Brief Announcement: Reduced I/O Latency with Futures

Kyle Singer
Washington University in St. Louis
kdsinger@wustl.edu

Kunal Agrawal
Washington University in St. Louis
kunal@wustl.edu

I-Ting Angelina Lee
Washington University in St. Louis
angelee@wustl.edu

ABSTRACT

Task parallelism research has traditionally focused on optimizing computation-intensive applications. Due to the proliferation of commodity parallel processors, there has been recent interest in supporting interactive applications. Such interactive applications frequently rely on I/O operations that may incur significant latency. In order to increase performance, when a particular thread of control is blocked on an I/O operation, ideally we would like to hide this latency by using the processing resources to do other ready work instead of blocking or spin waiting on this I/O. There has been limited prior work on hiding this latency. As far as we are aware, only one prior work exists that provides a theoretical bound for interactive applications that use I/Os.

In this work, we propose a method for hiding the latency of I/O operations by using the futures abstraction. We provide better execution time guarantees using this method than prior work. We also implemented the algorithm in a practically efficient prototype library that runs on top of the Cilk-F runtime, a runtime system that supports futures within the context of the Cilk Plus language, and performed experiments that demonstrate the efficiency of our implementation.

1 INTRODUCTION

With the prevalence of multicore processors, task parallelism has become increasingly popular. With task parallelism, the programmer expresses the *logical* parallelism of the computation and lets an underlying runtime system handle the necessary load balancing and synchronizations. Modern parallel platforms that implement task parallelism include but are not limited to OpenMP [21], Intel TBB [15], various dialects of Cilk [12, 16, 18, 19] and Habanero [7, 10], X10 [11], and the Java Fork/Join framework [17]. These platforms often schedule parallel computations using work stealing, which provides provable bounds on execution time [5, 6, 8, 9], good space bounds [9], good cache locality [3, 4], and allows for an efficient implementation [13].

Research on task parallel platforms has traditionally focused on optimizing for compute-intensive and throughput-oriented applications, such as ones found in the domain of high-performance and scientific computing. Multicore processors have become commonplace and are used in personal computers and servers, however, and a fundamental component of desktop software is its frequent interactions with the external world, done in the form of input/output (I/O), such as obtaining user input through key strokes or mouse clicks, waiting for a data packet to arrive on a network connection, or writing output to a display terminal or network.

The classic work stealing algorithm does not account for I/Os. I/O operations are typically done via low-level system libraries (e.g., the GNU C library) or through system calls provided by the Operating System (OS). While one can directly invoke functions provided by these libraries within a task parallel program, doing so has performance implications. In particular, when a ***worker thread*** — surrogate of a processing core managed by the scheduler — encounters an I/O operation, it can block for an extended period of time, leaving one of the physical cores underutilized.[1]

In this work, we design a scheduler that *hides* the I/O latency — when a worker encounters a blocking I/O, it suspends the current execution context and works elsewhere in the computation. When the I/O completes, some worker (not necessarily the worker that suspended it) picks up the suspended context and resumes it. Moreover, the programming model seamlessly integrates both blocking and nonblocking I/Os into the task parallel programming model. Finally, the scheduler provides provably good performance bounds and efficient implementation.

As far as we know, only one prior result provides a provably efficient scheduling bound of task parallel programs with I/Os. Muller and Acar [20] present a cost model for reasoning about latency incurring operations (such as I/Os) in task parallel programs. In their work, given a computation with ***work*** T_1 — the total computation time on one core — and ***span***[2] T_∞— the execution time of the computation on infinitely many cores — the scheduler executes the computation in expected time $O(T_1/P + T_\infty U(1 + \lg U))$, where the U is the maximum number of latency incurring operations that are logically in parallel. Their bound is ***latency-hiding*** in that, the latencies of I/O only appear in the span term and not the work term. If no latency-incurring operations are used, their bound is asymptotically equal to the standard work stealing bound of $O(T_1/P + T_\infty)$.

In this work, we improve the latency-hiding bound by using a scheduling algorithm based on ProWS [22], a recently developed work-stealing scheduler that efficiently supports futures. We implement I/O operations seamlessly within task parallel code using futures while getting nearly asymptotically optimal completion time. In particular, our latency-hiding scheduler provides an execution time bound of $O(T_1/P + T_\infty \lg P)$ in expectation; this bound is independent of the number of I/Os in the system. Compared to the standard work-stealing bound, it has an additional term of $\lg P$ on the span term. This implies that while the standard work-stealing

SPAA '19, June 22–24, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6184-2/19/06.
https://doi.org/10.1145/3323165.3323175

[1] Low-level system support for ***asynchronous*** (non-blocking) I/O exists, but resuming the context on getting the I/O completion (typically a signal) can be complex.
[2] The term span is sometimes called "critical-path length" and "computation depth" in the literature.

scheduler provides linear speedup when $T_1/T_\infty = \Omega(P)$, our scheduler provides linear speedup when $T_1/T_\infty = \Omega(P \lg P)$. ProWS has the same bound, but the analysis does not directly apply here due to the latency of the I/Os. We extend ProWS's bound to futures with I/Os.

The high-level intuition on why using futures to do I/Os and then using ProWS to schedule these futures provides better bounds is as follows: The work-stealing algorithm by Muller and Acar is ***parsimonious*** — a worker never steals unless it runs out of work to do. In contrast, ProWS's and our work-stealing algorithm is ***proactive*** — whenever a worker encounters a blocking I/O operation, it suspends the entire execution context and finds something else to do by work stealing. This behavior may seem counter-intuitive since it potentially increases the number of steal attempts. It turns out, however, that by carefully managing deques, one can amortize the steal cost against the work term sometimes, thereby obtaining a better bound.

2 IMPLEMENTATION AND EVALUATION

Our prototype system, Cilk-L, is based on Cilk-F [22], an extension of Intel Cilk Plus [14] that supports futures and implements ProWS. In Cilk-F, when a function F spawns off a function G by prefixing the call with `fut-create`, G may execute in parallel with the *continuation* of F. The completion of G is not tied to the lexical scope of F, however. Instead, the `fut-create` returns a ***handle*** h, with which the execution of G is associated. This handle h can later be used to ensure termination of G and retrieve the result of its computation. By invoking `get` on the handle, the control cannot pass beyond the `get` until the execution of G terminates and the future is marked as ready.

Cilk-L defines a special type of futures, called ***IO futures***, which utilize the parallelism abstraction provided futures to schedule I/Os in a latency-hiding manner which is composable with the rest of parallel constructs supported in Cilk-F (`spawn`, `sync`, `fut-create`, and `get`). When a worker invokes an I/O operation using an IO future, a handle is returned, and the I/O can be done either ***synchronously*** by calling `get` on the handle immediately, or ***asynchronously***, by calling `get` at a later time when the result is needed in order for the control to proceed. The I/O operations are managed using `epoll` [1] in separate per-worker I/O threads, with each I/O thread pinned to the same physical core as its associated worker but on a different hyperthread context.

Experimental setup. We ran our experiments on a machine with two Intel Xeon Gold 6148 processors, each with 20 2.40-GHz cores, with a total of 40 cores. Each core has a 32 KB L1 data cache, 32 KB L1 instruction cache, and a 1 MB L2 cache. Hyperthreading is enabled. Each sockets has a 27.5 MB shared L3 cache. The system has 768 GB of main memory. Cilk-L and `map-reduce` are compiled with LLVM/Clang 3.4.1 with `-O3 -flto` running on Linux kernel version 4.15. Each data point is the average of 10 runs. All data points have standard deviation less than 5% except for a couple data points at 6%.

Evaluation. We empirically evaluated Cilk-L with microbenchmarks that interleave compute-intensive kernels with operations that incur I/O latencies, as shown in Figure 1. Rather than opening a network connection at line 6, however, we use a timed file descriptor (`timerfd` [2]) which becomes ready for I/O when the timer expires. In this way we emulate delays for 5000 remote server connections. After the timer fires and the I/O is read, the microbenchmarks calculate the 35th Fibonacci number (function f on line 9). We replace the function g on line 16 with a simple addition operation.

In Figure 2 we compare the execution times of this benchmark implemented using Cilk-L to an "idealized' execution (*ideal*), where the I/O effectively does not incur any latency. In an ideal scenario, the I/O latency is completely hidden and all of the execution time of the microbenchmark would result from the calculation of the 35th Fibonacci number. The *ideal* data was collected using the Cilk-F runtime, which was shown by Singer et al. [22] to be provably and practically efficient at scheduling fork-join and futures parallelism. We ran the microbenchmark with emulated delays of 1 ms, 50 ms, 100 ms, and 500 ms. We collected this data for various processor counts, and calculated the overhead caused by the I/O latency for each run relative to the execution time of *ideal* on the same number of processors. As can be seen in Figure 2, the I/O latency overhead was not more than 1.14× in this experiment.

Our microbenchmarks are identical to the benchmarks used by Muller and Acar [20], with the exception that Muller and Acar instead calculate the 30th Fibonacci number. We did, however, collect data with our microbenchmarks calculating the 30th Fibonacci number, and found that Cilk-L achieves speedup greater than 1443× on 30 cores when the simulated latency is 500 ms. In contrast, Muller and Acar [20] achieve speedups around 80× using the same configuration with a 500 ms latency. Note, however, that this indirect comparison may not be apples-to-apples, since their implementation is based on parallel ML, which has additional inherent overhead in memory management for functional programming.

3 CONCLUSION

In order to support modern desktop and server software, I/O operations should be supported as a fundamental component of task parallel platforms. We extend the scheduling algorithm in Cilk-F to incorporate the latency-hiding cost model, and show that with I/O latency, the algorithm can schedule the computation in time $O(T_1/P + T_\infty \lg P)$ on P cores, independent of the number of I/O operations active in parallel. This bound is an improvement over the prior state-of-the-art by Muller and Acar. Since $\max\{T_1/P, T_\infty\}$ is a lower bound on the execution of this program on P processors, this bound is nearly asymptotically optimal except for the $\lg P$ overhead on the span.

We have developed Cilk-L by extending Cilk-F to support scheduling I/O in a latency-hiding way. By utilizing futures, one can perform asynchronous I/Os in task parallel code in a way that is composable with other parallel constructs, and this can be implemented efficiently.

REFERENCES

[1] 2019. Linux Programmer's Manual EPOLL(7). http://man7.org/linux/man-pages/man7/epoll.7.html. (2019). Accessed in January 2019.

[2] 2019. Linux Programmer's Manual TIMERFD_CREATE(2). http://man7.org/linux/man-pages/man2/timerfd_create.2.html. (2019). Accessed in January 2019.

[3] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. 2000. The Data Locality of Work Stealing. In *Proceedings of the 12th ACM Annual Symposium on Parallel Algorithms and Architectures*. 1–12.

[4] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. 2002. The Data Locality of Work Stealing. *Theory of Computing Systems* 35, 3 (2002).

```
1  Function distMapReduce(f, g, id, lo, hi)
2  |  n ← hi − lo;
3  |  if n = 0 then return id; //return identity.
4  |  else if n = 1 then
5  |  |  char buf[NBYTES] ; //buffer for input data.
6  |  |  fd ← openConnection(lo) //open network connection.
7  |  |  io_future fut ← cilk_read(fd, buf, NBYTES)
8  |  |  get(fut);
9  |  |  return f(buf);
10 |  end
11 |  else
12 |  |  mid ← (lo + hi)/2;
13 |  |  r1 ← = spawn distMapReduce(f, g, id, lo, mid);
14 |  |  r2 ← distMapReduce(f, g, id, mid, hi);
15 |  |  sync;
16 |  |  return g(r1, r2)
17 |  end
18 end
```

Figure 1: Pseudo code for our microbenchmark, distributed map and reduce.

latency	T_1	T_5	T_{10}	T_{15}	T_{20}	T_{25}	T_{30}	T_{35}	T_{40}
ideal	338.48 (1.00×)	67.70 (1.00×)	30.26 (1.00×)	19.73 (1.00×)	14.80 (1.00×)	11.95 (1.00×)	9.84 (1.00×)	8.73 (1.00×)	7.71 (1.00×)
1 ms	344.86 (1.02×)	69.02 (1.02×)	32.63 (1.08×)	20.87 (1.06×)	15.12 (1.02×)	12.35 (1.03×)	10.39 (1.06×)	9.00 (1.03×)	7.99 (1.04×)
50 ms	344.98 (1.02×)	68.83 (1.02×)	33.21 (1.10×)	22.39 (1.13×)	15.69 (1.06×)	12.95 (1.08×)	10.53 (1.07×)	9.06 (1.04×)	8.06 (1.05×)
100 ms	345.02 (1.02×)	66.38 (0.98×)	34.17 (1.13×)	21.47 (1.09×)	15.79 (1.07×)	12.64 (1.06×)	10.40 (1.06×)	9.38 (1.07×)	8.11 (1.05×)
500 ms	345.38 (1.02×)	69.61 (1.03×)	32.05 (1.06×)	21.19 (1.07×)	15.79 (1.07×)	12.69 (1.06×)	10.77 (1.09×)	9.60 (1.10×)	8.61 (1.12×)

Figure 2: The execution times, in seconds, of `map-reduce` using Cilk-L with different latencies (ms is milliseconds). The values in parentheses are overheads relative to the corresponding T_P time of *ideal*, which runs the baseline of `map-reduce` on Cilk-F with (effectively) zero latency and incurs no system overhead for latency hiding.

[5] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. 1998. Thread Scheduling for Multiprogrammed Multiprocessors. In *Proceedings of the 10th Annual ACM Symposium on Parallel Algorithms and Architectures*. 119–129.

[6] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. 2001. Thread Scheduling for Multiprogrammed Multiprocessors. *Theory of Computing Systems* (2001), 115–144.

[7] Rajkishore Barik, Zoran Budimlić, Vincent Cavè, Sanjay Chatterjee, Yi Guo, David Peixotto, Raghavan Raman, Jun Shirako, Sağnak Taşırlar, Yonghong Yan, Yisheng Zhao, and Vivek Sarkar. 2009. The Habanero Multicore Software Research Project. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*. ACM, Orlando, Florida, USA, 735–736.

[8] Robert D. Blumofe and Charles E. Leiserson. 1994. Scheduling Multithreaded Computations by Work Stealing. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*. 356–368.

[9] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling Multithreaded Computations by Work Stealing. *JACM* 46, 5 (1999), 720–748.

[10] Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. 2011. Habanero-Java: the new adventures of old X10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java (PPPJ '11)*. 51–61.

[11] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 2005. X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 519–538.

[12] John S. Danaher, I-Ting Angelina Lee, and Charles E. Leiserson. 2008. Programming with exceptions in JCilk. *Science of Computer Programming* 63, 2 (Dec. 2008), 147–171.

[13] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The Implementation of the Cilk-5 Multithreaded Language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*. ACM, 212–223.

[14] Intel 2013. Intel® Cilk™ Plus. https://www.cilkplus.org. (2013).

[15] Intel Corporation 2012. *Intel(R) Threading Building Blocks*. Intel Corporation. Available from http://software.intel.com/sites/products/documentation/doclib/tbb_sa/help/index.htm.

[16] Intel Corporation 2013. *Intel® Cilk™ Plus Language Extension Specification, Version 1.1*. Intel Corporation. Document 324396-002US. Available from http://cilkplus.org/sites/default/files/open_specifications/Intel_Cilk_plus_lang_spec_2.htm.

[17] Doug Lea. 2000. A Java fork/join framework. In *ACM 2000 Conference on Java Grande*. 36–43.

[18] I-Ting Angelina Lee, Silas Boyd-Wickizer, Zhiyi Huang, and Charles E. Leiserson. 2010. Using Memory Mapping to Support Cactus Stacks in Work-Stealing Runtime Systems. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*. ACM, 411–420.

[19] Charles E. Leiserson. 2010. The Cilk++ Concurrency Platform. *Journal of Supercomputing* 51, 3 (2010), 244–257.

[20] Stefan K. Muller and Umut A. Acar. 2016. Latency-Hiding Work Stealing: Scheduling Interacting Parallel Computations with Work Stealing. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, Pacific Grove, California, USA, 71–82.

[21] OpenMP 4.0 2013. *OpenMP Application Program Interface, Version 4.0*.

[22] Kyle Singer, Yifan Xu, and I-Ting Angelina Lee. 2019. Proactive Work Stealing for Futures. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. ACM, Washington, District of Columbia, 257–271.