CS 484: Project 2

PThread Barriers, Log Barriers, Busy-Wait Barriers

Duane Johnson
Jan 30, 2009

This report explores the implementation of the following 3 types of barriers within the context of a multithreaded "pthread" application and compares their measured (empirical) efficiencies:

1. a pthread linear "join" barrier
2. a log barrier, and
3. a busy-wait (spin) barrier

PThread Linear Join Barrier

The implementation of the linear "join" barrier was an adaptation of a predecessor project: where previously the OpenMP library had been used to automatically vectorize the main loop in a "hotplate" algorithm, this implementation attempted to mimic that solution directly using pthread_* calls.

Within the main loop of this implementation, several threads (equal to the number of processors on the host machine) are created at the beginning of each iteration. Subsequently, the threads are "joined" together after their work is done at the end of each iteration, resulting in a de facto linear barrier. As shown on the graph, this was the slowest solution to the problem: a single-processor machine took 2.24 seconds while a 16-processor machine clocked in at 0.86 seconds, just a little less than half the time. The bottleneck in this solution is clearly the overhead involved in creating and joining so many threads.

Log Barrier

The log barrier implementation uses pthreads, but creates only the necessary threads (one per processor) at the beginning of the program. Thus, during each iteration of the main loop, the processor(s) are not burdened with the overhead of thread maintenance.

A significant improvement in speed was achieved in part due to this lack of overhead, and also due to the log(n) efficiency class of the log barrier (hence the name). Each thread communicates with one other thread during each clock-tick, thus exponentially improving the time necessary to broadcast the barrier "go" signal. A single-processor machine took 1.54 seconds to complete, while a 16-processor machine took a mere 0.31 seconds on average, nearly a 5-fold improvement.

Busy-Wait (Spin) Barrier

The final graph on the next page shows a "busy-wait" barrier which, like the log barrier, has little thread overhead. In addition, it uses no semaphores or mutexes and is therefore more suited to a dedicated processor environment.

This implementation of the busy-wait barrier could possibly be improved since it does not use a "last man in" approach to signaling; rather, it uses the "main thread" (thread #0) to be the watcher thread. Since the watcher thread is not guaranteed to be last, it is possible that the detection of the entry of the final thread into the barrier is a little later than it could be.

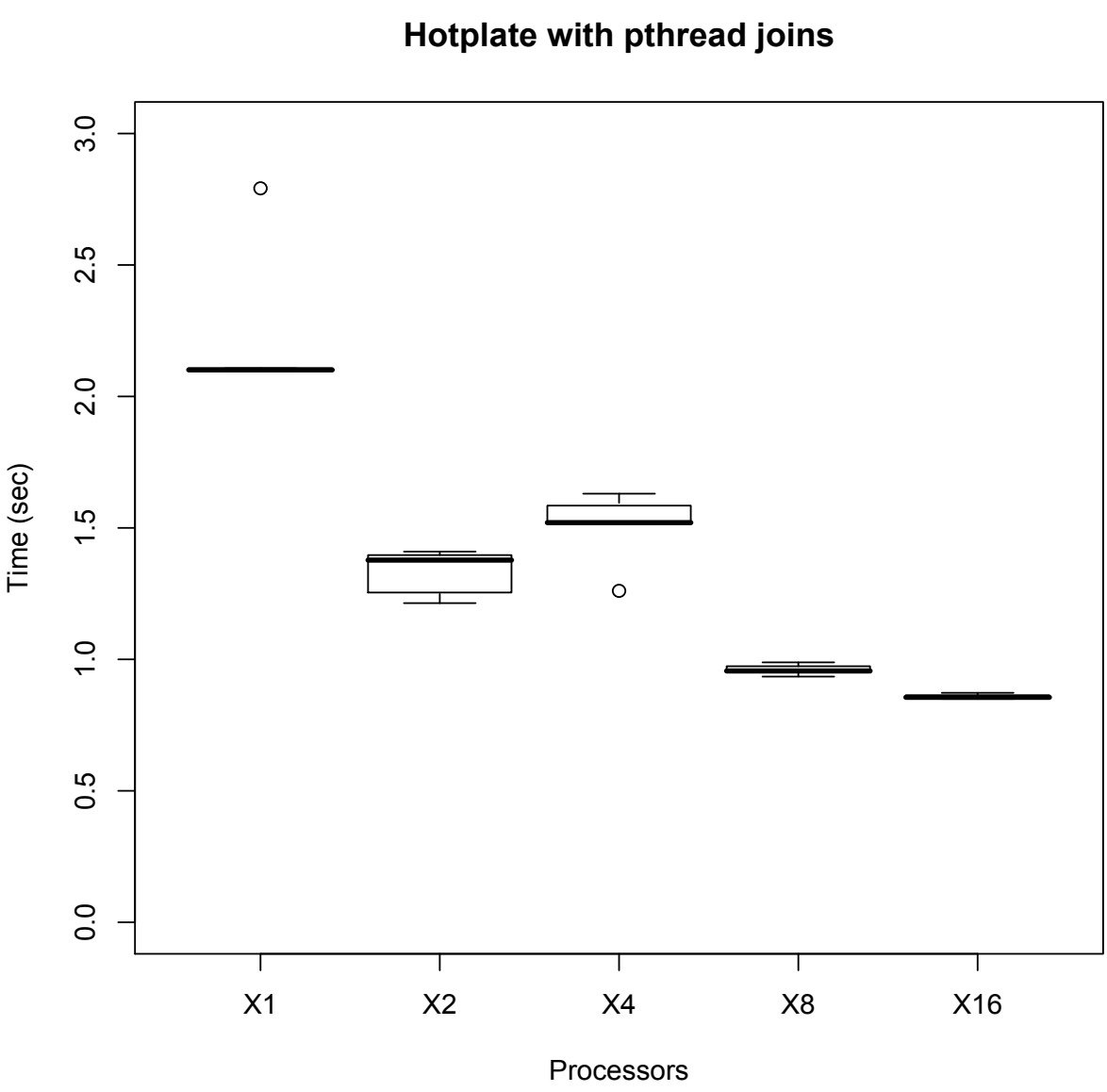
The speed benefit of the busy-wait barrier was only slightly better than the log barrier for fewer processors, but improved at the higher end: a single-processor system took an average of 1.54 seconds while a 16-processor machine took 0.13 seconds. This is a 12x improvement, nearly linear as a function of processor count.

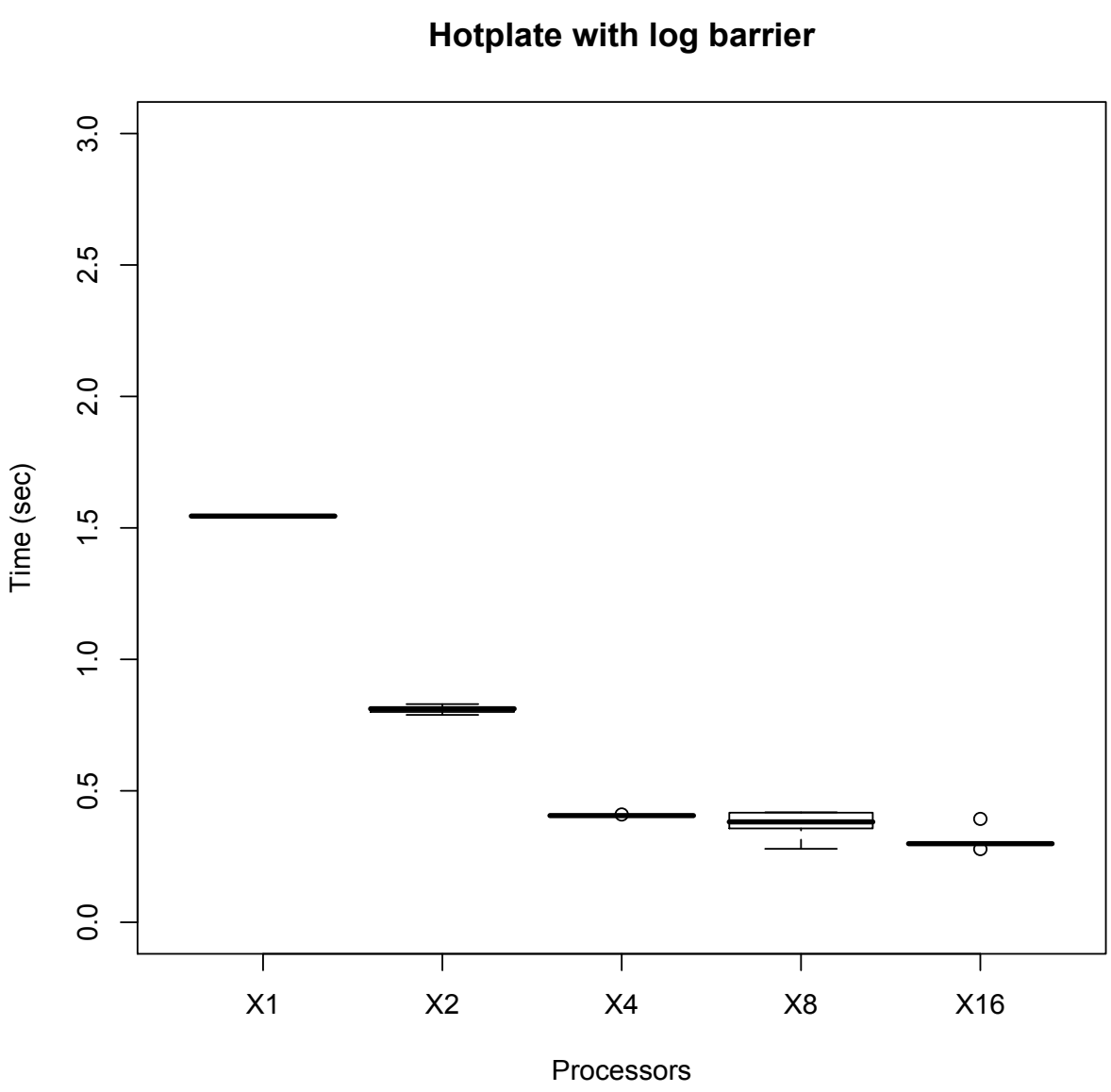
Conclusion

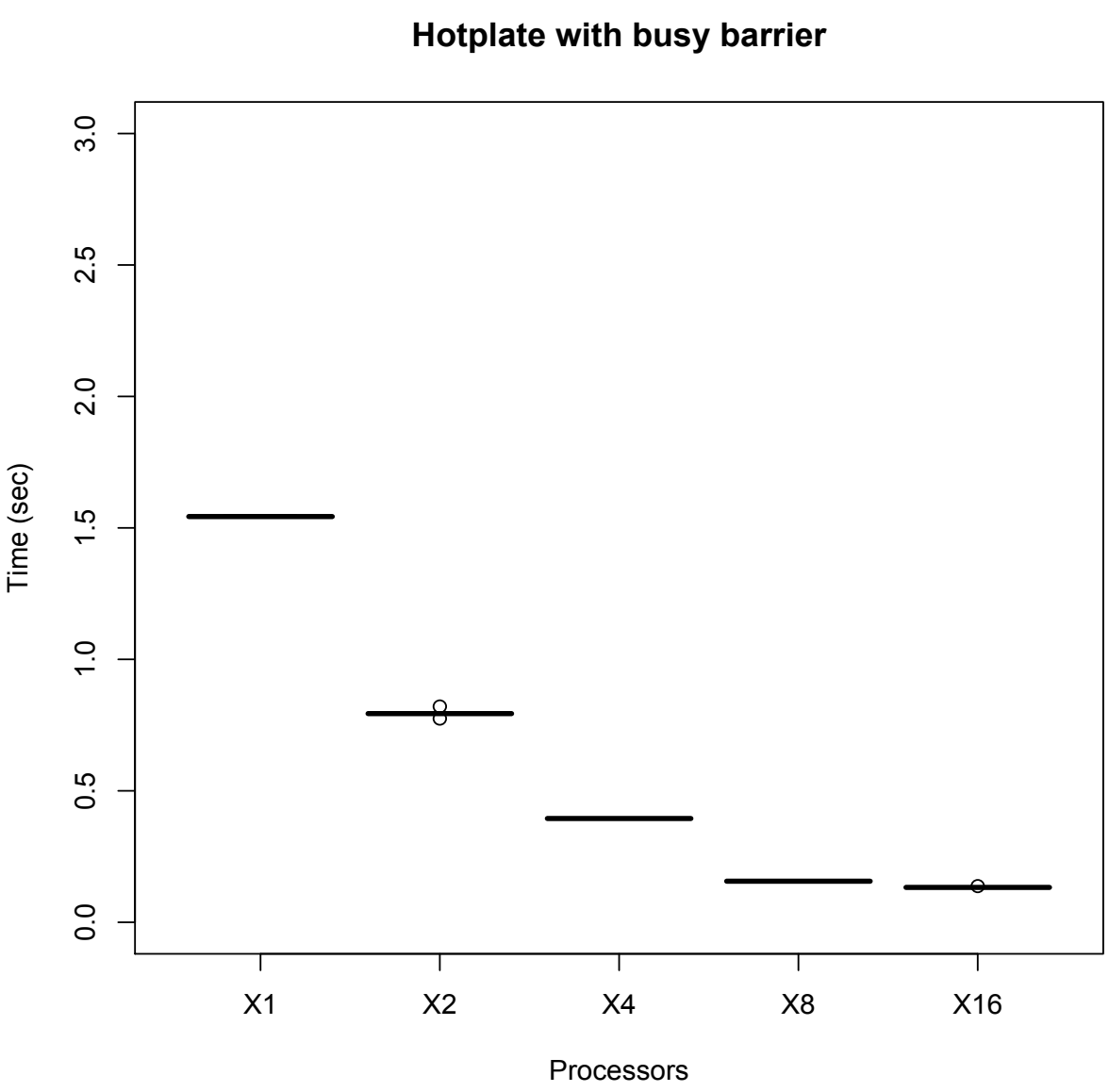
While the pthread "join" barrier was the easiest to implement, it was also the worst-performing of the trio. In addition, it had the greatest variation in performance, even with the same number of processors (for example, on a 2-processor machine, the range of times to completion was 1.2 to 1.4 seconds).

The log barrier was superior to the join barrier in almost every respect: it was faster, more reliable, and improved better with parallelization than the join barrier.

While the busy-wait barrier was the best of all, it is somewhat impractical for most uses except very intensive computations that have full control over the system on which they run. Nevertheless, the busy-wait barrier was the clear winner in the area of pure performance and predictability.


Hotplate with pthread joins
Time (sec)
0.0
0.5
1.0
1.5
2.0
2.5
3.0
X1
X2
X4
X8
X16
Processors


Hotplate with log barrier
Time (sec)
0.0
0.5
1.0
1.5
2.0
2.5
3.0
X1
X2
X4
X8
X16
Processors


Hotplate with busy barrier
Time (sec)
0.0
0.5
1.0
1.5
2.0
2.5
3.0
X1
X2
X4
X8
X16
Processors