Synchronization and Pipelining on Multicore: Shaping Parallelism for a New Generation of Processors

Anna Youssefi

December 2006

1 Introduction

The potential for higher performance from increasing on-chip transistor densities, on the one hand, and the limitations in instruction-level parallelism of sequential applications and in the scalability of increasingly complicated superscalar and multithreaded architectures, on the other, are leading the microprocessor industry to embrace chip multi-processors as a cost-effective solution for the general-purpose computing market.

Multicore processors allow manufacturers to integrate larger numbers of simpler processing cores onto the same chip, thereby shortening design time and costs. They provide higher throughput for multiprogrammed workloads by enabling simultaneous processing of independent jobs, and can improve the performance of parallel applications by exploiting thread-level parallelism. Additionally, the individual cores used might be superscalar or multithreaded, thereby exploiting more finely-grained levels of parallelism as well.

While many design alternatives exist for multicore processors, one common choice is sharing the lower levels of the on-chip memory hierarchy among multiple processing cores. Although larger, shared caches cause higher access latencies and more complex logic, they provide a larger aggregate pool and reduce duplicate cache lines, thereby generally reducing capacity misses. However, sharing the cache can also negatively impact performance when the cache use behaviors of concurrent processes interfere with each other. Thus a good balance can be achieved from combining small, private first-level caches for fast, contention-free access with large, shared lower-level on-chip caches for flexible workload tolerance.

Performance on multicore processors is impacted by many of the same factors that impact performance on other shared-memory parallel architectures. However, the tighter coupling of on-chip resources changes some of the cost ratios that influence the design of parallel algorithms. Shared-cache multicore architectures introduce the potential for cheap inter-core communication, synchronization, and data sharing. They also introduce greater potential for cache contention.

One alternative to the data-parallel programming model is pipelining a computation across multiple processors, effectively treating the processors as high-level vector units.

Allen and Kennedy discuss pipelined parallelism in the context of the DOACROSS loop [7]. Vadlamani and Jenks refer to this method as the *Synchronized Pipelined Parallelism Model* [12]. In this paper, we examine the opportunities a shared-cache multicore processor presents for pipelined parallelism.

Using the dual-core shared-cache Intel Core Duo architecture as our experimental setting, we first analyze inter-core synchronization costs using a simple synchronization microbenchmark. Then we evaluate a pipelined parallel version of *Recursive Prismatic Time Skewing* [6] on a 2D Gauss-Seidel kernel benchmark. RPTS is an optimization technique for iterative stencil computations that increases temporal locality by skewing spatial domains across a time domain and blocking in both domains.

In the next subsection, we introduce our experimental setting. In Section 2, we discuss background issues including factors impacting performance in shared-cache architectures, the effects of cache-sharing contention on application performance, and the synchronized pipelined parallelism approach to programming for shared-cache environments. In Section 3, we present a simple synchronization microbenchmark and analyze its performance on the Intel Core Duo. In Section 4, we discuss optimization techniques for iterative stencil computations, then analyze their parallelization for a shared-cache multicore context. Finally, we present experimental results from a pipelined parallel implementation of RPTS in Section 5.

1.1 Experimental Setting

Our experimental setting is a MacBook Pro with an Intel Core Duo processor. The Core Duo features two 2 GHz processing cores which share a 2 MB L2 cache. Each core has private, separate 32 KB L1 data and instruction caches. The L1 and L2 caches are all 8-way set associative with a line size of 64 bytes. The cache hierarchy is inclusive, and a write back policy is used in conjunction with cache line ownership status for maintaining cache coherence [1].

In order to get hardware performance counter data using the Intel VTune Performance Analyzer, we run the Windows XP operating system on the Mac. In addition, Windows allows us to set the thread affinity for different threads to ensure that each thread runs on a different core. Otherwise, the scheduler time-slices threads across both cores along with other system processes in a round-robin fashion.

2 Background

2.1 CMP Cache Design and Performance

The dominant tradeoff between shared and private caches is hit rate versus hit latency. Shared caches increase hit latency for several reasons. A shared cache must be larger than a private cache, and wire delays increase as the size of a cache grows. Shared caches

must accommodate access by multiple cores, leading to more complicated access logic and possible bank contention during accesses. Additionally, if a lower level cache is shared but higher level caches are private, the overhead for maintaining cache coherence among the private caches will increase the hit latency for the shared cache.

Hit rates, on the other hand, will generally increase with shared caches, assuming the size of the shared cache is larger than the alternate private caches would be. A larger, shared cache is more flexible in accommodating varying application working set sizes for independent applications running together or alone. In the event of data sharing, it reduces the number of redundant on-chip cache lines and allows faster access to shared data.

Because of increasing access latencies and contention, higher numbers of cores sharing a cache can significantly harm performance. Huh et al. show that while there is no optimal number of processors sharing cache for all workloads, a sharing *degree* of 4 achieves a good compromise across the workloads they examine [5]. Nonetheless, workload-specific cache contention can also degrade performance; we discuss cache-sharing contention in the following subsection.

Inclusivity and write policy are two other design elements of the on-chip memory hierarchy which impact hit rate and hit latency. Inclusive caches duplicate on-chip cache lines, reducing the aggregate data store on chip. On the other hand, they simplify the lookup and coherence mechanisms, and allow manufacturers to reuse cache designs already common in uniprocessors. Write-through policies maintain coherence but degrade performance; instead, write-back policies are used in conjunction with maintaining cache line ownership state, such as in the MESI protocol.

Due to the complexity of factors impacting the performance of different multicore cache designs even in the absence of workload-specific cache behavior, conclusions drawn from simulations comparing the performance of shared and private caches are vulnerable to even slight miscalculations of predicted latencies. A study by Nayfeh, Hammond and Olukotun compares the effects of shared L1 cache and shared L2 cache with shared memory for several applications [9]. Initially the authors simulate shared L1 latency as a single cycle and exclude L1 bank contention because their simulation environment does not support latency hiding mechanisms. On this simulation, shared L1 cache always results in the best performance. However, in a more thorough simulation in which shared L1 latency is three cycles and L1 bank contention is included, shared L1 cache never results in the best performance, despite the effects of latency hiding techniques.

2.2 Cache-Sharing Contention

To share or not to share, that is the *first* question. It is quickly followed up by, at which level to share, and then, how to share effectively.

Threads that share cache may either share an address space or belong to different processes from separate address spaces. In both cases, cache sharing may lead to contention that can degrade performance. Contention can arise from increased capacity misses when

a cache-hungry thread reduces space in the cache for another thread's working set, or from increased conflict misses from collisions in set-associative caches. Unless process ids are maintained for cache lines, shared caches are likely to be physically indexed, making conflict miss prediction particularly difficult in the absence of page coloring information.

Hily and Seznec provide a detailed study of the performance impact of memory hierarchy parameters on a simulated SMT processor [4]. They claim that SPEC benchmarks typically used in previous simulations, such as in [11], are unrealistic for evaluating cache performance due to low miss rates. Using what they believe to be a more representative benchmark suite, they demonstrate the impact of cache contention on system throughput for different workloads.

Contention is workload specific and changes dynamically; therefore, it is hard to manage. Kim, Chandra and Solihin propose *fairness* in cache sharing to prevent the potential hazards of cache contention: thread starvation, priority inversion and unpredictability [8]. They use cache partitioning to implement fairness but fail to prove that their method improves throughput beyond a cache partitioning scheme that simply aims to minimize the collective number of misses. Fedorova, Seltzer and Smith implement fairness through the thread scheduling algorithm [2]. Clearly, a cache-greedy thread behaving in a fair fashion will improve the performance of its cache-sharers but only at a detriment to its own performance. Neither study successfully demonstrates that fairness is an effective criteria for improving overall system throughput.

Unfortunately, share and share alike doesn't improve application performance; unless a better argument can be made that it improves *system* performance, it does not appear to be a worthwhile general approach. On the other hand, ignoring the cache use behaviors of co-running processes can allow both application and system performance to fall prey to the Tragedy of the Commons. For threads from separate address spaces, the most we can do is to try to minimize the potential for cache contention. For shared address space threads, however, we can also try to take advantage of the shared memory hierarchy to improve performance.

2.3 Synchronized Pipelined Parallelism

Pipeline parallelism, or the *Producer-Consumer model*, is a parallel algorithm model that streams data across processors, relying on inter-processor synchronization to preserve data dependences [7, 3]. Vadlamani and Jenks propose using what they term the *Synchronized Pipelined Parallelism Model* (SPPM) to improve utilization of shared caches and memory interfaces on CMP and SMT systems [12]. Rather than working on separate data sets, concurrent threads work on the same data set in a pipelined manner. The synchronization interval must be long enough to permit written data to be ready for a thread that consumes it, and short enough to ensure that the data is still on-chip when the thread is ready to consume it. In addition to promoting on-chip data reuse and reducing contention for the shared cache and memory interface, this parallelization model can be used in applications

in which data dependences prevent parallelization by simply partitioning the data set.

Vadlamani and Jenks propose a set of metrics to be used for evaluating the usefulness of SPPM for a particular application. They apply SPPM by hand, using trial and error to determine the synchronization interval, to two applications: a Gauss-Seidel equation solver and a Finite Difference Time Domain electromagnetic simulation. They run their experiments on a SMT processer, the Intel P4 with hyperthreading, and show an execution time improvement of roughly 10% and 38% respectively over parallel versions using spatial decomposition, and of 21% and 17% respectively over sequential versions.

Unfortunately, their evaluation method only assesses the usefulness of SPPM for a given application *after* already obtaining the results of applying SPPM to the application, and they do not yet address load balancing or accommodate more than a single producer and consumer thread.

3 Synchronization Analysis

Multicore processors present the opportunity for cheap on-chip synchronization. Synchronization overhead is a significant factor in parallel algorithm design and implementation. In particular, it impacts task granularity, load balancing, and, therefore, the achievable parallel efficiency. To gain an understanding of the synchronization costs on our experimental setting, we choose an efficient synchronization method and evaluate its performance using a simple microbenchmark.

3.1 Synchronization Method

To synchronize between threads running on separate cores, we implement a lock using a pair of counters, NOTIFY and WAIT. The producing thread increments NOTIFY when it has produced data; the consuming thread increments WAIT when it is ready to consume data and busy waits while WAIT is less than NOTIFY. Pseudo-code for the notify and wait operations is provided in Figure 1. The counters are declared volatile to prevent reading stale values from registers.

```
notify(lock){
     lock->notify++;
}
wait(lock){
     lock->wait++;
     while (lock->notify < lock->wait){ //spin }
}
```

Figure 1: Notify and Wait Operations

Since each thread only writes to distinct counters, there is no race condition on the values of the counters. Furthermore, assuming the NOTIFY and WAIT counters are not in the same cache line, reading from a variable written by a process on the other core should result in an exclusive ownership invalidation but not an eviction from the primary cache of the writing process. To ensure this and avoid false sharing, we buffer each counter with enough memory to total the size of a cache line.

Note that the use of busy waiting requires that threads run on separate processors for efficiency. Otherwise, one thread expending its scheduled quantum by busy-waiting will needlessly postpone the execution of the thread upon which it's waiting.

3.2 Microbenchmark

Our microbenchmark increments a local variable inside a simple loop for some number of iterations. Each thread has two locks, a writer lock and a reader lock. If two threads are running, each thread's writer lock corresponds to the other thread's reader lock, and vice versa. If one thread is running, writer lock and reader lock point to the same lock. For each iteration of the loop, the thread calls notify on its writer lock, increments its local variable, and then calls wait on its reader lock. The pseudo-code for the loop is presented in Figure 2.

```
int acc=0;
do i=1, LIM{
     notify(writer_lock);
     acc++;
     wait(reader_lock);

}
```

Figure 2: Microbenchmark Kernel

3.3 Performance and Evaluation

To evaluate the synchronization overhead, we run the microbenchmark with a single thread and with two threads. Each thread runs only on a single core. In addition, we time a loop that increments a local variable without calling the synchronization primitives, and a two-threaded version in which the notify counters are initialized to LIM so that threads never have to actually wait when they call the wait operation. Times in seconds, averaged over 5 runs, are presented in Table 1 for loops running for 10 million iterations.

The single CPU overhead of executing the synchronization operations can be calculated as the difference between the single-threaded versions with and without synchronization:

Table 1: Time (s) per 10M Iterations

Threads	seconds
1 (no synchronization)	0.0248
1	0.087
2 (initialized notifies)	0.334
2	1.721

0.00622 seconds per million notify and wait pairs. The inter-processor communication overhead can be calculated as the difference between the times of the two-threaded version with initialized notifies and the single-threaded version: 0.02474 seconds per million notify and wait pairs. Finally, the collective time spent waiting on the other thread can be measured as the difference between the two-threaded versions. We see that this is the largest component of the synchronization overhead and takes 0.1387 seconds per million notify and wait pairs.

The single CPU overhead for a pair of synchronization primitives is simply the cost of reading one data element and updating two data elements located in the primary cache of the process. We implement the operations using macros instead of function calls to avoid additional overhead. This overhead is therefore determined by L1 hit latency. The inter-processor communication overhead includes cache line state changes and reading values from the other core's cache, which is determined by L2 latency along with bus cycles. The L2 latency on the Intel Core Duo is 14 cycles, whereas the L1 latency is 3 cycles. The ratio of L2 to L1 latency is around 4.67, whereas the ratio of inter-core to single-core overhead is roughly 3.98. Therefore, without the additional time spent waiting on another thread, inter-core synchronization is rather cheap at roughly the cost of incurring L1 misses.

4 Application Analysis

An important class of applications, including those used for solving partial differential equations and image processing, is characterized by iterative stencil computations in which spatial loops iterate over a data domain while encompassing loops represent time steps [6]. The stencil computation repeatedly updates data elements using their current values and the values of neighboring elements. The time domain carries many dependences due to reusing the memory locations of previous iterations– the data domain of the enclosed spatial loops. We will use a 2D Gauss-Seidel kernel as a representative example from this class of applications.

4.0.1 Time Skewing Techniques

Time Skewing [13], a technique introduced by McCalpin and Wonnacott, improves temporal locality beyond what is obtainable through data tiling alone for iterative stencil

computations. It applies tiling in both the data and time domains by skewing a time step loop and a spatial loop over a data tile, preserving dependences carried by the time domain. However, this method does not handle dependences carried by the data domain.

Song and Li also use loop skewing and tiling to improve L2 cache temporal locality [10]. By using odd-even duplication, their method handles codes with dependences carried by the data domain at the expense of requiring additional storage. Both of these methods are confined to skewing a single spatial dimension.

In comparison with the previous methods, Recursive Prismatic Time Skewing [6], introduced by Jin, Mellor-Crummey and Fowler, can skew multiple spatial dimensions and handles dependences carried by the spatial loops without requiring extra storage. Additionally, it uses bi-directional skewing to handle periodic boundary conditions.

RPTS skews spatial loops across time steps to improve temporal reuse as well as with respect to other spatial loops to enable recursive blocking, and then applies recursive blocking to achieve effective cache reuse. The iteration space over the data domain can be visualized as a set of skewed time-space prisms. Prismatic tiles thus depend upon border data produced by neighboring prisms in the previous time step.

Jin et al. compare the performance of RPTS with the performance of spatial skewing and blocking, Wonnacott's time skewing, and the method by Song and Li on a 2D Jacobi kernel for different problem sizes. This kernel uses Jacobi relaxation with a four point stencil and has no carried dependences on the spatial loops. Spatial skewing and blocking alone achieves a dramatic improvement over the original code, while all time skewing methods achieve significant improvement over just spatial skewing and blocking. Among the time skewing methods, RPTS performs consistently the best when it uses as much storage as the other methods to avoid extra copying. RPTS can also be implemented to use reduced storage at the expense of extra copying. In this case, it still performs better than the Wonnacott method, but it only surpasses the Song and Li method for the largest problem size and still falls quite short of the RPTS implementation that uses more storage to avoid copying.

4.0.2 Parallel Time Skewing

Wonnacott offers a parallel version of time skewing for multiprocessor architectures [13], but because RPTS demonstrates consistently superior performance in sequential versions [6], we focus on a parallel implementation of RPTS for the Gauss-Seidel kernel.

In his programming model for hierarchical shared memory, Woodward proposes analyzing the costs of a variety of system factors to determine when repeated computation is more efficient than communicating data [14]. Along with system size, he parameterizes a system by bandwidth/flops and flops/latency to different levels of shared memory. The ratio latency/flops determines the data access granularity, and bandwidth/flops, the amount of data reuse, necessary to achieve high parallel performance. Because interaction overhead is high in distributed shared memory clusters, Woodward uses asynchronous interaction to

overlap the cost of communicating data with computation, and repeated computation to minimize interaction overhead as much as possible.

To preserve data dependences between neighboring prisms, a parallel version of RPTS must either synchronize between time steps of neighboring prisms, communicating the border data as well if it is not already in shared memory, or repeat computation of border data to avoid incurring communication costs. Prisms can be computed in any ordering that preserves the data dependences between them. Lexicographical ordering and Morton ordering are two methods mentioned in [6]. Morton ordering further increases the potential for temporal reuse by sharing along two border regions instead of one.

The trade-off between repeated computation and communication obviously depends on the relative costs of each, which are system-dependent. In the shared-cache multicore environment, cheap communication costs are likely to outweigh the advantages of repeating computation except when the amount of computation is trivial.

If concurrent threads recompute the border data they need, duplicating the computation in order to work on separate data sets, they will also have to duplicate *storage* for the values produced by the repeated computation. Therefore, this approach will increase the collective cache footprint on a shared-cache architecture in comparison with a pipelined execution. Cheap on-chip synchronization and data sharing at the secondary cache level suggest that pipelining would be a better approach.

As we saw in Section 3.3, the largest component in overhead for our synchronization microbenchmark was the time spent waiting on the other thread. In a pipelined parallel RPTS, it is unlikely that either thread will have to wait on the other, once the pipeline is full, with an appropriate balance between synchronization interval and task granularity.

5 Pipelined RPTS Performance Results

We evaluate a pipelined version of Recursive Prismatic Time Skewing applied to a 2D Gauss-Seidel kernel benchmark for different tile volumes, aspect ratios, and synchronization intervals. Prisms are computed in lexicographical ordering in both the pipelined parallel and sequential executions. Each thread has a writer and reader lock corresponding to the other thread's reader and writer locks respectively. A thread calls notify on its writer lock after finishing K time steps, and calls wait on its reader lock before performing the first of each K time steps. Therefore, each thread executes a total of T/K notify and wait pairs per prism it computes, where T is the total number of time steps. The notify count of the first thread's reader lock is initialized to K, since the first prism has no data dependences.

5.1 Tile Volume and Synchronization Interval

In Figure 3 we show speedup over the sequential version for different values of K when the block values are equal for row and column dimensions. Block values are in array elements, which are Doubles (8 bytes each). All versions except for those with block values 10 used

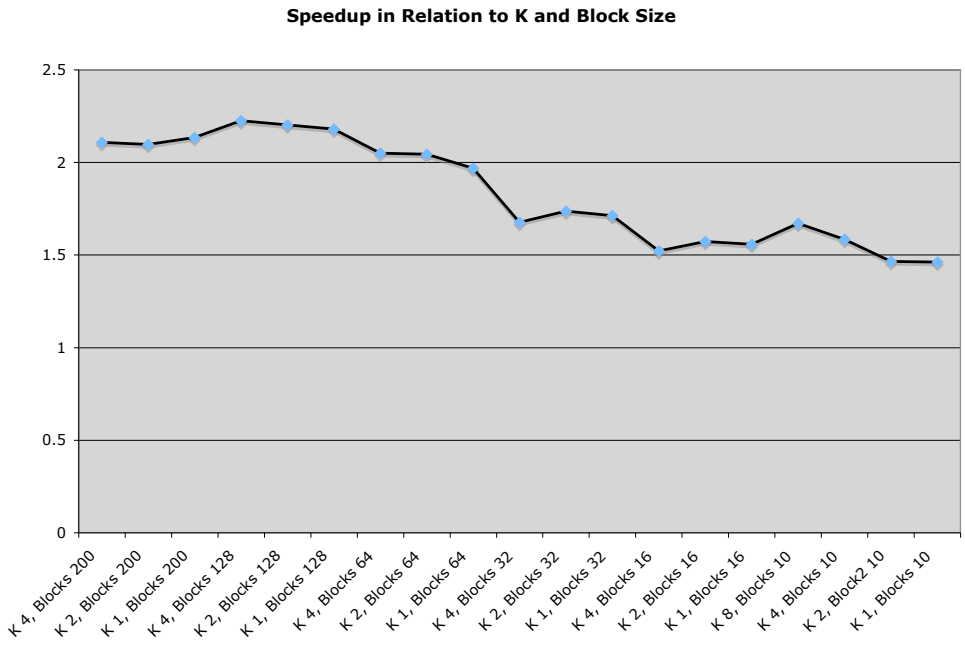

Figure 3: Speedup for Square Tiles

a total of 8 time steps and K values of 4, 2 and 1. In addition, the problem size was kept consistent for these versions (upper bounds of 6400 for both row and column loops). The last set used a total of 16 time steps, block values of 10, and K values of 8, 4, 2, and 1 (and upper bounds of 3200). Figure 4 depicts the relationship between tile volume and the number of notify and wait pairs used for these versions, excluding the smallest block sizes to retain visibility.

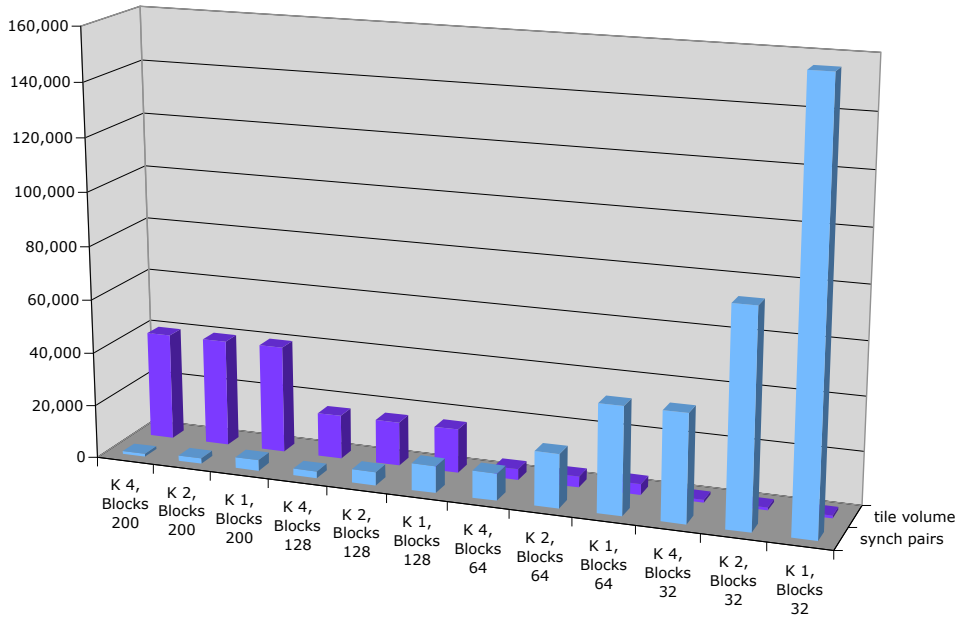

Figure 4: Tile Volume and Synchronization Pairs (Square Tiles)

The pipelined parallel version achieves a speedup of around 2 or greater for block sizes larger than 32. At block size 32 and under, speedup drops as the tile volume decreases and the number of synchronization pairs increase dramatically. For the larger block sizes, we see the best performance for block values of 128, and we see that the number of synchronization pairs is most balanced with tile volume for this block size. Among the results for this block size, we see that performance drops slightly as the synchronization interval decreases. For the larger block size of 200, performance improves a little at K1, but the synchronization

interval is still too small in relation to the tile volume. Although this block size performs better than 64, the task granularity is too large in comparison with a block size of 128.

5.2 Aspect Ratio

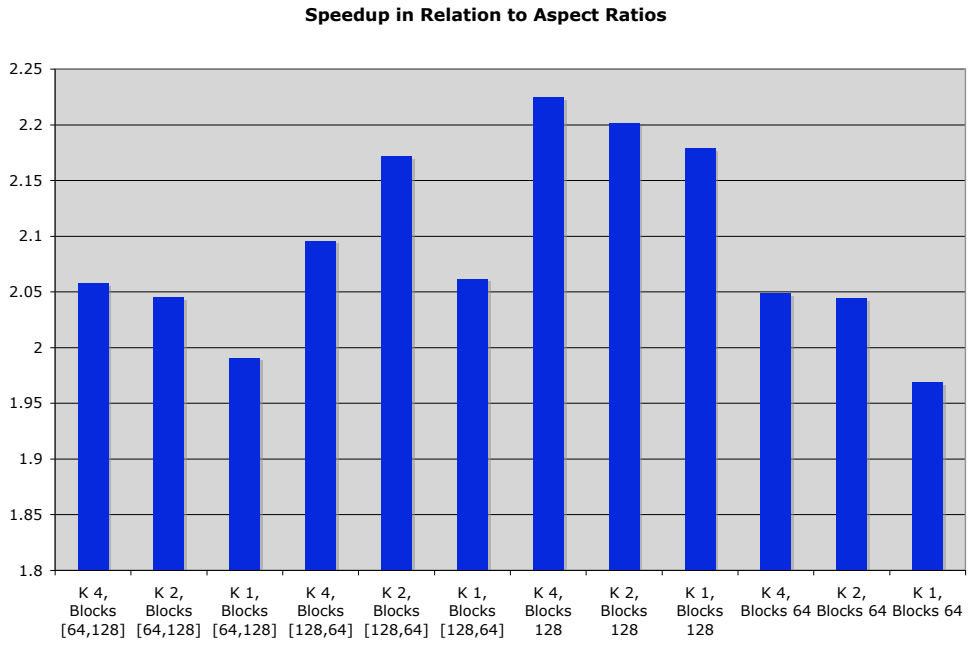

Figure 5: Speedup and Aspect Ratios

Figure 5 shows speedup for tiles with different aspect ratios using block values of 128 and 64. Block values are presented as [Row,Column]. Results for versions using square tiles with the same block values are included for comparison. Because the stencil computation uses neighbors in both row and column dimensions, the aspect ratios of 1 at the best block size 128 perform the best. The second best performance is achieved by versions with the best block value for the row dimension and aspect ratio of 2 (rows/cols) at K of 2 and 4 respectively. Square tiles with block size 64 and the [64,128] version perform roughly the same for corresponding values of K, and the same as [128,64] K1 at their best K value 4.

Since the code was written in C, which is row major, using a larger row dimension leads to better locality than using a larger column dimension. Consistent with the square tiles, performance declines a little as K drops from 4 to 2 (except in the case of [128,64]), then drops a larger fraction as K drops to 1. The one anomaly is [128,64] K2, in which case performance increases sharply. [The author cannot currently explain this anomaly!]

5.3 Summary

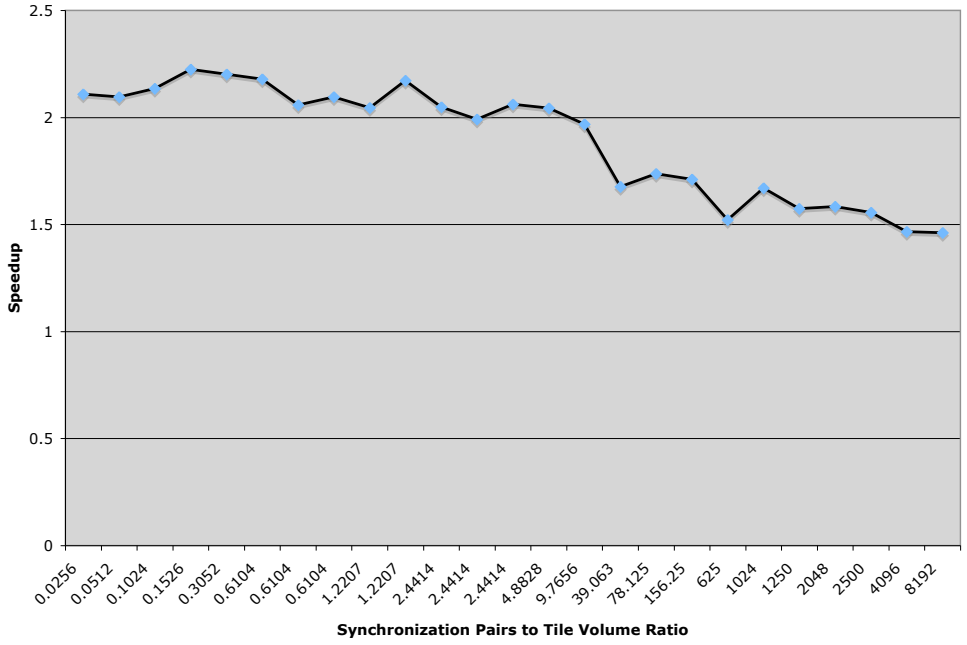

Figure 6: Speedup in Relation to the Ratio of Synchronization Pairs to Tile Volume

In Figure 6 we show speedup in relation to the ratio of synchronization pairs used to tile volume across all versions, in ascending order of ratios. [Note: it would be better to use the ratio of synchronization pairs *per prism* to tile volume.] Despite small fluctuations, we see a general decline in performance as the ratio increases. A block size of 128 in both rows and columns and K 4 achieves the best performance among all versions, a speedup of

2.22.

As long as the tile volume, and thus the task granularity, is large enough, we achieve a speedup of at least around 2, and thus a parallel efficiency of at least 1, because of low synchronization overhead: after the first K time steps, threads don't have to wait on each other, and communication is cheap at roughly the cost of L2 cache latency (14 cycles). Additionally, data reuse between cores improves our speedup further. A comparison with Morton ordering and a closer examination of the cache effects using a performance analyzer will help us gain a better understanding of the effects of data reuse in this benchmark. We propose to examine these in the near future!

References

[1] Intel Corporation. Intel 64 and IA-32 architectures optimization reference manual. http://www.intel.com/design/processor/manuals/248966.pdf.

[2] Alexandra Fedorova, Margo Seltzer, and Michael D. Smith. Cache-fair thread scheduling for multicore processors. Technical Report TR-17-06, Harvard University, 2006.

[3] Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. *Introduction to Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[4] Sébastien Hily and André Seznec. Contention on 2nd level cache may limit the effectiveness of simultaneous multithreading. Technical Report PI-1086, IRISA, 1997.

[5] Jaehyuk Huh, Changkyu Kim, Hazim Shafi, Lixin Zhang, Doug Burger, and Stephen W. Keckler. A NUCA substrate for flexible CMP cache sharing. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 31–40, New York, NY, USA, 2005. ACM Press.

[6] Guohua Jin, John Mellor-Crummey, and Robert Fowler. Increasing temporal locality with skewing and recursive blocking. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 43–43, New York, NY, USA, 2001. ACM Press.

[7] Ken Kennedy and John R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[8] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. *Proceedings. 13th International Conference on Parallel Architecture and Compilation Techniques (PACT 2004)*, pages 111 – 22, 2004//.

[9] Basem A. Nayfeh, Lance Hammond, and Kunle Olukotun. Evaluation of design alternatives for a multiprocessor microprocessor. In *ISCA '96: Proceedings of the 23rd annual international symposium on Computer architecture*, pages 67–77, New York, NY, USA, 1996. ACM Press.

[10] Yonghong Song and Zhiyuan Li. New tiling techniques to improve cache temporal locality. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 215–228, New York, NY, USA, 1999. ACM Press.

[11] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. *Conference Proceedings - Annual International Symposium on Computer Architecture, ISCA*, pages 392 – 403, 1995.

[12] Srinivas N. Vadlamani and Stephen F. Jenks. The synchronized pipelined parallelism model. *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems*, 16:163 – 168, 2004.

[13] David Wonnacott. Time skewing for parallel computers. In *LCPC '99: Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing*, pages 477–480, London, UK, 2000. Springer-Verlag.

[14] Paul R. Woodward and S. E. Anderson. Portable petaflop/s programming: Applying distributed computing methodology to the grid within a single machine room. In *HPDC '99: Proceedings of the The Eighth IEEE International Symposium on High Performance Distributed Computing*, page 9, Washington, DC, USA, 1999. IEEE Computer Society.