computation. This actually favors Round Robin and Task Bunching – Eager Scheduling performs a little poorer as it is non-preemptive and the number of tasks is not a multiple of the number of machines.

To level the playing field, we chose a medium grained task set, which uses 21 tasks. The number 21 is a multiple of 3. Here Eager Scheduling did as well as the others - but actually Round Robin fared a little poorer, as it has more preemption overhead than the others.

The fine grain tasks set is interesting - it uses 1500 tasks. Both Eager Scheduling and Round Robin became the same algorithm, as the time quantum never expired. However, the overhead of assigning the tasks to the workers dominated the compute time, giving poor performance. However, the total time was still less than the sequential time (540 sec's).

The results show the following clearly:

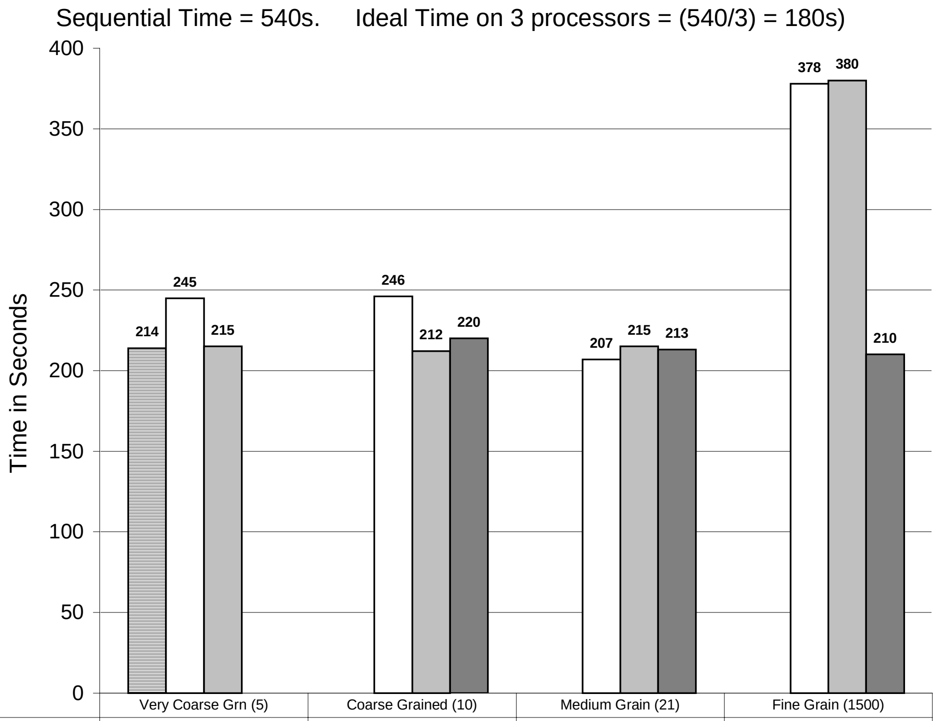
- Round Robin scheduling can be used for very coarse grained computations. Since our version of Round Robin is fault tolerant, this method will provide good performance even in spite of failures (the testing of that is not shown).
- For most other types of computations, Task Bunching is a good protocol. Its pre-emptive nature ensures good performance in a wide range of cases (except the very coarse grained case). In fine-grained cases, it maintains low overhead where other algorithms fail.
- Round Robin scheduling is not as silly as it sounds. In fact, it has quite respectable performance. It would be even better at providing fault tolerance than Eager Scheduling as it checkpoints a tasks at each switch point, providing low loss of computation in case of worker failure.
- Preemptive scheduling has definite advantages over non-preemptive scheduling, and the overhead is not overwhelming.

7. Conclusions

Scheduling in distributed systems is generally non-preemptive, however, we illustrate how preemptive scheduling can produce speedups of computations, in practice. For large grained computations, the overhead introduced by preemptive scheduling is more than offset by higher utilization of the machines and thus betters turnaround times of the computation. For fine-grained computations, task bunching used along with preemption provides good performance over a range of grain sizes.

In the simplest case, it is possible to benefit from preemptive scheduling by the use of the round robin scheduling. For round robin scheduling, the time slice is an important parameter and choosing it carefully is necessary for good performance. If some information is available about the execution times of the tasks, then the scheduling can be done using an optimal algorithm. Estimation of task length can also be used to obtain the schedules. Finally, task bunching is a scheduling algorithm that works on both coarse and fine grained task sizes.

The major challenge in the implementation is the proper migration of tasks. Task migration is somewhat simplified by using thread migration and not process migration. However, in a system that supports DSM, the interactions between the memory faulting handlers and the migration mechanisms can be tricky to resolve.



	Very Coarse Grn (5)	Coarse Grained (10)	Medium Grain (21)	Fine Grain (1500)
▨ Optimal	214			
□ Eager Scheduling	245	246	207	378
▨ Round Robin	215	212	215	380
■ Task Bunching		220	213	210

Figure 1: Performance of Preemptive Scheduling

The basic Chime system, which includes language support, shared memory, fault-tolerance and load balancing is available from the web at http://milan.eas.asu.edu. The software for preemptive scheduling in Chime will be made publicly available, shortly.

8. References

[ACD+96] C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29(2), pp. 18-28, February 1996.

[BDK95] A. Baratloo, P. Dasgupta, and Z. 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.*

[BZS93] B. Bershad, M. Zekauskas, and W. Sawdon. The Midway Distributed Shared Memory System. In *Proceedings of COMPCON '93*, February 1993, pp528-537.

[Cal96] The Calypso Home Page (under the Milan Project). http://milan.eas.asu.edu

[Car95] J. Carter. Design of the Munin Distributed Shared Memory System. *Journal of Parallel and Distributed Computing*, 29: pages 210, 227, 1995.

[CK92] K. M. Chandy and C. Kesselman, *CC++: A Declarative Concurrent, Object Oriented Programming Notation*, Technical Report, CS-92-01, California Institute of Technology, 1992.

[Das97] P. Dasgupta, Parallel processing with Windows NT Networks, USENIX Windows NT workshop, 1997.

[GBD+94] Al. Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Mancheck, and Vaidy Sunderam. PVM: Parallel Virtual Machine. The MIT Press, 1994.

[GLS94] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI Portable Parallel Programming with the Message Passing Interface*. MIT Press, 1994.

[GS76] T. Gonzalez and S. Sahni. Open Shop Scheduling to Minimize Finish Time. *Journal of the ACM*, 23(4), pp. 665-679, October 1976.

[Kha96] D. Khandekar. *Quarks: Distributed Shared Memory as a Basic Building Block for Complex Parallel and Distributed Systems.* Master's Thesis. University of Utah. March 1996.

[LLM88] M. Litzkow, M. Livny, and M. Mutka. Condor -- A hunter of Idle Workstations. In *Proceedings of the 8th IEEE International Conference on Distributed Computing Systems*, pages 104-111, 1988.

[Mc97] D. McLaughlin. *Scheduling Fault-Tolerant in a Distributed Environment*. Ph.D. Thesis, Arizona State University. December 1997

[ML92] E. Markatos and T. LeBlanc. Using Processor Affinity in Loop Scheduling on Shared-Memory Multiprocessors. In *Proceedings of the Conference on Supercomputing*, pp. 104-113, November 1992.

[MSD97] D. Mclaughlin, S. Sardesai, and P. Dasgupta. Calypso NT: Reliable, Efficient Parallel Processing on Windows NT Networks. Technical Report, TR-97-001, Department of Computer Science and Engineering, Arizona State University, 1997.

[Ou82] J. K. Ousterhout, *SchedulingTtechniques for Concurrent Systems* 3rd International Conference on Distributed Computing Systems, pp. 22-30, October 1982.)

[PK87] C. Polychronopoulos and D. Kuck. Guided Self-scheduling: A Practical Scheduling Scheme for Parallel Supercomputers. *IEEE Transactions on Computers*, C-36:1425-1439, December 1987.

[Rich95] J. Richter. *Advanced Windows: The Developers Guide to the Win32 API for Widows NT 3.5 and Windows 95*. Microsoft Press, Redmond, WA, 1995.

[Sar97] S. Sardesai. *Chime: A Versatile Distributed Processing System*. Ph.D. Thesis, Arizona State University. May 1997

[SGDM94] V. Sundaram, G. Geist, J. Dongarra, and R. Mancheck. The PVM Concurrent Computing System: Evaluation, Experiences, and Trends. *Parallel Computing*, **20**, 1994.

[SS96] A. Sen, A. Sengupta. On Task Assignment Problems in a Heterogeneous Computing Environment. 1996 Asilomar Conference on Circuits and Systems, Asilomar Conference Grounds, November, 1996.

[TG89] A. Tucker and A. Gupta. Process Control and Scheduling Issues for Multiprogrammed Shared Memory Multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 159 - 166, December 1989.

Sponsor Acknowledgment: Effort sponsored by the Defense Advanced Research Projects Agency and AFRL/Rome, Air Force Materiel Command, USAF, under agreement number F30602-96-1-0320. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation thereon.

Sponsor Disclaimer: The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, AFRL/Rome, or the U.S. Government.