SANDIA REPORT

SAND2013-5748
Unlimited Release
Printed July 2013

Design Issues in the Semantics and Scheduling of Asynchronous Tasks

Stephen L. Olivier

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2013-5748
Unlimited Release
Printed July 2013

Design Issues in the Semantics and Scheduling of Asynchronous Tasks

Stephen L. Olivier
Scalable System Software Department
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-1319
slolivi@sandia.gov

Abstract

The asynchronous task model serves as a useful vehicle for shared memory parallel programming, particularly on multicore and manycore processors. As adoption of model among programmers has increased, support has emerged for the integration of task parallel language constructs into mainstream programming languages, e.g., C and C++. This paper examines some of the design decisions in Cilk and OpenMP concerning semantics and scheduling of asynchronous tasks with the aim of informing the efforts of committees considering language integration, as well as developers of new task parallel languages and libraries.

Acknowledgment

The observations in this paper draw on discussions over several years with members of the OpenMP language committee, including Federico Massaioli, Alejandro Duran, and Brian Bliss.

Contents

1 Introduction 7
2 Syntax and Semantics of Cilk and OpenMP Tasks 8
2.1 Cilk and Cilk Plus 8
2.2 OpenMP Tasks 9
3 Rationale and Scheduling Implications 10
3.1 Work-first Scheduling in Cilk 10
3.2 Help-first Scheduling 10
3.3 Work-stealing for OpenMP Tasks 10
3.4 Victim Selection in Work Stealing 11
3.5 Parallel Depth-first Schedules 11
4 Conclusion 12
References 13

Figures

1 Calculating Fibonacci numbers in Cilk and OpenMP. 8

Tables

1 Introduction

Many of the concepts used in task parallel programming languages originated in early attempts at multithreaded programming using functional languages, e.g., MultiLisp [11]. Cilk borrowed from and expanded upon these concepts to design and to implement a seminal task parallel extension to the declarative C programming language [4, 7]. Its authors made several key choices in favor of language simplicity, compiler support, and run time efficiency that have been widely imitated in more recent task parallel languages such as OpenMP 3.0. In other key decisions, the languages have taken divergent paths. This paper explores those decisions and their implications, as an understanding of these issues may guide the development of future task parallel frameworks, including those in preparation for inclusion in the C and C++ language standards.

Section 2 presents the basic syntax and semantics of the task model as specified in Cilk and OpenMP, while Section 3 examines the rationale behind the decisions and their implications for scheduling. Section 4 summarizes the issues and offers some further observations. Note that the objective of this paper is not to condemn or sanction particular design decisions, but to clarify and examine them in context.

2 Syntax and Semantics of Cilk and OpenMP Tasks

2.1 Cilk and Cilk Plus

A Cilk program is a C program with three additional keywords: `cilk`, `spawn`, and `sync`. The `cilk` keyword indicates the declaration of a Cilk procedure, i.e., a function that may be executed in parallel. Parallel invocations to Cilk procedures are made using the `spawn` statement. When a new (child) task is generated, the currently executing (parent) task is suspended and the child task begins execution. The parent procedure may be executed concurrently on another thread, or following the execution of its descendants on the same thread. The `sync` statement ensures the completion of any outstanding child Cilk procedures spawned by the current procedure up to that point.

Intel® Cilk™ Plus changes the keywords `spawn` and `sync` to `cilk_spawn` and `cilk_sync`[1]. It also adds and implicit `cilk_sync` at the end of every function that contains a `cilk_spawn`. Cilk Plus provides a family of templated classes called *hyperobjects* to share and to update concurrent data objects safely. Hyperobjects are global objects with member functions and overloaded operators to present a well-defined interface to the implicitly synchronized data [8]. For example, a variable `count` that may be modified by different spawned procedures, the programmer simply declares it as an object of class `cilk::reducer_opadd<int>` and update it with a simple statement like `count++`.

```
cilk int fib(int n) {
  if (n < 2)
    return n;
  else {
    int x, y;

      x = spawn fib(n-1);

      y = spawn fib(n-2);
    sync;
    return (x+y);
  }
}
```

```
int fib(int n) {
  if (n < 2)
    return n;
  else {
    int x, y;
    #pragma omp task
      x = fib(n-1);
    #pragma omp task
      y = fib(n-2);
    #pragma omp taskwait
    return (x+y);
  }
}
```

Figure 1. Calculating Fibonacci numbers in Cilk and OpenMP.

[1] Intel and Cilk are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

2.2 OpenMP Tasks

The OpenMP® API's task parallel model is expressed through a compiler-supported language extension[2], the design of which has been helpfully documented [1]. Version 3.0 of the OpenMP specification for Fortran and C/C++ shared memory parallel programming introduced explicit task parallelism to complement its existing data parallel constructs [14]. The `task` and `taskwait` directives resemble Cilk `spawn` and `sync` statements respectively, as shown in Figure 1. However, the OpenMP `task` directive generates a task from a statement or structured block, not a procedure. In addition to the `taskwait` synchronization, the OpenMP `barrier` construct also provides task synchronization. Threads encountering a barrier must complete all outstanding tasks generated by threads in that team before they may pass the barrier. Data clauses specify whether tasks share variables from surrounding scopes or whether they make private or initialized private copies of the variables.

An unexecuted OpenMP task may be scheduled onto any thread. OpenMP defines two classes of tasks based on restrictions of task suspension and rescheduling: *tied* and *untied* tasks. As specified in OpenMP 3.0, they differ in two important ways. First, a tied task may only be suspended at specific *task scheduling points*, e.g., generation of new tasks, `taskwait` synchronizations, and barriers. An untied task may be suspended at any point during execution. Second, a tied task can be scheduled initially onto any thread but is not allowed to migrate between different threads during execution. An untied task may migrate between different threads during execution. Care must be taken if *threadprivate* variables or thread numbers (obtained through `omp_get_thread_num()`) are used in untied tasks. A threadprivate variable is a global variable of which each thread has a copy. Thus, an untied task may access different copies of threadprivate variables at different points in the execution of the task.

[2]OpenMP is a registered trademark of the OpenMP Architecture Review Board.

3 Rationale and Scheduling Implications

3.1 Work-first Scheduling in Cilk

The implementation of the Cilk run time system is based on a distributed scheduler. Each thread has its own queue of spawned procedure frames. Work stealing enables concurrent execution: idle threads steal procedure frames from the queues of busy threads. Recall that upon each new procedure spawn, Cilk suspends execution of the parent procedure, places its frame on the local LIFO queue, and begins execution of the child procedure. This approach is called *work-first* scheduling. By always scheduling the child procedure immediately, Cilk maintains sequential execution order of all work since the last steal. Several desirable properties follow: If a sequential execution of the original program has good temporal locality, so does a parallel execution. (The design of the queue – local access in LIFO order and stealing in FIFO order – is also crucial to maintaining locality.) A single-threaded execution of a Cilk program follows the same execution order as its sequential equivalent. If hyperobjects are used (in Cilk Plus), local views of the object in a parallel execution are maintained based on the work-first assumption, and they are combined in an order matching a single-threaded execution. For efficiency, book keeping for synchronization can be avoided in a procedure until it is stolen. Finally, time, space, and communication bounds have been proven based on the work-first approach.

3.2 Help-first Scheduling

All suspended spawned procedures in Cilk are partially executed. Thus, to enable concurrent execution, a work-first scheduler must allow the migration of partially-executed procedures. Consider instead a scheduling discipline in which the newly spawned child procedure is placed on the queue and the parent procedure continues execution. This approach is called *help-first scheduling*. In a help-first scheduler, queues would contain spawned procedures that have not yet begun execution. Such a scheduler could achieve concurrent execution without supporting the migration of partially-executed tasks. Furthermore, help-first scheduling can allow work to be distributed more quickly when new procedures are spawned in quick succession from a common generating parent and placed on the queue immediately for stealing by other threads. The earlier generation of sibling procedures exposes available parallelism earlier in the execution, which can be helpful for task graphs that are wide and shallow [9]. Unfortunately, sequential ordering is not maintained, and proven space guarantees are not established for help-first scheduling. Schedulers that switch between work-first and help-first schedulers based on available parallelism and space constraints have been demonstrated [10].

3.3 Work-stealing for OpenMP Tasks

Since tied tasks in OpenMP are prohibited from migrating between threads once they have been partially executed, a work-first scheduler would serialize the execution of an OpenMP program

using only tied tasks. No parent task would be eligible for stealing. However, a help-first scheduler could execute a program consisting of tied tasks in parallel, since executed child tasks on the threads' queues could be stolen. An OpenMP program consisting of untied tasks could be scheduled for parallel execution using either work-first or help-first scheduling [6].

Note that an OpenMP implementation can choose to treat untied tasks the the same as tied tasks, i.e., suspending tasks only at task scheduling points and not migrating suspended tasks. In practice many do, using a help-first or even a breadth-first scheduler. OpenMP 3.1 [15] introduces the `taskyield` construct to allow the user to specify a custom task scheduling point, informing the run time that task suspension at that point in the code may be useful and enabling such a suspension. OpenMP 4.0 [16] allows the suspension of untied tasks only at task scheduling points rather than at any point in the execution. This restriction makes reasoning about correctness easier. As of OpenMP 4.0, the only remaining difference between tied and untied tasks is that untied tasks are allowed to migrate.

3.4 Victim Selection in Work Stealing

Another requirement of Cilk is that the selection of the victim to steal from must be random. The work, space, and communication bounds for Cilk are based on this assumption. With processor topologies evolving in design, complexity, and heterogeneity, it is not clear that randomized stealing remains the best choice in practice. For example, it could be better for a thread to steal from its neighbor running on a different SMT thread on the same core rather than from a thread on another core. On a multi-socket system, it may better to steal from another thread running within the same chip rather than a thread on another chip. Locality-based scheduling remains an area ripe for research and can have profound impacts on performance [12].

3.5 Parallel Depth-first Schedules

Approaches outside the realm of work stealing are also worthy of consideration for task scheduling. The *parallel depth-first (PDF) schedule* is an approach designed to coordinate the threads to execute as close as possible to a single path in the computation, resulting in a lower memory requirement. The particular single path to be followed is the sequential order. Because sequential execution is depth-first, concurrent execution according to sequential order can be approximated using a LIFO queue shared among all threads. The resulting space bound is better that that of the work-first work stealing scheduler [3]. While work stealing is well suited to private caches, PDF scheduling is well suited to shared caches [2, 5]. On the other hand, PDF scheduling can cause sharing or false sharing of data in cache lines in a system using private caches, which causes the cache lines to bounce back and forth between cores. A hybrid approach combining work stealing and PDF scheduling can be used to schedule tasks hierarchically on systems with complex memory architectures such as NUMA systems and combinations of shared and private caches [13].

4 Conclusion

Semantics and scheduling are intimately intertwined in task parallelism. Restrictions on task scheduling, whether arising from semantic requirements or theoretical models, should be carefully considered before they are adopted in new task parallel programming frameworks, such as those under discussion for future C and C++ standards. Particular attention should be paid to the handling of thread-local variables and views of global variables. In OpenMP, the use of threadprivate variables is a primary motivation for tied tasks, which points away from the use of work-first scheduling. On the other hand, the use of reducers in Cilk Plus points toward work-first scheduling.

Work-first, help-first, PDF, locality-aware, and hybrid scheduling all have their merits and exhibit significant trade-offs. Allowing flexibility in scheduling would allow further innovation in this space for languages and libraries adopting the task parallel model.

References

[1] Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. The design of OpenMP tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20:404–418, March 2009.

[2] Guy E. Blelloch and Phillip B. Gibbons. Effectively sharing a cache among threads. In *SPAA '04: Proc. 16th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 235–244. ACM, 2004.

[3] Guy E. Blelloch, Phillip B. Gibbons, and Yossi Matias. Provably efficient scheduling for languages with fine-grained parallelism. *Journal of the ACM*, 46(2):281–321, 1999.

[4] 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.

[5] Shimin Chen, Phillip B. Gibbons, Michael Kozuch, Vasileios Liaskovitis, Anastassia Ailamaki, Guy E. Blelloch, Babak Falsafi, Limor Fix, Nikos Hardavellas, Todd C. Mowry, and Chris Wilkerson. Scheduling threads for constructive cache sharing on CMPs. In *SPAA '07: Proc. 19th ACM Symposium on Parallel Algorithms and Architectures*, pages 105–115. ACM, 2007.

[6] Alejandro Duran, Julita Corbalán, and Eduard Ayguadé. Evaluation of OpenMP task scheduling strategies. In Rudolf Eigenmann and Bronis R. de Supinski, editors, *IWOMP '08: Proc. Intl. Workshop on OpenMP*, volume 5004 of *Lecture Notes in Computer Science*, pages 100–110. Springer, 2008.

[7] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI '98: Proc. 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 212–223. ACM, 1998.

[8] Mateo Frigo, Pablo Halpern, Charles E. Leiserson, and Stephen Lewin-Berlin. Reducers and other Cilk++ hyperobjects. In *SPAA '09: Proc. 21st ACM Symposium on Parallelism in Algorithms and Architectures*, pages 79–90. ACM, August 2009.

[9] Yi Guo, Rajkishore Barik, Raghavan Raman, and Vivek Sarkar. Work-first and help-first scheduling policies for async-finish task parallelism. In *IPDPS '09: Proc. 2009 IEEE Intl. Symposium on Parallel and Distributed Processing*, pages 1–12. IEEE, 2009.

[10] Yi Guo, Jisheng Zhao, Vincent Cave, and Vivek Sarkar. SLAW: a scalable locality-aware adaptive work-stealing scheduler for multi-core systems. In *PPoPP '10: Proc. 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 341–342. ACM, 2010.

[11] Robert H. Halstead, Jr. MULTILISP: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.

[12] Stephen L. Olivier, Bronis R. de Supinski, Martin Schulz, and Jan F. Prins. Characterizing and mitigating work time inflation in task parallel programs. In *SC 12: Proc. Intl. Conference on High Performance Computing, Networking, Storage and Analysis*, pages 65:1–65:12. IEEE Computer Society Press, 2012.

[13] Stephen L Olivier, Allan K Porterfield, Kyle B Wheeler, Michael Spiegel, and Jan F Prins. OpenMP task scheduling strategies for multicore NUMA systems. *Intl. Journal of High Performance Computing Applications*, 26(2):110–124, May 2012.

[14] OpenMP Architecture Review Board. OpenMP API, Version 3.0, May 2008.

[15] OpenMP Architecture Review Board. OpenMP API, Version 3.1, July 2011.

[16] OpenMP Architecture Review Board. OpenMP API, Version 4.0 (Release Candidate 2), March 2013.

DISTRIBUTION:

1 MS 0899 Technical Library, 9536 (electronic copy)

Sandia National Laboratories