Distributed Preemptive Scheduling on Windows NT

Donald McLaughlin and Partha Dasgupta
Arizona State University

Distributed Preemptive Scheduling on Windows NT[1]

Donald McLaughlin and Partha Dasgupta
Arizona State University
`partha@asu.edu`

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. Often the tasks are not of equal length, the machines are not of equal speeds and tasks can create further subtasks. These situations lead to non-optimal matches of workers to tasks causing executions that do not complete as quickly as it would be possible in a better matched case. Also, the granularities of the tasks may be small, leading to high overhead.

2. Distributed Scheduling

Our research has addressed such problems in a variety of ways. We have developed scheduling algorithms, both non-preemptive and preemptive that provide good throughputs in managing distributed computations, even when the granularities of tasks are small.

Our research environment consists of the *Chime* parallel processing systems running on the Windows NT operating system. This system support 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*. Eager Scheduling is similar to a FIFO scheduling algorithm augmented to provide fault tolerance (by assigning uncompleted tasks repeatedly).

Hence, without intelligent scheduling, the faster machines idle at barrier points waiting for the slower machines to finish, causing reductions in throughput. In addition, small mismatches in the number of machines and tasks cause large idle times and low granularities cause high overhead.

We have found that various preemptive scheduling algorithms can be used in such situations for significant performance improvements, in spite of the overhead of preemptive scheduling in distributed systems.

3. Preemptive Scheduling

Over the last few years we have simulated and implemented a host of preemptive scheduling algorithms. We now present two such algorithms.

The first 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.

The second 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.

4. Implementation and Performance

We have implemented the algorithms on the Chime parallel processing system running on Windows NT. The major roadblock turned out to be process migration under NT. The lack of signals posed the greatest problem as a thread can only be interrupted by another thread that suspends it. Care has to be taken to ensure that the thread to be migrated is not suspended waiting for a runtime event. Race conditions and starvation conditions have been encountered.

The final system runs well, and performance results are very encouraging. We found that the round-robin scheduler provided acceptable performance on large grained programs, but was hampered by the migration overhead. The task bunching scheduler performed really well in a wide variety of situations. More information and papers can be found at http://milan.eas.asu.edu.

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