Coscheduling Based on Runtime Identification of Activity Working Sets

Dror G. Feitelson and Larry Rudolph

Received October 10, 1993; revised April 28, 1994

This paper introduces a method for runtime identification of sets of interacting activities ("working sets") with the purpose of coscheduling them, i.e., scheduling them so that all the activities in the set execute simultaneously on distinct processors. The identification is done by monitoring access rates to shared communication objects: activities that access the same objects at a high rate thereby interact frequently, and therefore would benefit from coscheduling. Simulation results show that coscheduling with our runtime identification scheme can give better performance than uncoordinated scheduling based on a single global activity queue. The finer-grained the interactions among the activities in a working set, the better the performance differential. Moreover, coscheduling based on automatic runtime identification achieves about the same performance as coscheduling based on manual identification of working sets by the programmer.

KEY WORDS: Coscheduling; gang scheduling; on-line algorithms; activity working set.

1. INTRODUCTION

The performance of multiprogrammed multiprocessors can be improved without additional burden on the programmer if the sets of closely interacting, fine-grained activities can be identified and scheduled to always execute simultaneously. Some twelve years ago John Ousterhout noticed an
analogy between memory management in multiprogrammed uniprocessor operating systems and processor management in multiprogrammed multiprocessor systems.\(^{(1)}\) The analogy was based on the observation that parallel applications require a “working set” of processes to execute simultaneously, just like uniprocessor applications require a working set of pages to be memory resident simultaneously. If a parallel application does not receive enough processors, it might thrash due to synchronization constraints among its processes. Executing processes become blocked waiting for responses from those that are currently not executing. The machine could spend most of its time context switching if an additional process is scheduled to execute when its interacting counterpart is descheduled from execution.

We use the term *activity working set* to refer to a set of activities that should be scheduled together. We use the term *activity* rather than *process*, as in Ousterhout’s “process working set,” to stress the fact that it is one of many components in a parallel processing application, and to avoid overloaded terms such as *task* or *thread*. The context of our discussion is parallel systems where programming is based on control parallelism, and activities can be created dynamically during execution.

The term *coscheduling* is used to describe scheduling algorithms that schedule on the basis of activity working sets, i.e., all the threads of an activity working set are scheduled to execute simultaneously. We shall be comparing our work to *uncoordinated scheduling* in which a single global activity queue (or workpile) is maintained, and each processor removes the next activity from the workpile, executes it for a whole time quantum, and returns it to the end of the workpile—the scheduling of one activity is uncoordinated with the scheduling of any other activity.\(^{(2,3)}\)

The subject of this paper is how to identify the activities that constitute a working set. The analogy with uniprocessor memory management is not so helpful here. The memory working set is approximated by using the least recently used (LRU) paradigm.\(^{(4)}\) Basically, this assumes that the past is indicative of the future and hence it may be expected that the next pages that the application will need are exactly those that it used most recently. These pages are therefore maintained in the primary memory, and only the least recently used pages are evicted when additional space is needed. This approach does not easily transfer to our case, because activities are active entities; there is no such thing as an “unused” activity that can be “evicted,” and because in general there can be a number of (disjoint) activity working sets within an application that each need to be scheduled simultaneously.

Activity working sets should be defined by the pattern of interactions among the activities. But it has usually been assumed that the working sets