Preemptive Task Scheduling for Distributed Systems*

Andrei Rădulescu and Arjan J.C. van Gemund

Delft University of Technology, The Netherlands

Abstract. Task scheduling in a preemptive runtime environment has potential advantages over the non-preemptive case such as better processor utilization and more flexibility when scheduling tasks. Furthermore, preemptive approaches may need less runtime support (e.g. no task ordering required). In contrast to the non-preemptive case, preemptive task scheduling in a distributed system has not received much attention. In this paper we present a low-cost algorithm, called the Preemptive Task Scheduling algorithm (PTS), which is intended for compile-time scheduling of coarse-grain problems in a preemptive distributed-memory system. We show that PTS combines the low-cost of the algorithms for the non-preemptive case with a simpler runtime support, while the output performance is still at a level comparable to the non-preemptive schedules.

1 Introduction

Compile-time scheduling for distributed-systems has recently received considerable attention in the context of non-preemptive environments in which tasks are scheduled to run uninterrupted until completion [1, 3, 6, 8, 9]. In particular it has been shown that there exist several scheduling algorithms (e.g., DSC [9], HLFET [1], FCP [6]) that pair good performance with low cost [6]. Efficient algorithms in the non-preemptive case are typically focused on scheduling first the most "important" tasks (i.e., tasks where delaying their execution will cause a longer completion time).

A preemptive scheme may be more attractive, as a less important task can run while waiting an important task to become ready. We can distinguish two approaches: (a) preemptive priority discipline, where the less important task is preempted when the more important becomes ready, (b) processor sharing discipline, where the ready tasks run concurrently, interleaved in small time slices. We chose the latter because (1) the scheduling process is simpler (i.e., faster), (2) the runtime system support is simpler as no task ordering is required, and (3) frequent context switching overhead is low if one of the available light-weight thread packages is used (e.g., PThreads [5]).

For shared-memory systems it has been proven that an optimal preemptive task schedule is indeed shorter than non-preemptive schedules [2]. In the distributed case however, there is no such proof, nor we are aware of compile-time task scheduling algorithms specifically designed for distributed-memory preemptive environments.

In this paper we show that the existing scheduling algorithms for the non-preemptive case suffer considerable loss of performance when executed in a preemptive runtime

* This research is part of the Automap project granted by the Netherlands Computer Science Foundation (SION) with financial support from the Netherlands Organization for Scientific Research (NWO) under grant number SION-2519/612-33-005.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 272–276, 2000.

environment based on a processor sharing discipline. A new scheduling algorithm for preemptive environments called the Preemptive Tasks Scheduling algorithm (PTS) is presented. PTS is focussed towards coarse-grain applications, as it aims to maintain an even processors load throughout the time rather than to reduce communication costs. Although PTS does not quite exploit the potential advantages of a preemptive scheme, it combines the low-cost of the algorithms for the non-preemptive case with a simpler runtime support, while the output performance is still at a level comparable to the non-preemptive schedules.

This paper is organized as follows: The next section describes the scheduling problem and introduces some definitions used in the paper. Section 3 presents the PTS algorithm, while Section 4 describes its performance. Section 5 concludes the paper.

2 Preliminaries

A parallel program can be modeled by a directed acyclic graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is a set of V nodes and $\mathcal{E}$ is a set of E edges. Nodes depict tasks, while edges represent inter-task communication. If two tasks are scheduled on the same processor, the communication cost between them is assumed to be zero. The length of a path is the sum of the computation and communication costs of the tasks and edges belonging to the path, respectively. The task's *bottom level* is the length of the longest path from the current task to any exit task. A task is said to be *ready* if all its parents have finished their execution. A task can start its execution only after all its messages have been received. The objective of the scheduling problem is to find a schedule of the tasks in $\mathcal{V}$ on the processors in $\mathcal{P}$ such that the parallel completion time (schedule length) is minimized.

3 The PTS Algorithm

Essentially, PTS schedules tasks in the order of their bottom levels on the least loaded processor (i.e., processor with the lowest number of tasks running on it). However, selecting the least loaded processor at a low cost is not a trivial problem. The processor load is given by the number of tasks simultaneously running on that processor. The time a task is running on its assigned processor is given not only by its size, but also by the processor load. Consequently, we need to simulate the preemptive execution of the tasks in order to compute the current processor load when scheduling a new task.

The PTS algorithm is formalized in Figure 1. Details about our simulation scheme and other implementation issues can be found in [7].

First, the tasks' priorities (bottom levels) are computed. Then, PST starts scheduling one ready task at a time. At each iteration, the ready task with the highest bottom level is selected. Using tasks' bottom levels ensures that the tasks are scheduled in the correct order with respect to dependencies.

Before scheduling the current task t, the task execution simulation is updated by stopping the tasks that finish before t's last message arrival time. As a consequence, the processor loads are also updated. Then, t is scheduled on the least loaded processor.

The PTS's complexity is $O(V(\log(V) + \log(P)) + E)$ as explained in [7].

```
PTS ()
BEGIN
    For all tasks compute bottom levels.
    WHILE NOT all tasks scheduled DO
        t ← Task with the highest bottom level.
        MAT ← t's last message arrival time.
        Stop the tasks finishing before MAT
            and update their successors.
        ST ← t's start time.
        p ← Least loaded processor.
        Start task t on p at ST.
    END WHILE
END
```

Fig. 1. The PTS algorithm

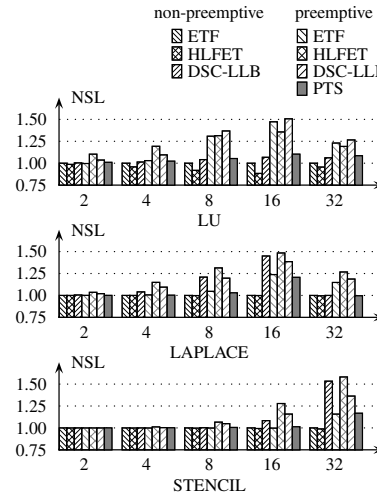

Fig. 2. Performance comparison

4 Performance Results

In this section we first investigate the loss of performance of three well-known non-preemptive scheduling algorithms (ETF [3], HLFET [1] and DSC-LLB [9, 8]) when simply applied to a preemptive environment. Next, we show the PST performance compared to the three above-mentioned algorithms in terms of both schedule lengths and execution times, but now run on a non-preemptive environment, for which they were originally designed.

We consider task graphs representing various types of parallel algorithms. The selected problems are *LU decomposition* ("LU"), *Laplace equation solver* ("Laplace") and a *stencil algorithm* ("Stencil"). For each of these problems, we adjusted the problem size to obtain task graphs of about 2000 nodes. As we consider the coarse-grain case, we generate task graphs with a communication to computation ratio (CCR) of 5. For each problem we generate 5 graphs with random execution times and communication delays (i.i.d. uniform distribution with unit coefficient of variance).

For performance comparison, we use the *normalized schedule length* (NSL), which is defined as the ratio between the schedule length of the given algorithm and the schedule length of a reference algorithm. As a reference algorithm we use ETF, applied to a non-preemptive environment. Schedule lengths are obtained by simulating the problems' execution in a homogeneous distributed system.

Scheduling Performance: As to be expected, in Figure 2 it can be seen that ETF, HLFET and DSC-LLB suffer a performance degradation when applied in a preemptive runtime environment. One can note that all ETF, HLFET and DSC-LLB algorithms yield schedules up to 47%, 53% and 41% longer in the preemptive case compared to the non-preemptive case. In contrast, PTS consistently yields better performance in the preemptive case. The figure also shows that PTS has schedule lengths comparable with

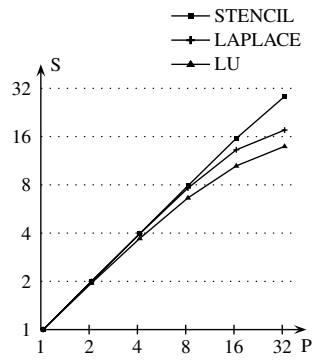

Fig. 3. PTS speedup

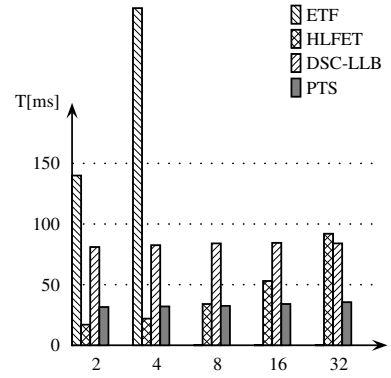

Fig. 4. Scheduling algorithm cost

those produced by ETF and HLFET in the non-preemptive case (for which they have been designed), and even improves DSC-LLB's schedule lengths with up to 31%.

Speedup: In Figure 3 we show the PTS speedup for the considered problems. For all the problems, PTS obtains significant speedup. For LU and Laplace, there are a large number of join operations. As a consequence, there is not much parallelism available and the speedup is lower. Stencil is more regular. Therefore more parallelism can be exploited and better speedup is obtained.

Running Times: In Fig. 4 the average running time of the considered algorithms is shown in CPU seconds as measured on a Pentium Pro/233MHz PC with 64Mb RAM. One can note that PTS's overall running time is the lowest, varying around 31 ms.

5 Conclusion

In this paper we investigate the potential advantages of compile-time task scheduling in a preemptive runtime system. A new scheduling algorithm, called Preemptive Task Scheduling algorithm (PTS) is presented, that is specifically designed for compile-time scheduling of coarse-grain problems in a preemptive runtime environment. PTS primarily focuses on obtaining a better processor utilization at a very low-complexity ($O(V(\log(V) + \log(P)) + E)$).

Experiments show that PTS performs comparable to ETF and HLFET, two top scheduling algorithms in the non-preemptive case, and outperforms DSC-LLB. Although, our results indicate that the potential advantages of using a preemptive scheme have not yet been exploited, PTS requires a simpler preemptive runtime management, as no task ordering is required. Further research will be performed to improve the scheduling algorithms in the preemptive case in order to outperform non-preemptive scheduling algorithms.

References

[1] T. L. Adam, K. M. Chandy, and J. R. Dickson. A comparison of list schedules for parallel processing systems. *Communications of the ACM*, 17(12):685–690, 1974.

[2] E. G. Coffman Jr. *Operating Systems Theory*. Prentice Hall, 1973.

[3] J-J. Hwang, Y-C. Chow, F. D. Anger, and C-Y. Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM J. on Computing*, 18:244–257, 1989.

[4] Y-K. Kwok and I. Ahmad. Benchmarking the task graph scheduling algorithms. In *Proc. IPPS/SPDP*, 1998.

[5] F. Mueller. A library implementation of POSIX threads under UNIX. In *Proc. Winter*, 1993.

[6] A. Rădulescu and A. J. C. van Gemund. On the complexity of list scheduling algorithms for distributed-memory systems. In *Proc. ACM ICS*, 1999.

[7] A. Rădulescu and A. J. C. van Gemund. Preemptive task scheduling for distributed systems. TR 1-68340-44(2000)04, Delft Univ. of Technology, 2000.

[8] A. Rădulescu, A. J. C. van Gemund, and H-X. Lin. LLB: A fast and effective scheduling algorithm for distributed-memory systems. In *Proc. IPPS/SPDP*, 1999.

[9] T. Yang and A. Gerasoulis. DSC: Scheduling parallel tasks on an unbounded number of processors. *IEEE Trans. on Parallel and Distributed Systems*, 5(9):951–967, 1994.