Hood: A User-Level Threads Library for Multiprogrammed Multiprocessors

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

October 28, 1998

Abstract

The Hood user-level threads library delivers efficient performance under multiprogramming without any need for kernel-level resource management, such as coscheduling or process control. It does so by scheduling threads with a non-blocking implementation of the work-stealing algorithm. With this implementation, the execution time of a program 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 program runs on a set of processors that arbitrarily grows and shrinks over time. In all cases, we observe linear speedup whenever the number of processes is small relative to the 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 [25] or subject to process control [27], they display poor performance in such multiprogrammed environments [5, 12, 13, 14, 16]. As an alternative to coscheduling or process control, in this paper we present Hood, a C++ user-level threads library, implemented entirely at user level with no operating-system modifications, whose scheduler achieves efficient performance under multiprogramming.

Hood runs on shared-memory multiprocessors only, and all experiments reported in this paper were 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 thread (or light-weight process) that is managed and scheduled by the kernel. All processes belonging to the same executing program can share memory and synchronize through the use of synchronization variables. We reserve the word "thread" to denote a user-level task that is managed and scheduled by a user-level library. In this two-level scheduling model, a user-level library, such as Hood, schedules a program's threads onto its processes, and the kernel schedules all processes onto all processors.

We show in this paper that Hood applications utilize efficiently whatever processor resources the kernel scheduler happens to allocate, regardless of the behavior of the kernel scheduler. Specifically, we show that this efficiency holds even if the kernel scheduler gives the application fewer processors than it has

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.

This paper has been submitted for publication.

processes and even if that set of processors grows and shrinks arbitrarily over time. 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 `Hood`. 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 [4].

1.1 The problem with static load balancing

Before considering `Hood`, we first review a well-known performance anomaly that occurs when parallel programs use static load balancing [20, 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 1 process running on 1 dedicated processor. Using P 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 will be given by $T = T_1/P$, thereby giving a ***speedup*** of $T_1/T = P$. 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 reasonably to achieve an execution time of $T = T_1/P_A$ and a speedup of P_A. In this case, the (processor) ***utilization*** is given by $T_1/(P_A T) = 1.0$, so processor resources are being fully utilized. This linear speedup and full utilization is our goal. Unfortunately, for some problem inputs, our statically load-balanced applications do not come close to fulfilling this goal unless we have $P_A = P$, effectively a non-multiprogrammed, dedicated machine.

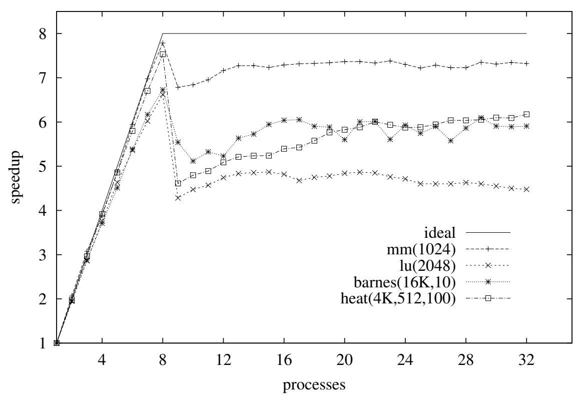
Figure 1(a) shows the measured speedup of several statically load-balanced applications for different numbers P of processes. More information about these applications is given in Table 1, and various characteristics for each of these applications, including the value of T_1, are given in Table 2. The applications are run on a dedicated machine with 8 processors, so the actual number P_A of processors used is given by $P_A = \min\{8, P\}$. 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 $P_A = 8 < P$, and performance drops off dramatically. In this regime, our experiment is an idealization of the situation that might occur if the machine has P processors and the program creates P processes, but while the program is running, some other running programs take away processor resources so that only 8 processors are actually available.

We observe in Figure 1(a) a performance cliff, because 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 $T = 2(T_1/P) = 2T_1/(P_A + 1)$, thereby giving a speedup of $T_1/T = (P_A+1)/2 \approx P_A/2$ and an utilization of $T_1/(P_A T) = (P_A+1)/(2P_A) \approx 0.5$ — roughly half the desired speedup and utilization. Our statically load-balanced applications fall off this performance cliff when the number of processes exceeds the number of available processors.

1.2 Summary of results

As an alternative to static load balancing, `Hood`’s thread scheduler dynamically assigns the application’s work to its processes, and the result is performance as shown in Figure 1(b). Here we have performed the same experiment as in Figure 1(a) with the same applications and a couple more, but now the applications

(a) Static load balancing.



(b) Hood.

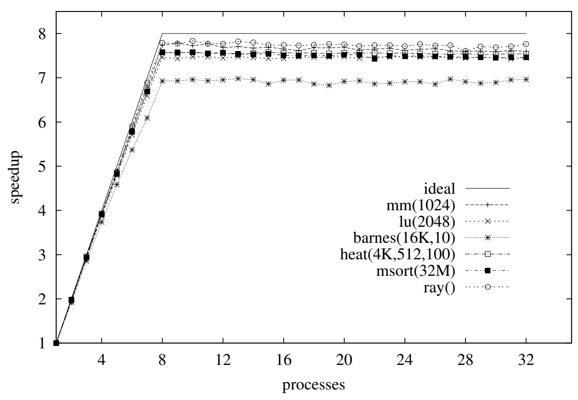


Figure 1: Measured speedup plotted as a function of the number P of processes used when run on a dedicated 8-processor machine. **(a)** Measured speedup for statically load-balanced applications. **(b)** Measured speedup for Hood applications.

are recoded to use Hood. We observe that our Hood applications come close to linear speedup across a wide range of numbers P of processes, including the cases when we have $P_A < P$. Moreover, as we document in Section 4, we have not sacrificed any performance in the cases when we have $P_A = P$. Hood eliminates the performance cliff.

In using Hood, an application partitions its work into threads, where the amount of work in each thread and the number of threads is completely independent of the number of processors or processes. Instead, the partitioning into threads is determined by the amount of parallelism in the parallel algorithm being used. For example, in a divide-and-conquer algorithm in which the recursive subproblems can be solved in parallel, a separate thread is created for each recursive call. Thus, Hood applications may create millions of threads. Because the threads are created and synchronized at user-level, only a small amount of work per thread is need to amortize the cost of creating and synchronizing the myriad threads.

To eliminate the performance cliff, Hood schedules threads using a non-blocking implementation of the work-stealing algorithm [4, 9]. This implementation employs non-blocking synchronization [18] for the concurrent data structures and judicious use of “yield” system calls. Effectively, Hood’s non-blocking work stealer automatically adapts to the kernel’s allocation of processes to processors. If we have $P_A < P$, then some processes get less processor time than others (or maybe no processor time at all), but the non-blocking work stealer assigns such processes fewer (or no) threads to execute.

Though we see in Figure 1(b) flat performance curves, eventually for large enough P, these curves must trend downwards, and we provide a model of Hood application behavior that characterizes where this downward trend begins. By studying the performance of several Hood applications with many input problems, we show that the execution time T of Hood applications can be bounded by the formula

$$T \leq c_1 T_1 / P_A + c_\infty T_\infty P / P_A \; ,$$

where c_1 and c_∞ are small constants, and T_∞ is the ***critical-path length*** of the computation, which is a lower bound on the execution time for any number of processes and processors. This bound holds even when the program runs on a set of processors that grows and shrinks over time, in which case we define P_A as the time-average actual number of processors on which the program runs. Importantly, we find that this bound holds with the constant c_1 close to 1. Thus, we obtain linear speedup — that is, $T \approx T_1/P_A$ — whenever the work term T_1/P_A dominates the critical-path term $T_\infty P/P_A$. This domination occurs whenever P is small relative to T_1/T_∞, a ratio that is naturally interpreted as the ***parallelism*** of the computation. Thus, Hood applications realize linear speedup whenever the number of processes is small relative the computation's parallelism. That is, the curves of Figure 1(b) will start downwards when P becomes comparable to the parallelism.

The remainder of this paper is organized as follows. In Section 2 we give a brief overview of the Hood library interface and implementation. Hood's non-blocking work stealer is described in Section 3, and some performance characteristics of Hood applications are discussed in Section 4. We present the results of a performance-modeling study in Section 5, the upshot of which is a characterization of the relationship between performance, processes, and parallelism. In Section 6 we discuss some related work, point out some of Hood's limitations, and mention some plans for future work. We conclude in Section 7.

2 The Hood library

In this section, we give a brief overview of the Hood library: its interface and implementation. Hood is implemented entirely at user level with no modification to the operating system. The following description is based on version 2.1 of the library, which runs on SPARC v9-based machines with the Solaris operating system. Hood threads can synchronize directly through signals. That is, a thread can block waiting for a specified number of signals, and other threads can send signals directly to the waiting thread. Alternatively, threads can synchronize indirectly through synchronization variables. The only synchronization variable supported by Hood 2.1 is a semaphore. Mutexes and condition variables will be supported in a later release.

The Hood library contains definitions for four C++ classes that programs use to create and synchronize threads: `HoodThread`, `HoodSemaphore`, `HoodSerMach`, and `HoodParMach`. The `HoodThread` class is an abstract class with one method: a virtual method called `run`. Programs define threads by subclassing `HoodThread` and supplying a definition for the `run` method with appropriate instance variables. A thread is created at runtime when an object of any type derived from `HoodThread` is instantiated.

The `run` method is always called with one parameter: a pointer to an object of type `HoodSerMach` — a "serial machine." Each process is associated with a serial machine, and when a process executes a thread, the serial machine that is associated with the process is passed into the thread's `run` method. `HoodSerMach` methods are used to perform all synchronization actions. For example, the `wait` method takes a pointer to a thread and an integer n, and it blocks the thread until it receives n signals. The `signal` method takes a pointer to a thread, and it sends a signal to that thread. This type of synchronization is implemented by giving each thread a join counter that represents the number of signals that the thread is waiting for. A ready thread has a join counter of 0. The `wait` method sets the join counter and blocks the thread. The `signal` method decrements the join counter, and when the join counter returns to 0, the thread is unblocked. Similar methods take a pointer to an object of type `HoodSemaphore` as a parameter and implement semaphore synchronization actions. In addition, the `HoodSerMach` class can be subclassed and augmented with process-specific data and methods.

To create multiple processes and their associated serial machines, a program instantiates an object of type `HoodParMach` — a "parallel machine." The `HoodParMach` constructor takes an integer argument P. It creates P serial machines, and it forks P processes. Initially, the P processes just park on a condition variable. After creating the parallel machine and one or more threads, the program calls the `HoodParMach` `runScheduler` method. The P processes are released from the condition variable, and they begin executing the scheduling loop, as described in the next section. When the scheduling loop terminates, they repark and `runScheduler` returns.

For Sun multiprocessors, Hood is built on top of the Solaris `thread` library, and it implements each process as a Solaris Light-Weight Process (LWP). As described in the next section, Hood schedules threads onto processes using the non-blocking work stealer. Hood has been instrumented to measure work and critical-path length. The work is measured by adding the elapsed time over each thread dispatch. Critical-path length is measured by timestamping [8]. Other statistics are also collected. The `Hood` library and user's manual can be obtained from `http://www.cs.utexas.edu/users/hood/`.

3 The non-blocking work stealer

`Hood`'s thread scheduler is a non-blocking implementation of the work-stealing algorithm. Idle processes steal threads from randomly chosen victims, and all concurrent data structures are implemented with non-blocking synchronization. In this section, we review the work-stealing algorithm and the non-blocking implementation, as described in [4].

3.1 The work-stealing algorithm

In the work-stealing algorithm, each process maintains its own pool of ready threads from which it obtains work, and when a process finds that its pool is empty, it becomes a thief and steals a thread from the pool of a victim process chosen at random. Each process's pool is maintained as a double-ended queue, or ***deque***, which has a bottom and a top. In the `Hood` implementation, each serial machine has a deque as one of its instance variables.

To obtain work, a process pops the ready thread from the bottom of its deque and commences executing that thread. That thread continues to execute until it either blocks or terminates, at which point the process goes back to the bottom of its deque to pop off another thread upon which it can work. During the course of its execution, if a thread creates a new thread or unblocks a blocked thread, then the process pushes the newly ready thread onto the bottom of its deque. Thus, so long as a process's deque is not empty, the process manipulates its deque in a LIFO (stack-like) manner.

When a process goes to obtain work by popping a thread off the bottom of its deque, if it finds that its deque is empty, then the process becomes a thief. It picks a victim process at random (using a uniform distribution) and attempts to obtain work by removing the thread at the top of the victim process's deque. If the victim process's deque is empty, then the thief picks another victim process and tries again. The thief repeatedly attempts to steal until it finds a victim whose deque is non-empty, at which point the thief reforms and commences work on the stolen thread as described above. Since steals take place at the top of the victim's deque, stealing operates in a FIFO manner.

3.2 The non-blocking implementation

We now describe `Hood`'s non-blocking implementation of the work-stealing algorithm. This implementation has two key features: the deques, which must support concurrent accesses, are implemented with non-blocking synchronization, and each process, between consecutive steal attempts, performs system calls to "yield" the processor.

In the non-blocking work stealer, the deques are implemented with non-blocking synchronization. That is, instead of using mutual exclusion, we use powerful atomic instructions, notably the SPARC v9 `casxa` (64-bit compare-and-swap) instruction. A complete description of this implementation can be found in [4]. This implementation is non-blocking, as opposed to wait-free [18], meaning that it is possible for a process to starve in its attempt to perform steal operations. Livelock, however, cannot occur because if one process starves, then others must be making progress. It turns out that wait-freedom is not needed to prove our analytical result [4] as stated in Section 5 — the non-blocking property is sufficient.

Other details of the implementation are also handled with non-blocking synchronization. For example, thread join-counters and semaphore thread-queues are implemented with non-blocking synchronization. Some of the instrumentation to measure critical-path length also uses non-blocking synchronization.

`mm`(n)	Multiply two dense $n \times n$ matrices of doubles using a blocked data layout. Each block is of size 16×16.
`lu`(n)	Compute LU-decomposition without pivoting of a dense $n \times n$ matrix of doubles using a blocked data layout. Each block is of size 16×16.
`barnes`(n, s)	Run Barnes-Hut n-body simulation [6] on n bodies for s time steps. This code is adapted from the SPLASH-2 [31] program, but for the `Hood` version, we parallelized the tree-building with a divide-and-conquer algorithm, so as to avoid the use of locks.
`heat`(n, m, s)	Simulate heat propagation on an $n \times m$ grid for s iterations using Jacobi iteration on a 5-point stencil. This application is similar to the SPLASH-2 Ocean program [31].
`msort`(n)	Merge sort n integers. Each recursive call is done in parallel, and in addition, the merging is done in parallel using a simple divide-and-conquer technique.
`ray`$()$	Raytrace scene to compute frame buffer of pixel colors. This application is adapted from the SPLASH-2 [31] program, and we use `balls4.env` as the scene to be rendered.

Table 1: Applications used in our study. All applications are written in `C++` and compiled with version 4.1 of the Sun `CC` compiler using flags `-xarch=v8plus -O5 -dalign -noex`. The `mm`, `lu`, `barnes`, and `heat` applications are all easily parallelized with a static load balancing, and our statically load-balanced versions are built directly on top of the Solaris `thread` library.

In addition to the use of non-blocking synchronization, the non-blocking work stealer also makes judicious use of "yields." Each process makes system calls to yield the processor between consecutive steal attempts. In `Hood`, we use a combination of `priocntl` (priority control) and `yield` system calls. Whenever a process becomes a thief, it calls `priocntl` to lower its priority. Once the thief has stolen a thread and reformed, it calls `priocntl` to restore its former priority. In addition, when a thief makes an unsuccessful steal attempt, it calls `yield`. In order to mitigate the high cost of these system calls, a thief delays its call to `priocntl` until after it has made enough unsuccessful steal attempts to amortize the cost of the `priocntl` call. Likewise, a thief calls `yield` only after it has made enough unsuccessful steal attempts to amortize the cost of the `yield` call.

Algorithmic and empirical analysis have both revealed the necessity of non-blocking synchronization and yields in eliminating the performance cliff [10]. If mutual exclusion is used instead of non-blocking synchronization, then a process that gets swapped out by the kernel while holding a lock can prevent other processes from making progress. Yields are needed to prevent the following anomaly. A process may get swapped out while in the middle of executing a thread. Other processes can still steal threads from the swapped-out process, but eventually the other processes reach a point where there are no ready threads available anywhere. Every other thread is blocked waiting for a signal from the thread that is being executed by the swapped-out process. This thread is not in the deque and cannot be stolen, so the other processes spin making unsuccessful steal attempts. By having these other processes call `yield` between steal attempts, they cause the swapped-out process to be swapped back in.

4 Performance of `Hood` applications

To evaluate `Hood` experimentally, we have implemented several parallel applications in `C++` using the `Hood` library. These applications are listed and described in Table 1. In addition, Table 2 gives a quantitative characterization of each application for a chosen input problem. Figure 1(b) (from Section 1) shows the speedup obtained for these applications with these inputs. Again, we point out that `Hood` applications do not suffer any performance cliffs. The performance curves are flat. In the next section, we report on more detailed experiments with many more input problems, and we provide a model that characterizes performance over the entire space of problem inputs, numbers of processes, and numbers of processors.

We are defining and measuring speedup as T_1/T, as opposed to T_s/T, where T_s denotes the execution

		T_s	T_1	T_1/T_s	T_∞	T_1/T_∞	T_8	T_1/T_8	T_s/T_8
`mm`(1024)	static	24.78	25.12	1.014			3.28	7.67	7.56
	`Hood`		25.36	1.023	0.01	2536	3.30	7.68	7.51
`lu`(2048)	static	66.85	60.26	0.901			9.41	6.41	7.11
	`Hood`		67.74	1.013	0.05	1394	9.07	7.47	7.37
`barnes`$(16384, 10)$	static	50.59	52.24	1.033			7.60	6.87	6.65
	`Hood`		52.04	1.029	0.51	102	7.41	7.02	6.83
`heat`$(4096, 512, 100)$	static	60.15	59.82	0.995			8.01	7.46	7.51
	`Hood`		60.04	0.998	0.23	264	7.93	7.57	7.59
`msort`$(32M)$	`Hood`	64.47	61.56	0.955	0.11	540	8.14	7.57	7.93
`ray`$()$	`Hood`	75.37	77.61	1.030	0.33	235	9.91	7.83	7.61

Table 2: Measured application characteristics. For each application, the row labeled "static" (when applicable) represents the statically load-balanced version of the application, and the row labeled "`Hood`" represents `Hood` version of the application. All times are in seconds. T_s is the execution time of a serial implementation. T_1 is the work of the computation — that is, the execution time with one process. For the `Hood` versions, T_∞ is the critical-path length. T_8 is the execution time with 8 processes running on 8 (dedicated) processors.

time for a (good) serial program. We use this definition in order to separate the performance effects of `Hood`'s scheduler implementation from the overheads induced by other aspects of the implementation. We shall refer to the ratio T_s/T as the ***application speedup*** in order to differentiate it from the ***computational speedup*** T_1/T, and we note that the two are related by $T_s/T = (T_1/T)/(T_1/T_s)$. The ratio T_1/T_s measures the ***overhead*** in our parallel implementation. It is the amount of work performed by the parallel computation divided by the amount of work performed by the serial computation. The computational speedup is independent of this overhead and measures the scheduler's ability to extract speedup from a computation. For convenience, we use the word "speedup" alone to denote the computational speedup.

From Table 2 we observe that the overhead T_1/T_s is near 1.0 for all of our applications. Thus, we are achieving efficient performance not just in a relative sense — that is, relative to T_1 — but in an absolute sense — that is, relative to T_s. For the dedicated case when we have $P_A = P$, `Hood` is performing just as well as static load balancing, and in the non-dedicated case when we have $P_A < P$, `Hood` is far outperforming static load balancing. `Hood` delivers almost perfect linear speedup — computational speedup as well as application speedup — in both cases.

Given the heavy cost of `priocntl` and `yield` system calls, it may come as a bit of a surprise that `Hood`'s non-blocking work stealer produces linear application speedup. An example of the "work-first" design principle [15], the key to this performance is the fact that these system calls occur only when a process is stealing, and this has two important consequences. First, the cost of these system calls does not show up as overhead T_1/T_s, because a 1-process execution never steals and consequently, never performs either of these system calls. Second, we know from prior analytical and empirical work [4, 8, 9] that the number of steals per process grows at most linearly with the critical-path length T_∞ and is independent of the amount of work T_1. Thus, when P is small relative to the parallelism T_1/T_∞, the execution incurs very few steals, and the cost of these system calls is negligible compared to the work per process. Effectively, the parallelism allows the cost of these system calls to be hidden by the amount of work per process, so linear computational speedup is achieved. The combination of low overhead and linear computational speedup means linear application speedup.

5 Performance modeling

In this section we show that Hood applications efficiently utilize whatever processor resources are provided by the kernel. Specifically, we show that whenever the number of processes is small relative to the parallelism, Hood applications obtain full utilization and linear speedup. For example, in Table 2, we see that for the chosen inputs, all six test applications have parallelism exceeding 100, so it is no surprise that the performance curves of Figure 1(b) remain flat up through 32 processes. This performance property holds even when the number of processors grows and shrinks arbitrarily over time.

Hood's non-blocking work stealer admits a simple performance bound based on work and critical-path length that has been proven analytically [4]. This result states that for any number P of processes, the expected execution time T is given by

$$T = O(T_1/P_A + T_\infty P/P_A) \ , \tag{1}$$

where T_1 is the work of the computation, T_∞ is the critical-path length of the computation, and P_A is the time-average number of processors that actually execute the computation. This analysis treats the kernel scheduler as an adversary with the one provision that it must obey yields.

To quantify this relationship, we replace the big-Oh notation with explicit constants. According to the asymptotic bound, there exist constants, c_1 and c_∞, such that the execution time is bounded by

$$T \leq c_1 T_1/P_A + c_\infty T_\infty P/P_A \ . \tag{2}$$

The bound, Inequality (2), has four independent variables — T_1, T_∞, P, and P_A — and we would like to show that this bound holds across all values of all of these variables. It turns out that we can perform a straightforward algebraic manipulation to derive a simpler bound that aggregates some of these variables.

Recall that we define the utilization as $T_1/(P_A T)$. If we plug Inequality (2) into this definition and divide the top and bottom by T_1, then we obtain the following lower bound for utilization:

$$\frac{T_1}{P_A T} \geq \frac{1}{c_1 + c_\infty P/(T_1/T_\infty)} \ . \tag{3}$$

Notice that the utilization is lower-bounded by a function of only one independent variable, $P/(T_1/T_\infty)$, that we call the ***normalized number of processes***. As the ratio of processes to parallelism, this quantity is the inverse of "parallel slackness" [28].

In the next two subsections, we investigate the relationship between utilization and the normalized number of processes for Hood applications. We begin in Section 5.1 with the case of fixed processor allocations, and then in Section 5.2, we consider the case when the number of processors grows and shrinks arbitrarily over time. In all cases, we find that Inequality (3) holds with small constants, c_1 and c_∞. Most importantly, we find that the constant c_1 is close to 1. Thus, we observe utilization near 1.0 whenever the normalized number of processes is sufficiently small. In other words, we observe linear speedup whenever the number of processes is sufficiently small relative to the parallelism.

5.1 Fixed processor allocations

Figure 2(a) shows the measured utilization plotted against the normalized number of processes for many runs of several Hood applications with different input parameters and with different numbers P of processes executed on a dedicated 8-processor machine. For any given run with P processes, we measure T_1, T_∞, and the execution time T. In addition, we have $P_A = \min\{8, P\}$. We then plot a data point at $(x, y) = (P/(T_1/T_\infty), T_1/(P_A T))$. We observe that, as predicted by the model, we obtain utilization near 1.0 so long as the normalized number of processes is small relative to 1. Unfortunately, we have little data in the

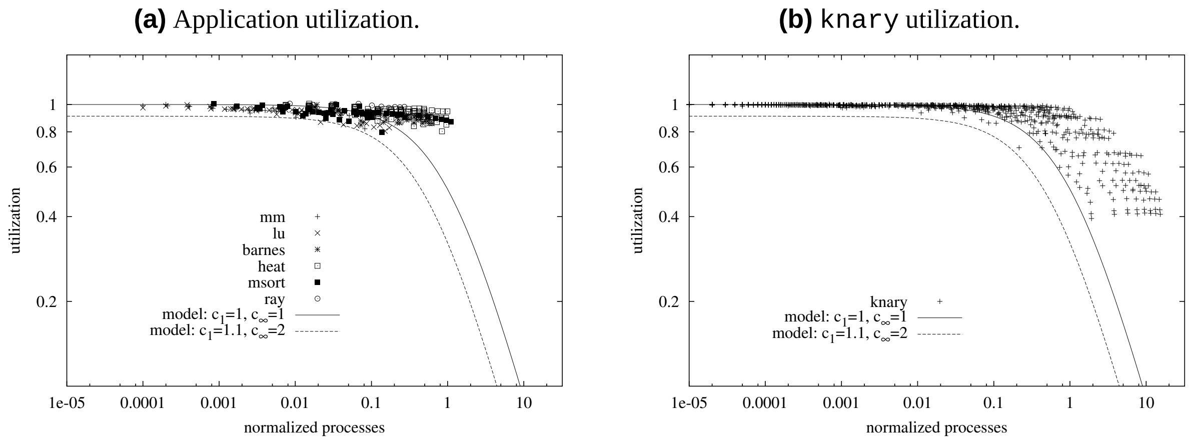


Figure 2: Measured utilization $T_1/(P_A T)$ plotted as a function of the normalized number of processes $P/(T_1/T_\infty)$ when run on a dedicated 8-processor machine. **(a)** Many runs of the `Hood` applications. **(b)** Many runs of the `knary` synthetic benchmark. The number P of processes ranges from 1 to 64. The work T_1 ranges from 1.2 seconds to 1371 seconds, and the critical-path length T_∞ ranges from 0.42 milliseconds to 99 seconds. Also shown are two curves defined by Inequality (3): the first with $c_1 = 1.0$ and $c_\infty = 1.0$, and the second with $c_1 = 1.1$ and $c_\infty = 2.0$.

regime where the normalized number of processes is large relative to 1. This limitation is a consequence of the fact that for all of our `Hood` applications, the input problems that generate reasonable values of T_1 tend to generate values of T_∞ within a narrow range. Thus, we shall continue our modeling study with a simple synthetic benchmark that is designed to generate arbitrary values of T_1 and T_∞.

The `knary`(h, d, s) synthetic benchmark grows a tree of height h and degree d in which for each non-leaf node, the first s children are generated serially and the remaining children are generated in parallel. When it generates a node, the program first executes a fixed number of iterations of an empty "for" loop before generating the children. Thus, we have $T_1 = \Theta(d^h)$, and by varying s in the range from 0 to d, the value of T_∞ will vary in the range from $\Theta(h) = \Theta(\log_d T_1)$ all the way up to $\Theta(T_1)$.

Figure 2(b) shows the measured utilization plotted against the normalized number of processes for many runs of `knary` with different input parameters and with different numbers P of processes executed on a dedicated 8-processor machine. The plotted data points represent a range of values of P from 1 to 64 while the work T_1 and critical-path length T_∞ range over more than 3 orders of magnitude, with work values as small as 1.2 seconds. Again, we observe utilization near 1.0 so long as the normalized number of processes is small. In addition, we now see that as the normalized number of processes rises above 1, the utilization drops off.

Also plotted in Figure 2 are two curves defined by the lower bound, Inequality (3). The first curve uses constants $c_1 = 1.0$ and $c_\infty = 1.0$, and the second curve uses constants $c_1 = 1.1$ and $c_\infty = 2.0$. Even with these modest values for the constants, these curves do a good job of lower bounding the utilization. Moreover, we observe that these lower-bound curves are quite tight in the regime where the normalized number of processes is small relative to 1. These lower-bound curves become less tight as the normalized number of processes grows. As the normalized number of processes gets large, this spread that we observe in the plotted data reveals the conservative nature of our model. Our analytical upper bound of Equation (1) is proven in a setting where the kernel scheduler is assumed to be an adversary. Though this assumption makes our results widely applicable, it also may be overly pessimistic. If Equation (1) is conservative, then our lower bound on utilization, Inequality (3), is also conservative, which is exactly what we observe in the plotted data.

The vast majority of the data plotted in Figure 2 are derived from runs in which the number P of processes exceeds the number P_A of processors used. Nevertheless, `Hood` applications achieve high utilization provided that the number of processes is reasonably small compared with the parallelism. This behavior holds over a dramatic range of problem inputs, with one number, the normalized number of processes, giving a lower bound on the utilization, as predicted by the model.

5.2 Variable processor allocations

The experiments of the previous subsection were run on a dedicated 8-processor machine, so our test programs ran on a dedicated set of $P_A = \min\{8, P\}$ processors. In these experiments, we observed the performance effects of having more processes than processors. We now consider a more dynamic setting in which `Hood` applications run on a set of processors that grows and shrinks over time. In this setting, P_A is the time-average actual number of processors on which the program runs, and we now wish to show that our performance model of Inequalities (2) and (3) continues to hold with this more general definition of P_A.

The difficulty in repeating the experiments of the previous section for the case when the number of processors grows and shrinks is that it is hard to measure P_A. We found that using the Solaris kernel's TNF probes was far too intrusive. In addition, if we run a `Hood` application concurrently with some arbitrary other application, then that other application may affect the `Hood` application in a manner that has nothing to do with the focus of this experiment — processor utilization. For these reasons we chose to build a synthetic application that uses almost no resources besides processor resources and admits estimation of P_A.

The `cycler`(p, w, W) synthetic application eats up a time-varying number of processor cycles in a manner that allows us to estimate the time-average number of processor cycles that it uses over any period of time. The `cycler`(p, w, W) application operates as follows. First, the main process forks p subordinate processes which park on a condition variable, and then the main process repeats the following iteration. It releases a number of subordinate processes chosen at random in the range from 1 to p, and then it waits for those processes to repark. A subordinate process that is released chooses a number at random in the range from 1 to w, and then it performs that number of increments to a shared counter before reparking. Between each increment of the shared counter, the process executes a fixed number n_1 of iterations of an empty "for" loop. The shared counter is implemented with non-blocking synchronization, using the SPARC v9 `casa` instruction. After each increment, the process checks to see if the counter value is a multiple of some fixed number n_2, and if so, it writes the counter value and a (wall-clock) timestamp into a buffer that gets flushed to a file when execution terminates. Execution terminates when the main process finds that, after an iteration has completed, the counter is at least W. In summary, at each iteration, a randomly chosen number of processes executes a randomly chosen number of counter increments, and every time the counter reaches a multiple of n_2, a timestamp is written. The fixed numbers n_1 and n_2 are chosen so that a process will increment the counter roughly every few hundred microseconds, and a process working alone will write a timestamp roughly every few milliseconds. Thus, `cycler` uses almost no memory bandwidth, and the overhead of writing timestamps is negligible.

After calibration, we can estimate the time-average number $P_A($`cycler`$)$ of processors being used by `cycler` over any (reasonable-length) period of time. For calibration, we run `cycler` with $p = 1$ on a dedicated machine with a large value of w and $W = 1$, so the program will run for 1 iteration with a single process incrementing the counter some large number of times. By looking at the execution time t and the counter value v at the end, we can compute that `cycler` runs at $r = v/t$ increments per second per processor. With the calibration done, we can now run `cycler` using arbitrary values of p and w concurrently with other programs, and over any interval of time, we can estimate $P_A($`cycler`$)$ as follows. For any two timestamps with times t_1 and t_2 and counts v_1 and v_2, the time-average number of processors used by `cycler` over the interval of time from t_1 to t_2 is given by $P_A($`cycler`$) = ((v_2 - v_1)/(t_2 - t_1))/r$.

Figure 3 shows the measured utilization for many executions of `Hood` applications, with each execution running concurrently with `cycler`. The applications were all run with many different input values, and

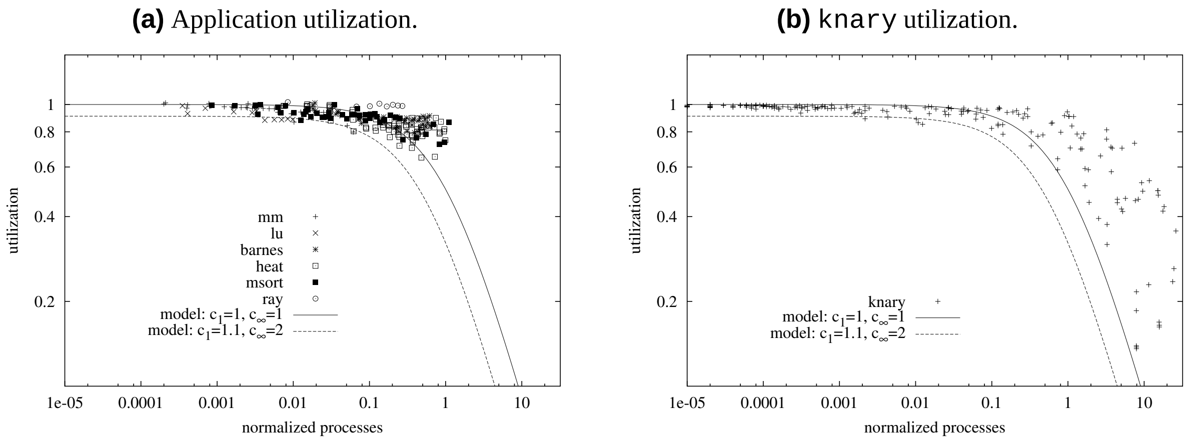


Figure 3: Measured utilization $T_1/(P_A T)$ plotted as a function of the normalized number of processes $P/(T_1/T_\infty)$ when run on an 8-processor machine simultaneously with the `cycler` program. The time-average number of processors $P_A(\texttt{cycler})$ consumed by `cycler` ranges from 0.25 to 4.9, with instantaneous consumption ranging from 0 to 8. Also shown are two curves defined by Inequality (3): the first with $c_1 = 1.0$ and $c_\infty = 1.0$, and the second with $c_1 = 1.1$ and $c_\infty = 2.0$.

`cycler` was also run with many different input values. As in the experiments of the previous section, the normalized number of processes is $P/(T_1/T_\infty)$, and the utilization is $T_1/(P_A T)$. The difference is that now to compute P_A, we must account for the processors being used by `cycler`. Thus, P_A is given by $P_A = \min\{P, 8 - P_A(\texttt{cycler})\}$. Again, we find that one number, the normalized number of processes, predicts the utilization behavior. Moreover, it does so even when the program runs on a set of processors that grows and shrinks over time.

6 Related Work

Though numerous threads libraries [11, 19, 24, 29] and multithreaded languages [3, 15, 17] have been developed, we are not aware of any besides `Hood` that do not suffer from the performance cliff. Like `Hood`, some of these libraries and languages are based on the two-level scheduling model, with user-level threads scheduled onto a pool of processes. And like `Hood`, some of these libraries and languages are based on algorithms that are provably efficient, at least for the case when the number of processes equals the number of processors. Typically, however, these systems implement the scheduler with mutual exclusion, sometimes even using spin locks, and consequently, when the number of processes exceeds the number of processors, they fall off the performance cliff.

It is possible that the performance cliff might be eliminated through the use of first-class user-level threads [21] or scheduler activations [2], possibly in conjunction with preemption-safe locking [1, 7, 23], but we have not investigated any such approach.

Prior efforts to eliminate the performance cliff have focused on the management and scheduling of kernel-level resources, specifically processes [12, 16, 22, 25, 27, 30]. Many of these studies have concluded that some form of coscheduling or space partitioning with process control offers the best solution.

Coscheduling [25], which is a generalization of “gang scheduling,” attempts to run all of the processes of any given parallel program concurrently as a “gang,” thereby giving each program the illusion of running on a dedicated machine. Interestingly, it has been shown recently that in a network of workstations, coscheduling can be achieved implicitly with little or no modification to existing kernel schedulers [13, 26]. The main advantage of coscheduling over our approach is that coscheduling may be able to achieve “superlinear" speedup due to caching effects. Some programs require a large amount of cache resource due to large working sets. Such a program runs poorly on one processor and will benefit from superlinear speedup once sufficiently many processors are employed so that the working set fits within their collective caches. Whereas coscheduling may produce this superlinear speedup, Hood's non-blocking work stealer can only achieve speedup that is linear relative to the poor one-processor performance. The main drawback to coscheduling, whether explicit or implicit, is that it cannot be applied effectively for some job mixes. Consider, for example, a parallel program with 8 processes running concurrently with a serial program on a 8-processor machine. While the serial program is executing on a processor, we can either leave the other 7 processors idle or run 7 of the parallel program's 8 processes. In the former case, we are leaving most of the processors idle. In the latter case, we may fall off the performance cliff.

As an alternative to the above scenario, the process control approach [27] would have the parallel program kill one of its processes. In general, with process control, a parallel program creates and kills processes dynamically so that it continuously runs with a number of processes equal to the number of processors available to it. This strategy requires some kernel-level support, so that programs can be informed as to how many processes they should have.

Our experience with Hood, however, suggests a different strategy. As we found in Section 5, the key to performance is not the relationship between processes and processors. It is the relationship between processes and parallelism. Having more processes than processors is fine. In fact, for the purpose of overlapping I/O with computation, this situation is often desirable. The key to efficiency is keeping the number of processes small relative to the parallelism. The suggested strategy then is to have the runtime system monitor the parallelism and automatically create and kill processes to maintain the appropriate relationship between processes and parallelism. We are currently investigating strategies to measure parallelism automatically. Our goal is that Hood applications will be turnkey — that is, they will run without requiring that users specify the number of processes on the command line.

7 Conclusion

Hood applications efficiently utilize whatever processor resources are provided by the kernel. So long as the number of processes is small relative to the parallelism, Hood applications deliver full utilization and linear speedup. This result holds even when the number of processes exceeds the number of processors and even when the number of processors grows and shrinks over time. Hood eliminates the performance cliff. In the case of a dedicated, non-multiprogrammed environment, Hood performs as well as statically load-balanced solutions, while far outperforming the static solutions in non-dedicated, multiprogrammed environments. In a dynamic, multiprogrammed environment, even programs that can use static load balancing shouldn't. Hood achieves these results with a user-level implementation and without coscheduling or process control, and we have demonstrated these results using some of the very same applications that have been used in the past to argue for coscheduling and process control.

Acknowledgments

Coming up with a correct non-blocking implementation of the work-stealing algorithm was not easy, and we have several people to thank. Keith Randall of MIT found a bug in an early version of our algorithm, and Mark Moir of The University of Pittsburgh, suggested ideas that lead us to a correct algorithm. We acknowledge Nimar Arora and Greg Plaxton of UT Austin, our collaborators on the analysis of the non-blocking work-stealing algorithm, and we thank Charles Leiserson and Matteo Frigo of MIT for listening patiently as we tried to hash out some of our early ideas. Thanks also to Umut Acar and Emery Berger of UT Austin for providing assistance and feedback throughout. Finally, we express our great appreciation to

Patti Spencer and Boyd Merworth of UT Austin for all of their technical assistance with the running of our experiments.

References

[1] Juan Alemany and Edward W. Felten. Performance issues in non-blocking synchronization on shared-memory multiprocessors. In *Proceedings of the Eleventh Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 125–134, Vancouver, British Columbia, Canada, August 1992.

[2] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles (SOSP)*, pages 95–109, Pacific Grove, California, October 1991.

[3] Ken Arnold and James Gosling. *The Java Programming Language*. Addison-Wesley, 1996.

[4] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 119–129, Puerto Vallarta, Mexico, June 1998.

[5] Remzi H. Arpaci, Andrea C. Dusseau, Amin M. Vahdat, Lok T. Liu, Thomas E. Anderson, and David A. Patterson. The interaction of parallel and sequential workloads on a network of workstations. In *Proceedings of the 1995 ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, pages 267–278, Ottawa, Canada, May 1995.

[6] J. Barnes and P. Hut. A hierarchical $O(N \log N)$ force-calculation algorithm. *Nature*, 324:446–449, 1986.

[7] Brian N. Bershad. Practical considerations for non-blocking concurrent objects. In *Proceedings of the 13th International Conference on Distributed Computing Systems (ICDCS)*, May 1993.

[8] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, August 1996.

[9] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 356–368, Santa Fe, New Mexico, November 1994.

[10] Robert D. Blumofe and Dionisios Papadopoulos. The performance of work stealing in multiprogrammed environments. Technical Report TR-98-13, The University of Texas at Austin, Department of Computer Sciences, May 1998.

[11] Eric C. Cooper and Richard P. Draves. C Threads. Technical Report CMU-CS-88-154, School of Computer Science, Carnegie-Mellon University, June 1988.

[12] Mark Crovella, Prakash Das, Czarek Dubnicki, Thomas LeBlanc, and Evangelos Markatos. Multiprogramming on multiprocessors. In *Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing*, December 1991.

[13] Andrea C. Dusseau, Remzi H. Arpaci, and David E. Culler. Effective distributed scheduling of parallel workloads. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 25–36, Philadelphia, Pennsylvania, May 1996.

[14] Dror G. Feitelson and Larry Rudolph. Coscheduling based on runtime identification of activity working sets. *International Journal of Parallel Programming*, 23(2):135–160, April 1995.

[15] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Montreal, Canada, June 1998.

[16] Anoop Gupta, Andrew Tucker, and Shigeru Urushibara. The impact of operating system scheduling policies and synchronization methods on the performance of parallel applications. In *Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1991.

[17] Robert H. Halstead, Jr. Implementation of Multilisp: Lisp on a multiprocessor. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 9–17, Austin, Texas, August 1984.

[18] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 11(1):124–149, January 1991.

[19] David Keppel. Tools and techniques for building fast portable threads packages. Technical Report UWCSE93-05-06, University of Washington, 1993.

[20] Steve Kleiman, Devang Shah, and Bart Smaalders. *Programming with Threads*. SunSoft Press, Prentice Hall, 1996.

[21] Brian D. Marsh, Michael L. Scott, Thomas J. LeBlanc, and Evangelos P. Markatos. First-class user-level threads. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles (SOSP)*, Pacific Grove, California, October 1991.

[22] Cathy McCann, Raj Vaswani, and John Zahorjan. A dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 11(2):146–178, May 1993.

[23] Maged M. Michael and Michael L. Scott. Relative performance of preemption-safe locking and non-blocking synchronization on multiprogrammed shared memory multiprocessors. In *Proceedings of the 11th International Parallel Processing Symposium (IPPS)*, Geneva, Switzerland, April 1997.

[24] F. Mueller. A library implementation of POSIX threads under UNIX. In *Proceedings of USENIX Conference, Winter '93*, pages 29–41, 1993.

[25] John K. Ousterhout. Scheduling techniques for concurrent systems. In *Proceedings of the 3rd International Conference on Distributed Computing Systems*, May 1982.

[26] Patrick G. Sobalvarro and William E. Weihl. Demand-based coscheduling of parallel jobs on multiprogrammed multiprocessors. In *Proceedings of the IPPS '95 Workshop on Job Scheduling Strategies for Parallel Processing*, April 1995.

[27] Andrew Tucker and Anoop Gupta. Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles (SOSP)*, pages 159–166, Litchfield Park, Arizona, December 1989.

[28] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[29] Mark T. Vandevoorde and Eric S. Roberts. WorkCrews: An abstraction for controlling parallelism. *International Journal of Parallel Programming*, 17(4):347–366, August 1988.

[30] Raj Vaswani and John Zahorjan. The implications of cache affinity on processor scheduling for multiprogrammed, shared memory multiprocessors. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles (SOSP)*, Pacific Grove, California, October 1991.

[31] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA)*, pages 24–36, Santa Margherita Ligure, Italy, June 1995.