Ocean Warning: Avoid Drowning

Mark Heinrich
School of EECS
University of Central Florida
Orlando, FL 32816
Email: heinrich@cs.ucf.edu

Mainak Chaudhuri
Computer Systems Laboratory
Cornell University
Ithaca, NY 14853
Email: mainak@csl.cornell.edu

***Abstract*— Ocean is a popular program from the SPLASH-2 parallel benchmark suite. A complete application, as opposed to a computational kernel, Ocean is often used as a representative of a well-tuned parallel program in architectural studies. However, we find there is a danger in using Ocean to evaluate proposed enhancements that purport to either improve scalability or reduce synchronization overhead. The default Ocean code contains an ill-advised code segment that seriously handicaps its "base" performance in common architectural studies. We provide the one-line fix for the offending code that improves performance by as much as a factor of 2.3, and suggest that architecture researchers using Ocean to evaluate their new ideas—especially when discussing scalability or synchronization—change their code immediately.**

I. Introduction

Ocean is an oft-used program from the SPLASH-2 parallel application suite [3]. While no claim was ever made that the SPLASH-2 programs are the most optimal parallel versions of those applications, a serious effort was made to not handicap the codes or prevent their scalability so that they would be of general use to the architecture community. The care taken in Ocean is case in point. Ocean uses a 4-D, as opposed to a simpler 2-D, decomposition to ease page placement requirements on ccNUMA machines. Ocean is also structured so that reads are remote and writes are local, and so that prefetches can be easily inserted to mitigate the impact of large remote read latencies.

Unfortunately Ocean contains one vile code segment that does much to undo its otherwise careful crafting, hampering both its scalability and unduly increasing the percentage of execution time spent in synchronization primitives. In Section II we provide the one-line fix that significantly decreases synchronization time in Ocean (inside both locks and barriers) and correspondingly improves its performance and scalability. Architecture studies using the original unnecessarily crippled version of Ocean may result in misleading conclusions about the efficacy of the proposed scheme, especially in studies related to improvements in scalability or synchronization that happen to address or remove the same inefficiency that should have been removed in the code to begin with. We strongly urge architects to remove the paper tiger and apply the one-line patch in Section II to Ocean to yield more meaningful comparisons of their architectural ideas.

In the next section we detail the problem and provide the line of code that fixes it. Section III presents simulation results for a variety of common configurations of both the original "unoptimized" Ocean and the new version with our fix properly applied. We conclude in Section IV.

II. Problem

The problem in Ocean stems from the use of a lock that protects the global error value `multi->err_multi`. During this phase of Ocean, each processor computes its local error (`local_err`) and tests to see if its local error is larger than the global error. If so, it sets the new global error to the larger local error value. The code to perform the error update resides in file `multi.C` in the procedure `multig`:

```
LOCK(locks->error_lock)
if (local_err > multi->err_multi) {
   multi->err_multi = local_err;
}
UNLOCK(locks->error_lock)
```

The code first acquires a lock, then performs the error test and possible set, then releases the lock. As the number of processors increases, it is obvious that this lock can become highly-contended and negatively impact performance. Unfortunately, the lock is necessary for correctness, since the update of the global error must be atomic. However, it is a relatively rare event for the local error to be greater than the global error (see Section III for statistics). That fact, coupled with the observation that the global error is monotonically increasing, means that the simple lock, test, and set sequence above, can trivially be changed to a test, lock, test, and set sequence as follows:

```
if (local_err > multi->err_multi) {
   LOCK(locks->error_lock)
   if (local_err > multi->err_multi) {
      multi->err_multi = local_err;
   }
   UNLOCK(locks->error_lock)
}
```

The corrected code is identical except that the test inside the critical section is also performed *before* acquiring the lock. This has the effect of entirely eliminating most of the acquire attempts for the error lock without changing the program behavior, and as we shall see in the next section, dramatically improves the performance of Ocean. Note that the test must also be done a second time inside the critical section for correctness.

III. Simulation Results

Using the multi-threaded execution-driven simulator for ccNUMA DSM machines that was validated against real hardware in [1], with the parameters given in Table I, we present Ocean simulation results for both the original code and the version with our fix properly applied. The simulated machine runs a directory-based cache coherence protocol similar to that in the SGI Origin 2000 [2]. In all cases we use the standard Ocean parameters of 20km grid resolution, 8hr relaxation time, and relaxation tolerance of $1e^{-5}$. We also enable the standard prefetching and data placement options.

TABLE I
Simulation parameters

Processor speed	2GHz
ITLB	8 entries/Fully Assoc./Random
DTLB	64 entries/Fully Assoc./Random
Page size	4KB
L1 I$	32KB/64B lines/2-way/LRU
L1 D$	32KB/32B lines/2-way/LRU
L2 cache	2MB/128B lines/2-way/LRU
Memory access time	90ns
Topology	2-way bristled hypercube
Router hop time	25ns

Figure 1 shows the results for 514x514 Ocean, a problem size that scales to moderately large machines (64-128 processors) that is often used in architectural studies. For each processor count, the graph shows execution times for `UNOPT` (the original SPLASH-2 code) and `OPT` (our fixed version) normalized to the performance of the original code. Further, each execution time bar is broken down into processor busy time, memory stall time, barrier stall time, and lock stall time.

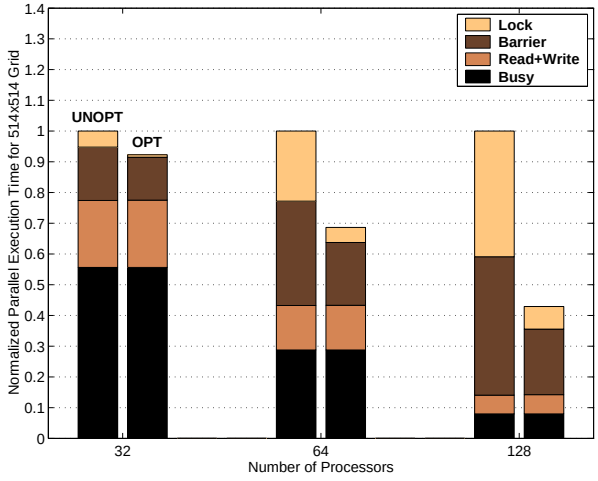

Fig. 1. Effects of optimization on 514x514 Ocean

The first thing to note from Figure 1 is that the optimized Ocean code significantly improves execution time and its effect becomes more important as the machine size scales. At 32 processors the optimized code is 8.4% faster than the original code, at 64 processors it is 45.7% faster, and at 128 processors it is 133% (2.3 times) faster. Note that as expected, this optimization has no effect whatever on processor busy times or memory stall times. However, lock stall time is dramatically reduced for all three processor counts. Less obviously, the barrier stall time is also reduced significantly because of improved load balance resulting from the conversion of a heavily-contended lock to a lock that is mostly free. The combination of these effects yields impressive performance improvement. We argue that it is this performance of Ocean that should be treated as the "base" performance in architecture studies that use Ocean to evaluate new designs or techniques, especially when related to reducing synchronization overhead or improving scalability.

Table II explains why such large reductions in synchronization times are possible. The table lists the percentage of time the global error lock is actually updated when the original code grabs the error lock for the problem sizes used in both Figures 1 and 2. As the table shows, the global error is updated infrequently, and therefore most of the lock acquires ($> 80\%$ in all cases and $> 92\%$ in all cases but one) in the original code are useless and serve only to slow down the program. Generally, the update frequency decreases with increasing problem size and machine size, making our optimization all the more effective.

TABLE II
Dynamic error update frequency

Problem Size	Processor Count	Frequency
130x130	16	18.23%
258x258	64	4.89%
258x258	64 (small caches)	6.39%
514x514	32	7.81%
514x514	64	3.85%
514x514	128	2.16%

Figure 2 shows results for Ocean on other "popular" problem sizes. Because Ocean simulations can be long-running, some studies use undersized problem sizes to save time. For small processor counts, the woefully small problem size of 130x130 is sometimes used, and we show the results of our optimization here for 16 processors, where the fixed code is 14.5% faster than the original. Though too small to scale well, a 258x258 grid is sometimes used at larger machine sizes. We present that problem size here for 64 processors, where the fixed code is 79.2% faster. In recognition of running the smaller grid size, some studies simultaneously run with "small caches" to get the effect of running a bigger problem on normally-sized caches. We show 258x258 with a 64 KB secondary cache and 16 KB primary caches, where the fixed code is 27.7% faster.

In all cases our optimization drastically reduces both lock stall time and barrier stall time by reducing the number of lock attempts in this critical parallel section of the Ocean code. The performance effects are so drastic that we view our code change more as a fix of a glaring omission in the original code than as an optimization.

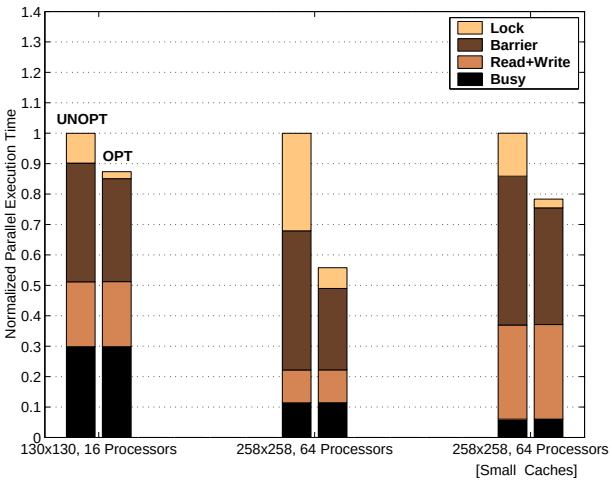

Fig. 2. Effects of optimization on smaller Ocean problems

IV. Conclusions

We show that the lack of a simple test before acquiring the global error lock in Ocean significantly increases the synchronization time, especially as you scale the size of the machine. Across several commonly-used problem size/processor count configurations, we show how adding the one-line test decreases not only the amount of time spent in lock acquires (obviously) but also significantly decreases the barrier stall time because of the resulting improved load balance. Analysis of architectural enhancements that improve scalability or synchronization overhead can be clouded by the use of the original Ocean code. So do not drown the analysis of your architectural ideas in an Ocean crippled by the unfortunate absence of a single line of code. Change your code, and full steam ahead!

References

[1] J. Gibson et al. FLASH vs. (Simulated) FLASH: Closing the Simulation Loop. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 49–58, November 2000.

[2] J. Laudon and D. Lenoski. The SGI Origin: a ccNUMA highly scalable server. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 241–251, June 1997.

[3] S. C. Woo et al. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, June 1995.