The Performance of Work Stealing in Multiprogrammed Environments

Robert D. Blumofe Dionisios Papadopoulos
Department of Computer Sciences, The University of Texas at Austin
{rdb,dionisis}@cs.utexas.edu

May 28, 1998

Abstract

We study the performance of user-level thread schedulers in multiprogrammed environments. Our goal is a user-level thread scheduler that delivers efficient performance under multiprogramming without any need for kernel-level resource management, such as coscheduling or process control. We show that a non-blocking implementation of the work-stealing algorithm achieves this goal. With this implementation, the execution time of a computation running with arbitrarily many processes on arbitrarily many processors can be modeled as a simple function of work and critical-path length. This model holds even when the processes run on a set of processors that arbitrarily grows and shrinks over time. We observe linear speedup whenever the number of processes is small relative to the average parallelism.

1 Introduction

As small-scale multiprocessors make their way onto desktops, the high-performance parallel applications that run on these machines will have to live alongside other applications, such as editors and web browsers. Similarly, users expect multiprocessor compute servers to support multiprogrammed work loads that include parallel applications. Unfortunately, unless parallel applications are coscheduled [40] or subject to process control [44], they display poor performance in such multiprogrammed environments [10, 17, 18, 19, 26].

As an alternative to coscheduling or process control, in this paper we investigate the use of dynamic, user-level, thread scheduling in order to achieve efficient performance under multiprogramming. We show that a non-blocking implementation of the well-known and provably efficient "work-stealing" scheduling algorithm [15] delivers efficient performance under multiprogramming. Moreover, we develop and evaluate a simple performance model based on "work" and "critical-path length" that characterizes accurately the performance of parallel applications that use this non-blocking work stealer. In fact, this performance model is based on an analytical bound that we have proven to hold in a model where the kernel-level scheduling is actually performed by an adversary [9]. Thus, our model is extraordinarily robust.

We shall restrict attention to shared-memory multiprocessors, and all experiments are performed on a Sun Ultra Enterprise 5000 with 8 167-Mhz UltraSPARC processors running Solaris 2.5.1. We shall use the word "process" to denote a kernel-scheduled entity, and we shall assume that all processes belonging to the same executing program can share memory and synchronize through the use of synchronization variables. Such processes are often referred to as "light-weight processes" or "kernel threads." We shall reserve the

This research was supported in part by the Defense Advanced Research Projects Agency (DARPA) under Grant F30602-97-1-0150 from the U.S. Air Force Research Laboratory. Multiprocessor computing facilities were provided through a generous donation by Sun Microsystems, Inc.

word "thread" to denote a user-level task that is scheduled by a user-level library. The user-level library schedules threads onto processes, and the kernel schedules processes onto processors.

Our goal is to develop a scheduler for a user-level threads library that performs well under multiprogramming, regardless of the behavior of the kernel scheduler. Specifically, our scheduler should utilize efficiently whatever set of processors the kernel scheduler happens to give it, even if the kernel scheduler gives it fewer processors than it has processes and even if that set of processors grows and shrinks over time. Such a scheduler could be employed by a parallelizing compiler, or the runtime system for a multithreaded language such as Cilk [14] or Java [8].

1.1 The problem with static partitioning

Before considering dynamic thread scheduling, we first review a well-known performance anomaly that occurs when parallel programs use a static partitioning of the work [31, pages 284–285]. In the simplest case when such a program executes, it creates some number P of processes, where typically P is selected by a command-line argument, and each process performs a $1/P$ fraction of the total work. Let T_1 denote the ***work*** of the computation, which we define as the execution time with $P = 1$ process. Using $P \geq 1$ processes, each process performs T_1/P work, and if the overhead of creating and synchronizing these processes is small compared to the T_1/P work per process, then we can hope that the execution time T_P will be given by $T_P = T_1/P$, thereby giving a ***speedup*** of $T_1/T_P = P$. Of course, this aspiration assumes that we have at least P processors on which to execute the program.

In a multiprogrammed environment, we might find that the actual number P_A of processors on which our program runs is smaller than the number P of processes, and in this case we cannot hope to achieve a speedup of P. Note that we always have $P_A \leq P$, because a program cannot run on more processors than it has processes. Thus, in a multiprogrammed environment, we can aspire more reasonable to achieve an execution time of $T_P = T_1/P_A$, thereby giving a speedup of $T_1/T_P = P_A$ — that is, ***linear speedup*** — and a (processor) ***utilization*** of $T_1/(P_A T_P) = 1.0$. Unfortunately, for some problem inputs, our statically partitioned applications do not come close to fulfilling this aspiration unless we have $P_A = P$, effectively a non-multiprogrammed, dedicated machine.

Figure 1(a) shows the measured speedup of several statically partitioned applications for different numbers P of processes. More information about these applications is given in Table 1, and various characteristics for each of these applications, including the value of T_1, are given in Table 2. The applications are run on a dedicated machine with $P_M = 8$ processors, so the actual number P_A of processors used is given by $P_A = \min\{P_M, P\} = \min\{8, P\}$. Observe that when we have $P \leq 8$, we have $P_A = P$, and all four applications come reasonably close to the ideal linear speedup. On the other hand, when we have $P > 8$, we have $P_A < P$, and performance drops off dramatically. In fact, the worst case is when we are off by only 1 — that is, when $P = 9 = P_A + 1$. In this case, the P_A processors begin by executing $P - 1$ of the processes, all of which complete in time T_1/P. Then, one of the processors executes the one remaining process, which also completes in time T_1/P. Thus, we have an execution time of $T_P = 2(T_1/P) = 2T_1/(P_A + 1)$, thereby giving a speedup of $T_1/T_P = (P_A + 1)/2 \approx P_A/2$ and an utilization of $T_1/(P_A T_P) = (P_A + 1)/(2P_A) \approx 0.5$ — roughly half the desired speedup and utilization.

The traditionally proposed solution to this problem is to use a number P of processes that is significantly greater than the number P_M of machine processors, so that we are guaranteed to have $P \gg P_A$ [31, page 285]. Indeed, using extra processes can improve the load imbalance, but as we see in Figure 1(a), it does not solve the problem. As P grows, the overhead of creating and synchronizing the processes grows and the work per process T_1/P shrinks. For sufficiently large values of T_1, this problem will not occur, because the time slicing divides each process into smaller pieces and fixes the load imbalance. Ultimately, however, this observation cannot console us. We want our applications to perform well for all input problems.