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

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

1. Introduction

Parallelization is the process of converting a sequential program into a parallel form. It is recognized that a general solution needs to incorporate user knowledge and hence there is a need for a programming interface for parallelization.

Speculative parallelization divides a program into possibly parallel tasks and runs them in parallel if they produce the same output as the original program. A run-time system monitors the execution for conflicts and reverts to sequential execution if needed. Previous studies have developed various programming primitives [1, 3, 6, 8, 9, 11]. Unlike parallel constructs, many of these new primitives are hints and can be wrong. We call them *parallelization hints*. They mark parallelism but do not change the output of a program.

This short paper discusses the constraints and opportunities of parallelization hints. The main points are summaried below. The first two mean lower performance, while the next two may mean better programmability and more parallelism, compared to programming not using hints.

- *No concurrency constructs.* A parallelization hint cannot allow out-of-order updates to shared data (unless they produce the same result). This rules out the use of concurrency constructs such as locks, barriers, critical sections, and transactions.
- *Needing speculation support.* Speculative execution is needed to protect against incorrect hints. The cost of speculation limits the efficiency and scalability of parallel execution.
- *Ease of parallel programming.* Parallelization is done by adding hints into a sequential program. The hints, however incorrect or incomplete, never cause the program to produce incorrect results. The hints may be inserted by hand or automatically.
- *Allowing speculative parallelism.* A user may use hints to express parallelism that exists only in some but not all executions of a program.

These points, especially the last three, are well known in cases when hints are used to mark parallelism. In this paper we focus on the implications on synchronization especially the comparison between parallelization hints and conventional concurrency constructs. We make the comparison concrete using two examples, which we will parallelize using the following two hints:

- *The parallel code block* bop_parallel*.* The hint specifies a possibly parallel region (*PPR*) [3]. The *PPR* code can be run in parallel with the code after the *PPR*. It is similar to a safe future [9] or an ordered transaction [8].
- *The serial code block* bop_serial*.* The executions of a serial code block should not overlap and should happen in the same order as in the sequential program. It is similar in intent to the ordered directive in OpenMP. The scope of ordered is a parallel construct, while bop_serial can appear anywhere in a program. bop_serial is implemented by speculative post-wait [4], which is an extension of Cytron's *do-across* construct [2].

```
while ( has_work( ) )
  w = get_work( )

  t = do_work( w )

  tree.insert( t )

end while
```

```
while ( has_work( ) )
  w = get_work( )
  bop_parallel {
    t = do_work( w )
    bop_serial {
      tree.insert( t )
    }
  }
end while
```

Figure 1. The work-loop example. The calls to `has_work` and `get_work` are run sequentially. The calls to `do_work` inside the parallel block are executed speculatively in parallel. The speculative tasks perform `tree.insert` one at a time in their loop order.

2. A Work Loop

A generic work-processing loop, as shown in Figure 1, tests for more work, gets the next work item, processes the work, and inserts the result into a tree. The parallelized version is shown in the same figure. The parallel block suggests that the processing of different work items is parallel. We refer to each instance of the parallel block a possibly parallel task or a *PPR* task.

The tree-insertion step needs synchronization. With manual parallelization, one can place `tree.insert` into a critical section to ensure exclusive access to the shared tree. To improve parallelism, a critical section allows tree insertions to happen in any order—a later loop iteration may insert its result first if it finishes earlier than others. However, this flexibility leads to non-determinism and complicates parallel programming. The shape of the tree depends on the relative finishing times of loop iterations, which depend on resource allocation in hardware and scheduling by the operating system. If the tree implementation has an infrequent error that manifests once every hundred executions, debugging will be difficult since it requires reproducing the error in a debugger.

Using hints, one can put `tree.insert` in a serial block. A serial block ensures that every loop iteration waits for all previous insertions to finish their tree insertion before it can start its tree insertion. A serial block is more restrictive than the critical section and may lose parallelism as a result. Consider the scenario in which the first iteration takes much longer than other iterations, and later iterations have to wait for a significant amount of time to complete their last step. The loss of parallelism may be compensated by more aggressive speculation. For example, the *BOP* system tries to start a new *PPR* task whenever there is an idle processor. It has been shown effective in tolerating uneven task sizes and non-uniform processor speeds due to hyperthreading [12].

The main benefit of the serial block is safety. A user or a programming tool can insert hints without having to ensure that they are always correct [3]. The speculation system guarantees that the resulting tree is always the same as in the sequential execution. No parallel debugging is needed. There are other benefits. A user marks the point of fork (in fork-join parallelism) without having to specify the point of join. The user does not have to understand all program code to insert hints. In the worst case, speculative execution is almost as fast as the sequential execution.

The safety guarantee comes at a cost in not just efficiency but also expressiveness. Since automatic correctness checking is necessary, a hint cannot express any type of parallelization that may produce an output different from sequential execution. Therefore, concurrency constructs cannot be hints, and the options for synchronization are limited when using hints. In light of this restriction, we next put parallelization hints to test in a more complex example.

3. Time Skewing

Iterative algorithms are widely used in numerical analysis and other domains to compute fixed-point or equilibrium solutions. Figure 2 shows the typical structure of an iterative solver. The outer level is a convergence loop. In each iteration is an inner loop that computes on some domain data. We refer to an outer loop iteration as a *time step*. For this example, we are concerned with three dependences: the convergence check, which depends on the results from the entire time step; the continuation check, which determines whether to terminate or to start the next time step; and the cross time-step data conflict, since different 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 integer factor performance improvements from better locality [7, 10]. Similar benefits are recently shown for parallel executions [5]. Wonnacott called the transformation *time skewing* [10].

Time skewing can improve parallel efficiency because it removes the periodic barrier synchronization. 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. It uses two serial blocks. The first ensures serial data updates, and the second ensures that dependence check happen after data updates are completed. This satisfies the first dependence.

The second dependence, the continuation check, is a problem only at the last iteration. The third, cross-iteration conflicts, happens between time steps as a whole, but it does not prohibit their partial overlap if the active *PPR*s are computing on different parts of the domain data. The speculation support will detect these two dependences and roll-back to sequential execution when needed. For the majority length of an execution, the parallelism between time steps can be profitably exploited. We have tested this transformation on a clustering algorithm using the *BOP* system [4]. Each time step has 15 tasks. When using 7 processors, time skewing was 18% faster than an OpenMP implementation that used a critical section and a barrier. The example shows the distinct benefits of parallelization hints, despite their inherent limitations and overheads.

```
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_parallel {
      r = compute( data[ i ] )

      bop_serial {
        s = s.add_result( r )
      }
    }
  end for

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

Figure 2. The time skewing example. The first parallel block allows each time step to be parallelized. The second allows consecutive time steps to overlap. The two serial blocks ensure that the convergence check happens after all data updates are completed.

References

[1] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: Safe multithreaded programming for C/C++. In *Proceedings of the International Conference on Object Oriented Programming, Systems, Languages and Applications*, 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 the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 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 the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 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 the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 65–76, 2010.

[7] Y. Song and Z. Li. New tiling techniques to improve cache temporal locality. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 215–228, Atlanta, Georgia, May 1999.

[8] C. von Praun, L. Ceze, and C. Cascaval. Implicit parallelism with ordered transactions. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Mar. 2007.

[9] A. Welc, S. Jagannathan, and A. L. Hosking. Safe futures for Java. In *Proceedings of the International Conference on Object Oriented Programming, Systems, Languages and Applications*, pages 439–453, 2005.

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

[11] 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.

[12] C. Zhang, C. Ding, X. Gu, K. Kelsey, T. Bai, and X. Feng. Continuous speculative program parallelization in software. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 335–336, 2010.