Post-Scheduling Optimization of Parallel Programs

Stephen Shafer[1] and Kanad Ghose[2]

[1] Lockheed Martin Postal Systems, 1801 State Route 17 C,
Owego, NY 13827-3998
shafer@cs.binghamton.edu

[2] Department of Computer Science, State University of New York,
Binghamton, NY 13902-6000
ghose@cs.binghamton.edu

Abstract. No current task schedulers for distributed-memory MIMD machines produce optimal schedules in general, so it is possible for optimizations to be performed after scheduling. Message merging, communication reordering, and task duplication are shown to be effective post-scheduling optimizations. The percentage decrease in execution time is dependent on the original schedule, so improvements are not always achievable for every schedule. However, significant decreases in execution time (up to 50%) are possible, which makes the investment in extra processing time worthwhile.

1. Introduction

When preparing a program for execution on a distributed-memory MIMD machine, choosing the right task scheduler can make a big difference in the final execution time of the program. Many static schedulers are available which use varying methods such as list scheduling, clustering, and graph theory for deriving a schedule. Except for a few restricted cases, however, no scheduler can consistently produce an optimal answer. Thus, post-scheduling optimizations are still possible. Assuming that tasks are indivisible, and that the ordering of tasks on each processor does not change, there are two possible sources for improvements in execution time: the tasks themselves, and the inter-processor messages.

There are two different communication-related optimizations that we explore here. First, there is the possibility that pairs of messages from one processor to another may be combined, or merged, so that the data from both are sent as one message. This can result in a reduction of the execution time of the program. The second communication-related optimization deals with those tasks that have multiple messages being sent to tasks on other processors. For those tasks, the order of the messages may be changed so that more time-critical messages are sent first. We show two different orderings and how they affect the final execution time. Finally, we have implemented a task-related optimization that duplicates tasks when the messages that they send are causing delays in the destination tasks.

P. Amestoy et al. (Eds.): Euro-Par'99, LNCS 1685, pp. 440-444, 1999.

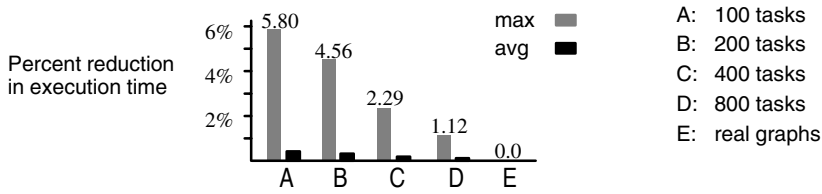
2. Message Merging

2.1 The Concept of Merging

Wang and Mehrotra [WM 91] proposed a technique for merging messages that correspond to data dependencies. Their approach reduces the number of communications by combining a pair of messages from a source processor to the same destination processor into a single message that satisfies the data dependencies. In [SG 95], we showed that this method can actually increase the execution time, and might even introduce deadlock. We also showed that message merging can reduce execution time without introducing deadlock if the right pairs of messages are merged. In our previous work, however, I/O channel contention was not considered when execution times were calculated, so our results could have been more accurate. The results in this paper include the effects of channel contention. The entire merging process is explained in [SG 95].

2.2 Merging Results

The merging process was performed on sets of randomly created task graphs with varying numbers of tasks, from 100 tasks up to 800 tasks. In addition, several task graphs from actual applications were tested. The results are shown in Figure 1.



igure 1. Message merging results on task graphs of varying sizes.

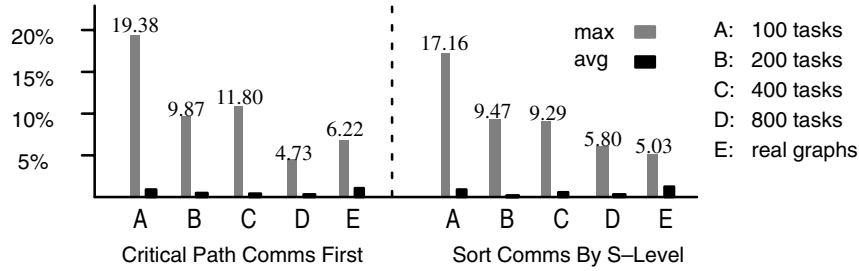
The actual applications fared the worst, with no merges being performed for any of the graphs tested. However, merging does not always guarantee a positive result. Given the small number of real graphs tested (five), it is possible that they just did not have the right circumstances for a profitable merge. The random graphs did much better. The number of graphs tested for each number of tasks was around fifty. That is, there were about 50 graphs with 100 tasks, 50 with 200 tasks, and so on. The results are shown as percent reduction in execution time, with maximums and averages shown. The results of merging depend on the original schedule. We expect that task graphs partitioned and scheduled by hand will show much better results than these.

3. Reordering Outgoing Communications

When multiple messages are being sent by one task, it is possible that more than one needs to be sent on the same link. When that occurs, and one of those messages is on the critical path, it may be delayed waiting for another message before it can get sent.

We have implemented two reordering schemes that determine whether the execution time of a schedule can be reduced by changing the order of outgoing communications.

The first reordering consists of sending any messages on the critical path first. This should avoid any extra communication delay caused by the critical path message waiting for the link unnecessarily. The second reordering consists of sorting all outgoing messages from all tasks by their scheduling level, or s-level, which is defined as the length of the longest path from a node to any terminal node. This optimization attempts to send messages first to those tasks that are highest up in the task graph in the belief that the schedule length will be reduced if the higher tasks are executed first.



igure 2. Reorder communications sent by one task.

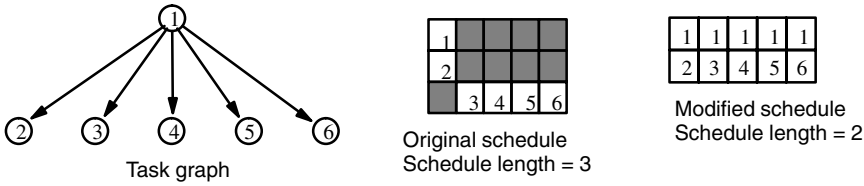
The results (see Figure 2) show that a delay due to the order of outgoing communications can occur in schedules, and can be avoided. Again, the numbers shown are percent decreases in the execution times of the schedule. Like the results of the merging optimization, the average decreases are small, caused by not being able to improve all schedules. The results here, however, are better than those for merging. The best decrease was a 19.38% reduction in execution time. Like merging, however, the results depend to a great degree on the schedule itself. The average reduction in execution time is rather small, but the chance of a greater reduction is still there.

4. Task Duplication

The study of task duplication has been motivated by the fact that the execution time of a parallel program can be greatly affected by communication delays in the system. Sometimes, these delays cause a task on the critical path to wait for a message from another task. If such a delay could be reduced, or removed, then the execution time of the program can be reduced. Task duplication tries to decrease these delays.

The simplest situation in which to duplicate tasks is when there is only one start node of the task graph, and it has multiple children. Figure 3 shows an example of this where it is assumes there are five processors, and each task and communication take unit time. In this situation, task 1 can be duplicated on every processor, allowing tasks 2 through 6 to start execution without waiting for a message from another processor. The final schedule length has been reduced by the duplication. A more complex example would be where a task with multiple incoming messages and multiple outgoing messages is causing one of its child tasks to be delayed. In this case, when the task is duplicated, all of its incoming messages must also be sent to the copies,

and each copy must only send a subset of the outgoing messages, with none of the messages being sent twice.



igure 3. Simple case of task duplication.

4.1 Duplication During Scheduling

There are several schedulers including [CC 91], [CR 92], [SCM 95], and [AK 98] that have used task duplication in the scheduling process. [AK 98] includes a comparison of these and other schedulers. All of them use task duplication during scheduling, which typically searches for an idle time slot on the destination processor in which to put the duplicated task. This ensures that nothing besides the delayed task's start time will be affected by the duplication. We feel that this is too restrictive. By allowing tasks only to be placed in idle processor time, opportunities for duplication are passed up that may have adversely affected some task's start time, but if that task is not on the critical path then it might not matter. The possible benefit of duplication is lost.

If task duplication were to allow a duplicated task to adversely affect some task's start time, however, there is no way to know during scheduling whether or not the duplication has increased the final schedule length. We feel that the answer is post-scheduling task duplication.

4.2 Duplication After Scheduling

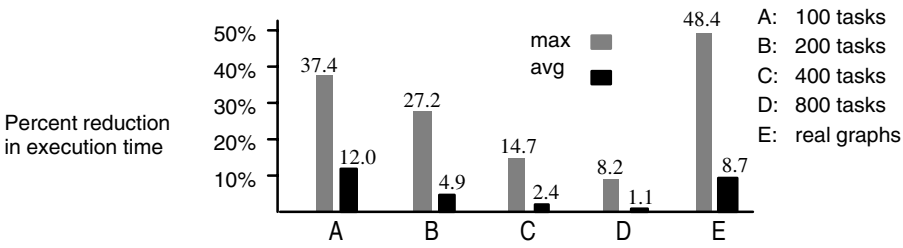
Duplicating tasks can reduce the execution time of a program, but it can also increase it. Copies of tasks other than start tasks require duplication of incoming messages also. Before the final schedule is produced, the effects of these extra messages on the schedule length can't be known. If duplication is performed after scheduling, however, all affects of any change can be known before that change is accepted. In addition, unnecessary duplication can be avoided since only the critical path needs to be checked for possible duplications. The major steps that need to be performed are:

- copy the task to the new PE
- duplicate messages sent to the task so that both copies of the task receive them
- have each copy of the task send a disjoint subset of outgoing messages, depending on the destination

The results of testing this approach are shown in Figure 4. There are significant decreases in execution time seen for all categories of task graph tested. One of the actual application graphs, a task graph for an atmospheric science application even had an almost 50% reduction in execution time.

5. Summary

We have shown three different types of post-scheduling optimizations: message merging, communication reordering, and task duplication. None of them promises to



igure 4. Results from Post Scheduling Task Duplication

be able to reduce the execution time of every schedule. In almost all cases tested, the average reduction seen was rather low, around 1 to 2 percent. However, the chances of achieving some reduction ranges from a low of 39.4% (comm reordering) to a high of 82.5% (task duplication). This makes it very likely that the execution time can be reduced to some extent. Given the maximum reductions seen for these optimizations, the possibility of significantly reducing the execution time is worth the extra processing time to try. We believe that these optimizations are best tried after scheduling has already been completed. The actual critical path is known at that point and there is no chance of making some change to the system without knowing what effect it will have on the final execution time.

References

[AK 98] I. Ahmad, Y.K. Kwok, "On Exploiting Task Duplication in Parallel Program Scheduling", *IEEE Trans. Parallel and Distrib. Systems,* vol. 9, no. 9, September 1998.

[CR 92] Chung, Y.C., Ranka, S., "Application and Performance Analysis of a Compile-Time Optimization Approach for List Scheduling Algorithms on Distributed-Memory Multiprocessors", *Proc. Supercomputing '92*, pp.512-521, November 1992.

[CC 91] Colin, J.Y., Chretienne, P., "C.P.M. Scheduling With Small Computation Delays and Task Duplication", *Operations Research*, pp. 680-684, 1991.

[SG 95] S. Shafer, K. Ghose, "Static Message Combining in Task Graph schedules", *Proc. Int'l. Conf. on Parallel Processing*, August 1995, Vol.-II.

[SCM 95] Shirazi, B., Chen, H., Marquis, J., "Comparative Study of Task Duplication Static Scheduling versus Clustering and Non-Clustering Techniques", *Concurrence: Practice and Experience*, vol. 7, no. 5, pp. 371-390, August 1995.

[WM 91] Wang, K.Y., Mehrotra, P., "Optimizing Data Synchronizations On Distributed Memory Architectures, *Proc. 1991 Int'l. Conf. on Parallel Processing*, Vol. II, pp. 76-82.