Parallelizing Programs for the Multicore Era

David Liao, Ryan Morey, and Alex Rucker

July 25, 2013

1 Abstract

Modern processors are very advanced but still suffer the bottleneck of slow memory access speeds. Processors now have the ability to run multiple tasks in parallel. This means that the processor runs multiple tasks concurrently, rather than sequentially, thus reducing the effect of memory access times.

Utilizing this ability is called parallelization. There are many standards in place that allow programmers to implement this technique in their programs. This can improve the speed of the program, as well as reduce memory usage.

The techniques of parallelization, also called multithreading, are investigated in the context of matrix multiplication, an important mathematical operation. Also studied are the methods by which a processor stores and retrieves memory to the cache, which exists within the processor itself, and methods by which cache usage can be optimized. Parallelization was found to have a significant improvement on program running time, achieving up to 5 times speed boosts in certain cases.

2 Introduction

As fabrication processes increased the transistor density of processors in accordance with Moore's law, multiple cores became a popular solution to the pressing challenge of utilizing this extra space. Multiple cores allowed for processors to scale linearly with the increasing area, and enabled certain programs to increase their performance. Now, processors can have dozens of cores, and therefore have the ability to vastly increase the performance of certain applications.

However, to fully utilize all computational power available, programs must be restructured to be run as independent steps, which run simultaneously, a process called parallelization. Although this is not optimal for certain applications that do not have repetitive steps, others, like matrix multiplication, can easily be separated, because the nature of the computation is such that, theoretically, each value in the resultant matrix can be computed independently. Therefore, a program is written to allow for matrices to be multiplied in parallel; to help increase the performance of the program, its runtime is analyzed across a variety of conditions and optimizations.

2.1 Matrix Multiplication

Matrix multiplication is important in a variety of fields. Matrices are used to solve systems of linear equations, which appear in many applied mathematics problems. Furthermore, the time required to perform a matrix multiplication scales at least quadratically as the size of the matrix increases, rendering it difficult to perform on the large data sets that must be processed in many high-performance computing applications.

3 Multithreaded Programming

3.1 CPU Architecture

The central processing unit (CPU) carries out the instructions of a computer program by performing arithmetic, logical, and input/output operations. While some CPUs carry out instructions in order (serial), certain CPUs employ a parallelized architecture at the instruction level (superscalar) in order to speed up CPU throughput. Superscalar CPUs are able to carry out more instructions at a given clock rate than normally possible by dispatching multiple instructions. Such design gives rise to the parallel capabilities of CPUs of today. Certain CPU operations, memory access particularly, take a much longer to finish than others, so certain Intel CPUs implement a technique called Hyper-Threading, which allows the CPU to perform many faster operations concurrently

as the slower ones finish. For each physical core that exists, the operating system assigns two logical cores (threads) that share workload. By taking advantage of the superscalar architecture, Hyper-Threading increases the amount of instructions run in parallel as opposed to those run in serial.[2]

A computer has many separate levels of memory, including caches and main memory, as well as swap space on disk. Of these, caches are the smallest, and can be accessed fastest, with main memory being larger and slower. In systems with little memory, additional disk space can be allocated by the Operating System (OS), but this space is extraordinarily slow. This additional space allows systems that would otherwise not have enough memory to run, and is acceptable for programs that do not regularly access all their memory. Although many data sets can fit entirely in main memory, thereby eliminating the time required to switch out data to a slow drive, most large data sets cannot fit entirely into even the highest level cache.

Caches form an important part of the memory hierarchy of any modern system. They are controlled at runtime by the CPU, and consist of multiple levels. The main purpose of caches is to store accessed or computed data in hopes of reusing it in the near future, thus increasing computing efficiency. Due to inherent difficulties in fabricating large, fast memory, each cache level has a different size and speed. As the level of the cache increases, the amount of data that the cache can store also increases and the speed decreases. Caches are structured such that each core has its own level one (L1) cache while L2 and L3 are generally shared.

Caching relies on the processor being able to reliably predict the future memory behavior of the program it is executing. It does this by making several assumptions about memory accesses. The first is that if a datum is used, it will be used again soon; this is called temporal locality. The second idea, called physical locality, is that if a datum is used, data around it will be used soon, and it is therefore wise for the processor to fetch these data for immediate use. These behaviors guide the structure of programs, because if a program is structured in a way that the processor does not understand, the processor will not be able to perform to its maximum capacity.

3.2 Multithreading

To increase interactivity, modern systems run multiple tasks at once, as opposed to running all tasks one after the other. Each application can contain one or more processes, each of which is treated as a separate entity by the CPU. Each process can contain multiple threads. Threads are composed of a stream of instructions, which are to be executed on a CPU core; by splitting them, it is possible to use multiple cores at once. Threads share virtual memory (processes do not), which allows for easy inter-thread communication: all a thread has to do to communicate with another thread in the same memory space is simply to write its data to memory.

Multithreading requires modifications to programs and data structures beyond simply splitting them, however. Because each thread is an independent stream of operations, and each is scheduled independently, it is possible for a thread to execute in an order other than that which would be expected. Therefore, to rectify this, techniques must be taken to ensure that no two threads can create problems by accessing the same resource at the same time.

To ensure the integrity of memory, a technique known as mutual exclusion can be used. In this technique, a special data structure known as a mutex is used to control access to data. When a thread attempts to access certain data, it must attempt to secure the mutex. However, only one thread can have an exclusive lock on the mutex at a time. Therefore, if two threads attempt to modify the data, one of them will be unable to lock the mutex, and it will have to wait for the other thread to finish and release the mutex.

Another problem that arises when attempting to write multithreaded code is the problem of allocating work. Because the execution of threads is inherently unpredictable, any work allocation system must be able to account for differences in the time that each thread is allowed to run on the processor. Furthermore, allocation has a fixed cost, because mutual exclusion prevents other threads from accessing memory while the operation is in progress. This places constraints on any allocation algorithm, because it must be fine-grained enough to not waste potential thread productivity, but cannot be too fine or else productivity will suffer

The final issue with multithreading, and programming in general, is modifying threads to properly adapt to cache behavior. Because threads share the same memory space, it is possible for them to share the same data structures. It is important that deduplication of data be performed whenever possible—if a thread keeps its own copy of data, it will have to be

duplicated across every thread, which, in the event of a large dataset, can quickly become costly.

3.2.1 Pthreads

Pthreads is the POSIX-compliant threading interface that ships with every UNIX-like system. It is a C library that allows for threads to be created, as well as for other constructs, such as mutexes, to be managed. When a thread is created using Pthreads, it begins to execute a function; the only argument to the function are a single pointer. This helps to keep the size of the private data kept by each thread low, and thereby helps to increase performance. Mutexes in the library are blocking, which means that, when a program attempts to lock a mutex and fails, it will simply stall until the mutex becomes available.[1]

3.3 Intel Performance Counters

Although there are many techniques for profiling the behavior of code, Intel processors contain specialized registers known as Performance Counters. Each of these can be configured to count a certain hardware event; the total count can then be read back at the end of the program's execution. By tracking events in this manner, it is possible to perform highly precise monitoring of what actually goes on within a core, including the detection of cache misses in every level of the cache.

4 Matrix Multiplication

In matrix multiplication, each element in the product matrix is the result of the dot product of a row from the first matrix, and a column from the second. The most basic algorithm for matrix multiplication of two square matrices, of size $n * n$ runs in $O(n^3)$ time. This is the algorithm implemented. Without loss of generality, all testing was done with two square matrices.

4.1 Multiplication Algorithms

4.1.1 Serial Algorithm

The serial algorithm is the point of comparison for the parallelization. The program computes each element of the resultant matrix one at a time. It does this by first iterating over all the calculations necessary to compute one element in the result matrix, then by iterating over all the elements in a row of the result matrix, and then by iterating over all the rows in the result matrix.

4.1.2 Parallel Algorithm

This program was parallelized by splitting up the computations that had to be performed over a series of worker threads. The program first instantiates t worker threads. Each of these would then attempt to access a mutually-exclusive counter, which held the index of the next row available to be computed. When the thread gained access to the counter, it would decrement it by a certain amount, and release the counter. That amount was either a fixed n or chosen at random from the range 1 to n. That amount would be the number of rows the worker thread would compute, after which it would return to the counter mutex for its next rows. This algorithm was tested for values of n ranging from 1 to $\frac{\text{rows}}{\text{threads}}$.

Another optimization that was implemented was the transposition of the second matrix in memory. To transpose a matrix is to convert rows into columns and columns into rows. In the original method of storage, elements in the same row are stored adjacently. This has different effects on the accessing of first matrix as opposed to the second, because the CPU operates on an assumption of physical locality. In the first matrix, the data is traversed along rows, adhering to the CPUs assumption that entries in memory which are stored near to each other are likely to be accessed successively. This is ideal for the first matrix, but is pessimal for the second. The second matrix is accessed by column, meaning that if the matrix is stored by row, the program almost never accesses adjacent locations in memory, increasing the number of cache misses. For this reason, the program was written so that the first matrix is stored by rows, and the second is stored by columns. This is not strictly parallelization, but is very relevant to optimization, and demonstrates the effect of cache misses on performance.

4.2 Parallel Matrix Multiplication

The study was conducted in a UNIX environment with C as the primary language. To generate matrices used as testing inputs, a C program created a text file consisting of user-defined number of random doubles. These are formatted into a vertical file and are read in at run-time by another program that performs the matrix multiplication. The program handled the resource allocation and the multiplication at

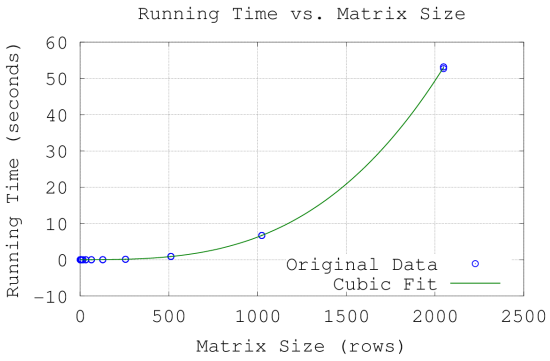

Figure 1: Serial algorithm performance

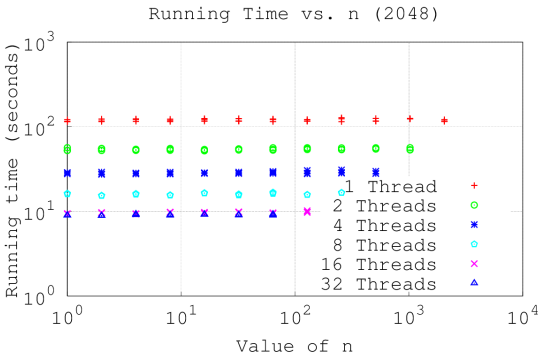

Figure 2: Performance for different values of n

the same time. The output of that program is a matrix of appropriate size which, at first, was stored to check for code correctness; the resultant matrix was later deemed unnecessary data and was not stored after computation.

5 Automated Testing

Most of the data that was generated came from automated Bash scripts that iterated through multiple conditions. Using the Bash time command, each script recorded the times of the various instances of matrix multiplication. The first script that was written with the purpose of investigating the runtime growth rate as matrix size grew and number of threads remained the same. The second script, which was split into two sections, investigated the effect of modulating the amount of threads while maintaining a constant matrix size. The second part to the second script was to investigate the effects of transposition. Similarly to the first part, the second part scaled the amount threads while maintained the matrix size constant. Finally, the last script that was used recorded the effects of modulating both the size of the matrix and the amount of threads. In addition, the allocation of work to the threads was also modulated to find the most optimal method of assigning work and resources.

6 Results

The data in Figure 1 show that the running time of the matrix multiplication algorithm increases cubically with the size of the matrix; this means that the performance of the algorithm was exactly what was predicted theoretically. This indicates that the program functioned as expected.

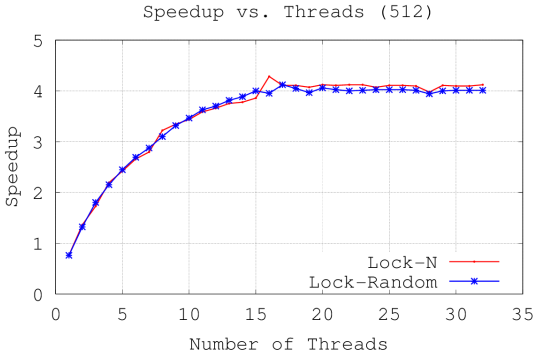

Figure 3: Speedup as thread count varies for size 512 matrix

Figure 2 shows the effect of changing the number of matrix rows allocated to each thread. Each different type of marker indicates a different number of threads. From this graph, it is evident that there is no major increase or decrease in running time as n is changed.

Figures 3 and 4 shows the speedup achieved by each of the two work-allocation algorithms for different matrix sizes. Here, it is evident that there is a plateau in the speedup that can be attained, which, depending upon the size of the matrices being calculated, ranges from around 4 to 6. The plateau begins at 16 threads, which is the number of physical cores in

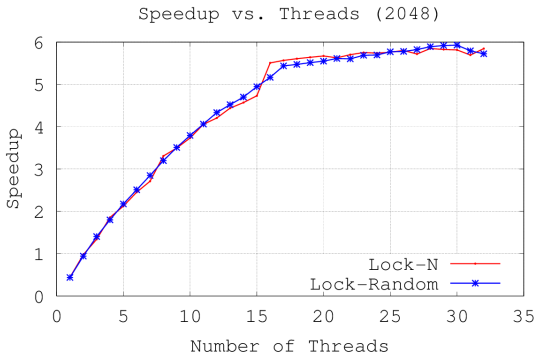

Figure 4: Speedup as thread count varies for size 2048 matrix

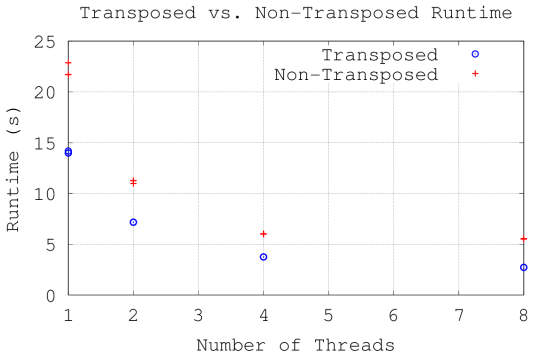

Figure 5: Runtime for changes for transposed matrices

the CPU; therefore, it is likely that the performance of the program is not dependent on the number of threads that can be run at once, but on the memory performance of the processor. If the speedup depended on the number of threads that could be run at once, it would continue to scale as threads began to share cores. However, there is no ability for the program to scale past the memory bandwidth of each core by adding more threads per core, because a single thread is capable of saturating the memory bandwidth.

Note that the first data point, where 1 thread is used, is actually below a speedup of 1. This is due to the significant amount of overhead involved in multithreading.

The final test that was run compared the speed of

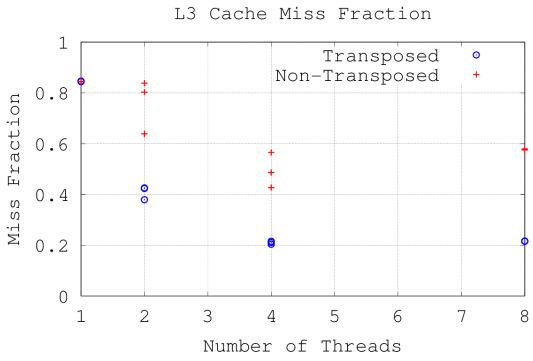

Figure 6: Fraction of L3 cache misses for transposed vs. non-transposed matrices

the program when the matrices were transposed to when they were not. This test showed that, regardless of the number of threads, there was a significant improvement in the runtime of the program.

Next, a test was run using performance counters to determine what fraction of the accesses to the L3 cache were misses. This test showed that non-transposed matrices had a significantly higher fraction of misses; for 8 threads, 60% of all cache accesses were misses for the non-transposed matrices, compared with 20% for the transposed matrices. This confirms the hypothesis that the transposition of the matrix in memory would increase performance. Furthermore, there was reasonably low variability in the number of cache misses for the transposing program, suggesting that there were fewer conflicts with other programs for cache space. This is because, if there had been conflicts, other programs would have caused some matrix data to be unloaded, and this would be inherently unpredictable. The non-transposing program had a high variability in the number of cache misses, suggesting that it was much more sensitive to how long its data stayed in the cache.

7 Conclusions

First, a serial matrix multiplication algorithm was developed and tested. A parallel version was then devised, and its performance was measured. Several performance improvements were then made, including the transposition of matrix elements in memory, and a variety of work allocation algorithms were tested for the parallel version. Transposing the ma-

trix had major performance benefits, but varying allocation methods, and their parameters, had almost no effect on performance. Speedups were observed to be about 5 times, and were found to plateau at 16 threads. This is thought to be the result of a limit on memory and cache bandwidth being reached. The other 16 threads ran on virtual cores, which did not add any additional memory bandwidth.

8 Acknowledgements

The authors would like to thank Dr. Abhishek Bhattacharjee of the Rutgers University Department of Computer Science for his invaluable knowledge and mentorship. We also extend our thanks to all of the Resident Teaching Assistants for the NJ Governors School of Engineering and Technology, in particular Brian Belding, for his continued guidance. We would also like to thank the director and assistant director of the New Jersey Governors School for Engineering and Technology, Ilene Rosen and Jean Patrick Antoine respectively. Lastly, we extend our thanks to our sponsors: Rutgers University, the State of New Jersey, Morgan Stanley, NJ Resources, South Jersey Industries, PSE&G and the GSET alumni community.

References

[1] Barney, B. Posix threads programming.

[2] Marr, D. T., Binns, F., Hill, D. L., Hinton, G., Koufaty, D. A., Miller, J. A., and Upton, M. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal 06* (February 2002).