Two Examples of Parallel Programming without Concurrency Constructs (PP-CC)

Chen Ding
University of Rochester
cding@cs.rochester.edu

Categories and Subject Descriptors D.1.3 [*Concurrent programming*]: Parallel programming

General Terms Languages, Performance

1. Introduction

Speculative parallelization divides a program into possibly parallel tasks. Parallel execution succeeds if it produces the same output as the original program; otherwise, it is reverted to sequential execution. Previous studies have developed various programming primitives [1, 3, 6, 8–11, 13]. Many of these new primitives are *hints* and do not change program output even if they are wrong.

Consider the two hints below. In each case, a hint annotates a code block and suggests the presence or absence of parallelism in the execution of the code block. They are developed as part of the *BOP* system and hence bear a common prefix [3].

- A *possibly parallel region* bop_ppr suggests that run-time instances of the code block are parallel. A *PPR* block is similar to a safe future [11] or an ordered transaction [10].
- A *likely serial region* bop_ordered suggests that run-time instances of the code block should be executed sequentially in program order. The meaning is similar to the ordered directive in OpenMP except that bop_ordered is a hint (implemented using speculative post-wait [4], which is an extension of Cytron's *do-across* construct [2]).

The two hints do not change the program output. They are more appropriately called a *parallelization interface* rather than a parallel programming interface. While a parallel language controls both parallelism and its semantics, a parallelization interface affects only parallelism but not the program result.

A parallelization interface cannot use concurrency constructs. In this paper, a *concurrency construct* is defined as a synchronization primitive that permits out-of-order access to shared data. Common examples are locks, barriers, critical and atomic sections, and transactions. Concurrency constructs allow a program, when given the same input, to produce one of several possible results that are all correct. Parallelization hints are limited to producing a single outcome—the sequential result—and have to enforce in-order updates of shared data in general (except when out-of-order access does not change the result).

Copyright is held by the author/owner(s).
PPoPP'11, February 12–16, 2011, San Antonio, Texas, USA.
ACM 978-1-4503-0119-0/11/02.

```
while q.has_work
 w = q.get_work

 t = w.do_lots_work

 result_tree.add(t)


end while
```

```
while q.has_work
 w = q.get_work
 bop_ppr {
  t = w.do_lots_work
  bop_ordered {
   result_tree.add(t)
  }
 }
end while
```

Figure 1. A work loop

The benefit of hints is safe parallel programming. Many issues affect safety. A program may call third-party code that is not thread safe. It may not be completely parallel, and the degree of parallelism may change from input to input. In addition, the granularity may be unknown or input dependent. Some tasks may be so short that sequential execution is fastest. Traditionally, a user should resolve all these issues in order to parallelize a program. With hints, the parallelism can always be suggested. Furthermore, no parallel debugging is needed. The parallel execution is correct if the sequential one is.

Concurrency constructs have been a mainstay of parallel programming, a question naturally arises as to how expressive and usable a parallelization interface can be without them. It is difficult to quantify usability. This short paper tries to answer the question through two examples of parallel programming without concurrency constructs.

2. A While Loop

Information processing is often implemented by a while loop, as shown in Figure 1. The loop body has three steps: dequeue the next work, process it, and insert the output into result_tree. The parallelized version encloses the last two steps in a *PPR* block, suggesting that the work can be done in parallel once it is dequeued.

Consider the step of tree insertion. On the one hand, it is part of the *PPR* block so it obtains the result of parallel processing. On the other hand, the tree is shared, so the insertion is serialized by the bop_ordered hint. The resulting tree is identical to one from sequential execution.

As a comparison, consider parallelizing the work loop using OpenMP. The step of tree insertion is marked as a critical section. As a concurrency construct, the critical section allows tree insertions to happen in any order—a later iteration may insert first if it finishes earlier. Out-of-order insertion improves parallelism—a later task does not have to wait—but it leads to non-determinism—

the shape of the tree may differ from run to run. Non-determinism complicates debugging. Suppose the implementation has an infrequent error that manifests once every hundred executions. A user will have difficulty reproducing it in a debugger.

An ordered block requires in-order tree insertion and loses parallelism. However, the loss may be compensated by starting future *PPR* tasks when processors are idle [14]. As a hint, a *PPR* block does not have to finish before starting the subsequent *PPR* blocks (if they exist). In comparison, it is unsafe for OpenMP to overlap the execution of consecutive loops.

Next we show that hints may increase the amount of parallelism without complex programming.

3. Time Skewing

Iterative solvers are widely used to compute fixed-point or equilibrium solutions. Figure 2 shows the structure of a typical solver as a two-nested loop. The outer level is the time-step loop. In each time step, the inner loop computes on some domain data. The inner loop is usually data parallel. The time steps, however, have three types of dependences: the convergence check, which depends on the entire time step; the continuation check, which depends on the convergence result of the last step; and the cross time-step data conflict, since time steps use and modify the same domain data.

In manual parallelization, a barrier is inserted between successive time steps to preserve all three dependences. Although simple to implement, the solution precludes parallelism between time steps. Previous literature shows that by overlapping time steps, one may obtain large performance improvements for both sequential [7, 12] and parallel [5] executions. Wonnacott termed the transformation *time skewing* [12].

Parallelization hints can express time skewing with four annotations shown in Figure 2. It uses two parallel blocks. The first block allows each time step to be parallelized. The second allows consecutive time steps to overlap. In addition, it uses two ordered blocks. The first serialize data updates in the inner loop. The second delays the convergence check after all data updates are completed.

```
while ( not converged )

  for i is 1 to N

      r = compute( data[ i ] )

      s = s.add_result( r )

  end for

  if good_enough?( s )
C:   converged = true
  end if
end while
```

```
while ( not converged )
  for i is 1 to N
    bop_ppr {
      r = compute( data[ i ] )
      bop_ordered {
        s = s.add_result( r )
      }
    }
  end for

  bop_ppr {
    bop_ordered {
      if good_enough?( s )
C:      converged = true
      end if
    }
  }
end while
```

Figure 2. Time skewing—successive time steps may overlap

The hints ignore the continuation dependence and cross time-step conflicts since they happen infrequently. The continuation check fails only at the last iteration. Cross-iteration conflicts usually do not happen immediately if the inner loop processes domain data in a fixed order. The *BOP* speculation support will detect and correct these errors when they happen. In other times, the parallelism between time steps is profitably exploited. We have tested a case in which the *BOP* version was 18% faster than the OpenMP version on 7 processors, thanks to time skewing. In this case, parallelization hints outperformed critical section and barrier synchronization while still maintaining the sequential result.

4. Summary

The two examples show the limitation and benefits of parallel programming without concurrency constructs:

- *No concurrency constructs.* No out-of-order updates as permitted by locks, barriers, atomic sections, and transactions.
- *Speculation support.* The cost limits the efficiency and scalability.
- *Ease of parallel programming.* Hints relieve a user from correctness considerations and may be inserted by hand or automatically.
- *Speculative parallelism.* A user may parallel code that is often but not always parallel.

The first two mean less parallelism and more overhead. The next two, however, mean easier, safe parallelization. Ease of programming may mean faster speed, as shown by the second example.

References

[1] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: Safe multithreaded programming for C/C++. In *Proceedings of OOPSLA*, 2009.

[2] R. Cytron. Doacross: Beyond vectorization for multiprocessors. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 836–844, St. Charles, IL, Aug. 1986.

[3] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang. Software behavior oriented parallelization. In *Proceedings of PLDI*, pages 223–234, 2007.

[4] B. Jacobs, T. Bai, and C. Ding. Distributive program parallelization using a suggestion language. Technical Report URCS #952, Department of Computer Science, University of Rochester, 2009.

[5] L. Liu and Z. Li. Improving parallelism and locality with asynchronous algorithms. In *Proceedings of PPOPP*, pages 213–222, 2010.

[6] A. Raman, H. Kim, T. R. Mason, T. B. Jablin, and D. I. August. Speculative parallelization using software multi-threaded transactions. In *Proceedings of ASPLOS*, pages 65–76, 2010.

[7] Y. Song and Z. Li. New tiling techniques to improve cache temporal locality. In *Proceedings of PLDI*, pages 215–228, Atlanta, Georgia, May 1999.

[8] C. Tian, M. Feng, and R. Gupta. Supporting speculative parallelization in the presence of dynamic data structures. In *Proceedings of PLDI*, pages 62–73, 2010.

[9] C. Tian, M. Feng, V. Nagarajan, and R. Gupta. Copy or Discard execution model for speculative parallelization on multicores. In *Proceedings of ACM/IEEE MICRO*, pages 330–341, 2008.

[10] C. von Praun, L. Ceze, and C. Cascaval. Implicit parallelism with ordered transactions. In *Proceedings of PPOPP*, Mar. 2007.

[11] A. Welc, S. Jagannathan, and A. L. Hosking. Safe futures for Java. In *Proceedings of OOPSLA*, pages 439–453, 2005.

[12] D. Wonnacott. Achieving scalable locality with time skewing. *International Journal of Parallel Programming*, 30(3), June 2002.

[13] A. Zhai, J. G. Steffan, C. B. Colohan, and T. C. Mowry. Compiler and hardware support for reducing the synchronization of speculative threads. *ACM Transactions on Architecture and Code Optimization*, 5(1):1–33, 2008.

[14] C. Zhang, C. Ding, X. Gu, K. Kelsey, T. Bai, and X. Feng. Continuous speculative program parallelization in software. In *Proceedings of PPOPP*, pages 335–336, 2010.