18.337/6.338: Parallel Computing
Final Project Report

Parallelization of Matrix Multiply

A Look At How Differing Algorithmic Approaches and CPU Hardware Impact Scaling Calculation Performance in Java

Elliotte Kim
Massachusetts Institute of Technology
Class of 2012
nakoruru@mit.edu

1.) Motivation

In the almost two years that I have spent at MIT as an undergraduate student, I found that 18.06 (Linear Algebra) was amongst my favorite subjects. I particularly enjoyed performing matrix calculations, and finding and interpretting the significance of eigenvalues. I thought it was interesting that matrix mathematics were applicable to other subjects I took, including 8.02 (Physics: Electricity and Magnetism), 6.01 (Introduction to Electrical Engineering & Computer Science I), and 6.045 (Automata, Computability and Complexity).

In taking Professor Alan Edelman's 18.337 (Parallel Computing) class, I felt this project would be a good opportunity to gain some Java Threads experience by implementing parallel matrix multiply. I was also very interested to see just what sort of performance results such code would have, in relation to serially-run calculations. Additionally, I was curious to see how differing hardware architecures might impact said code.

2.) Introduction

2.1) Parallel Computing

Traditionally, computer programs have been written to solve problems serially. To solve a problem, algorithms are constructed and implemented as a serial stream of instructions, that are executed on a single central processing unit of a computer. Only one instruction is executed at a time.... once one instruction is completed, the next instruction is subsequently executed.

Parallel computing, however, takes advantage of multiple processing units to solve a problem. This is usually accomplished by dividing up some, if not all, of the problem into independent parts, so that each processing unit can execute some parts of the problem simultaneously. Thus, parallel computing allows for problems to be solved more quickly, than if they were run serially.

Up until 2004, improving the computer performance of serially-run programs has heavily relied on increasing processor clock frequencies. Increasing a processor's clock frequency decreases the average amount of time it takes to execute an instruction. However, since the power consumption of a chip is proportional to its processor clock frequency, we are restricted in our ability to further increase processor clock frequencies. Increasing processor power consumption ultimately led to chip manufacturers, such as Intel and AMD, to switch computer architecture paradigms, from increasing clock frequencies of a single processing unit (or core), to adding additional cores onto chips, in order to further improve computational performance.

However, traditional computer programs designed to run serially does not take advantage of additional processing units. Programs must be written with parallelization in mind, in order to utilize additional processors.

Ideally, the speed boost from parallelization would scale exactly with the number of processing units available to a program. Running a parallel program on two processing units would take exactly half the time to complete as the same program on a single unit. However, very few parallel algorithms achieve such linear performance improvement. Instead, most exhibit performance results in accordance to Admahl's Law, which states that a small portion of the program cannot be parallelized will limit the overall speed-up from parallelization. The maximum speed-up of the program is given by the equation...

$$S = 1 / (1-P)$$

...where S is the speed-up of the program, and P is the fraction that is parallelizable. So, for example, if a program is 75% parallelizable, then we cannot get better than 4x speed-up via paralleization, regardless of how many processors we make available to the program.

Computationally intensive applications benefit from the use of multiple processors via parallelization. Matrix multiplication is a good example of this.

2.2) Matrix Multiplication

Matrix multiplication (matmul) is one of the most fundamental operations in linear algebra. Matmul serves as the primary operational component in many different algorithms, including the solution for systems of linear equations, evaluation of the determinant a matrix, and the transitive closure of a graph.

Since matrix multiplication has numerous applications, including the fields of graph theory, analysis, probability and statistics, physics, quantum mechanics and electronics, finding ways of performing matrix multiplication calculations faster has obvious value.

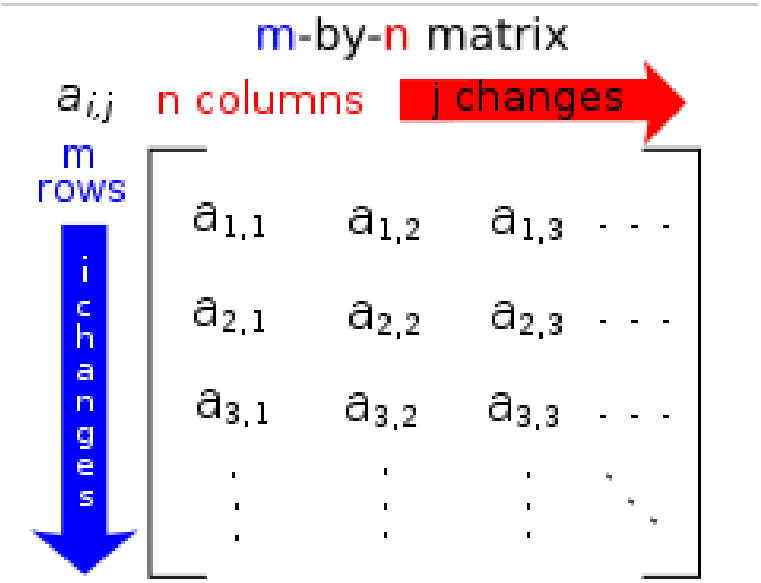
3.) Project Description

3.1) Problem

Based on a suggestion by Professor Edelman, I decided to compare the parallel performance of matrix multiplication for pairs of regular matrices, and for pairs of irregular matrices.

In particular, I was curious to see how long an (n x n) matrix multiplied by an (n x n) matrix would take, compared to that of an (n x kn) matrix multiplied by a (kn x n) matrix. In both cases, matrix multiplication would result in (n x n) dimensioned matrices.

Note: For notation purposes, I refer to an m-by-n matrix as "(m x n)". Similarly, the multiplication of an m-by-n matrix with a n-by-p matrix is denoted as "(m x n) * (n x p)".



$$A * B = C$$
$$(m \times n) \quad (n \times p) \quad (m \times p)$$

3.2) Hypothesis

My initial assumption was that, an (n x kn) * (kn x n) Matrix Multiply would require, at minimum, k-times the duration of an (n x n) * (n x n) Matrix Multiply, if the same matrix multiplication algorithm was applied to both calculations. I based this assumption on the notion that:

$[A_{11}] \times [B_{11}] = [A_{11} * B_{11}]$

whereas:

$$[A_{11}\ A_{12}\ ...\ A_{1f}] \quad x \quad \begin{bmatrix} B_{11} \\ B_{21} \\ [\ ...\ ] \\ B_{f1} \end{bmatrix} \quad = \quad [A_{11}*B_{11} + A_{12}*B_{21} + ... + A_{1f}*B_{f1}]$$

There would ultimately be k-times the number of $A_{1x}*B_{x1}$ operations (as well as k-times the number of addition operations, assuming that the resulting matrix had its element values initialized to the value of zero) involved in an (n x kn) * (kn x n) Matrix Multiply, versus an (n x n) * (n x n) Matrix Multiply, based on the example given above.

Would the results of experimental testing reflect my hypothesis?

3.3) Secondary Objectives

In addition to seeing how experimental results compared to my hypothesis, I also wanted to observe how differing matrix multiplication algorithms performed compared to each other, duration-wise.

Also, I wanted to see how dffering processing unit architectures impacted the performance of Matrix Multiply. Would the same algorithm run on two different processors run similarly (after taking processor clock frequencies into account).

4.) Implementation

4.1) Algorithms

I wrote two separate Matrix Multiply programs in Java, each one taking a different approach to calculate a matrix product.

ParallelMatrixMultiply
RecursiveParallelMatrixMultiply

Both programs take the following command-line arguments:

n — an Integer, the n value represented in multiplying a (n x kn) matrix A with a (kn x n) matrix B.

k — an Integer, the k value represented in multiplying a (n x kn) matrix A with a (kn x n) matrix B.

For example, to do a (2 x 2) matrix times a (2 x 2) matrix, you would input a value of 2 for n, and 1 for k. Similarly, to do a (2 x 4) matrix times a (4 x 2) matrix, you would input a value of 2 for n, and 2 for k.

In addition, ParallelMatrixMultiply takes the following command-line argument:

numProcs — an Integer, representing how many threads to create. This argument is optional. If it is not provided, then numProcs is assigned a value equal to the output of Runtime.getRuntime().availableProcessors();

Also, RecursiveParallelMatrixMultiply takes the following command-line argument:

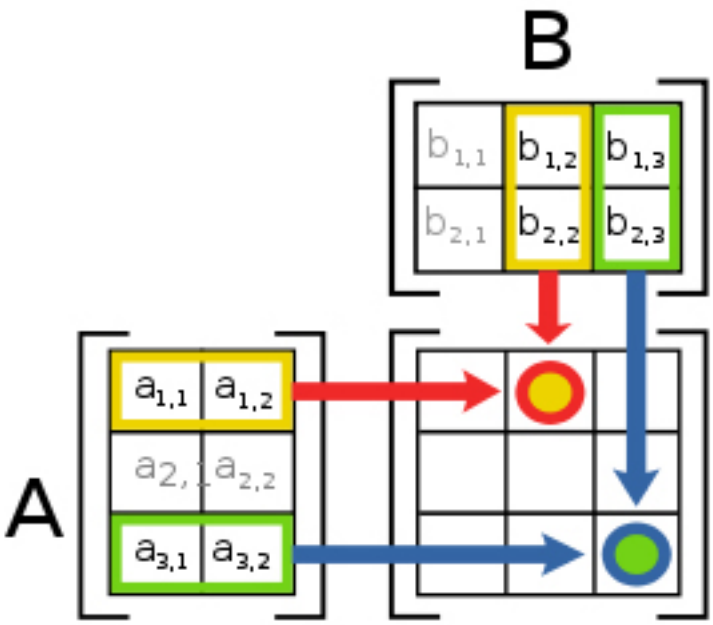
threshold — an Integer, specifying the maximum dimension a matrix can have and not be subdivided into smaller matrices

Both programs do the following:

1.) Randomly generate a (n x kn) matrix A, represented by an int[][] datatype.
2.) Randomly generate a (kn x n) matrix B, represented by an int[][] datatype.
3.) Create a (n x n) zero matrix C, represented by an int[][] datatype.

In the case of ParallelMatrixMultiply:

1.) It performs what Wikipedia (read entry: matrix multiplication) calls "ordinary matrix product."



$$\text{Matrix } \mathbf{A} \text{ is } 3x4 \begin{bmatrix} 8 & 3 & 0 & 1 \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{bmatrix} \begin{bmatrix} 5 & \cdot & \cdot & \cdot \\ 4 & \cdot & \cdot & \cdot \\ 3 & \cdot & \cdot & \cdot \\ 1 & \cdot & \cdot & \cdot \end{bmatrix} = \begin{bmatrix} 53 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{bmatrix} \text{ Matrix } \mathbf{C} \text{ is } 3x4$$

$$\text{because } c_{11} = \sum_{k=1}^{4} a_{1k} b_{k1} = 8 \cdot 5 + 3 \cdot 4 + 0 \cdot 3 + 1 \cdot 1 = 53$$

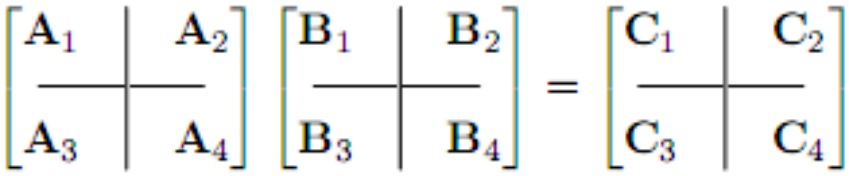
2.) Rows of matrix A (and consequently, the corresponding rows of matrixC) are assigned to particular threads.

Ex: If n=8 and numProcs=4, then 4 threads wil be created.
- Thread #0 will be assigned rows 0 and 4
- Thread #1 will be assigned rows 1 and 5
- Thread #2 will be assigned rows 2 and 6
- Thread #3 will be assigned rows 3 and 7

3.) Each thread then uses the "ordinary matrix product" methodology to calculate the values for the various entries for their respective rows in matrix C.

In the case of RecursiveParallelMatrixMultiply:

1.) It divides a large matrix into 4 smaller sub-matrices



Sub-matrix C_1 is obtained by calculating $A_1*B_1 + A_2*B_3$ where $A_1...A_4$ and $B_1...B_4$ are sub-matrices.

2.) Each sub-matrix is assigned to a new thread
3.) Steps 1 and 2 are repeated until a matrix has no dimension bigger than the specified threshold value mentioned above.
4.) A thread then performs "ordinary matrix product" serially (within the confines of the thread).
5.) The final result for matrix C is determined when all the Java threads are "joined."

The use of threshold was included in the RecursiveParallelMatrixMultiply code, since I believe there comes a point when the time overhead from continued spawning threads is greater than performing a serial matrix product calculation, for a sufficiently small sized matrix.

Note: The threshold value to get the fastest regular matrix multiplication performance may vary across different processor hardware.

The correctness of both programs was tested by using n values ranging from 1 to 8, and k values ranging from 1 to 4, and comparing results with BlueBit's online Matrix Multiplication calculator (http://www.bluebit.gr/matrix-calculator/matrix_multiplication.aspx).

4.2) Software Tools

As mentioned earlier, for this project I decided to write my matrix multiplication code in Java (version 1.6.0.16), using Java Threads and the Concurrent libraries.

Java Threads have their own stack call, but can also access shared data. Thus, there are two basic problem: visibility and access. A visibility problem occurs when thread A reads shared data which is later changed by thread B, but without thread A being aware of this change. An access problem occurs when several threads try to access and change the same shared data at the same time. Visibility and access problems can result in

responsiveness failure, due to data access issues such as deadlocks, or safety failure, where the program creates incorrect data.

Java's Concurrent libraries, which were first introduced in Java 1.5, overcomes these issues by adding datatypes that perform atomic updates, allowing for thread-safe read/write operations.

4.3) Hardware

The following processors were used in running both ParallelMatrixMultiply and RecursiveParallelMatrixMultiply:

Intel Atom N270
Details:

Clock Speed:	1.6 GHz
# of Cores:	1
# Threads per Core:	2
# Total Threads:	2
L1 Cache:	56 KB
L2 Cache:	512 KB

AMD Turion 64 X2 TL-60
Details:

Clock Speed:	2.0 GHz
# of Cores:	2
# Threads per Core:	1
# Total Threads:	2
L1 Cache:	128 KB per Core
L2 Cache:	1 MB total, (512 KB dedicated per Core)

Intel Core2 Quad Q6700
Details:

Clock Speed:	2.66 GHz
# of Cores:	4
# Threads per Core:	1
# Total Threads:	4
L1 Cache:	128 KB per Core
L2 Cache:	8 MB total (shared)

On each platform, I ran the code using the following parameter values:

n = 1024
k ranging from 1 to 8
numProcs ranging from 1 to 2

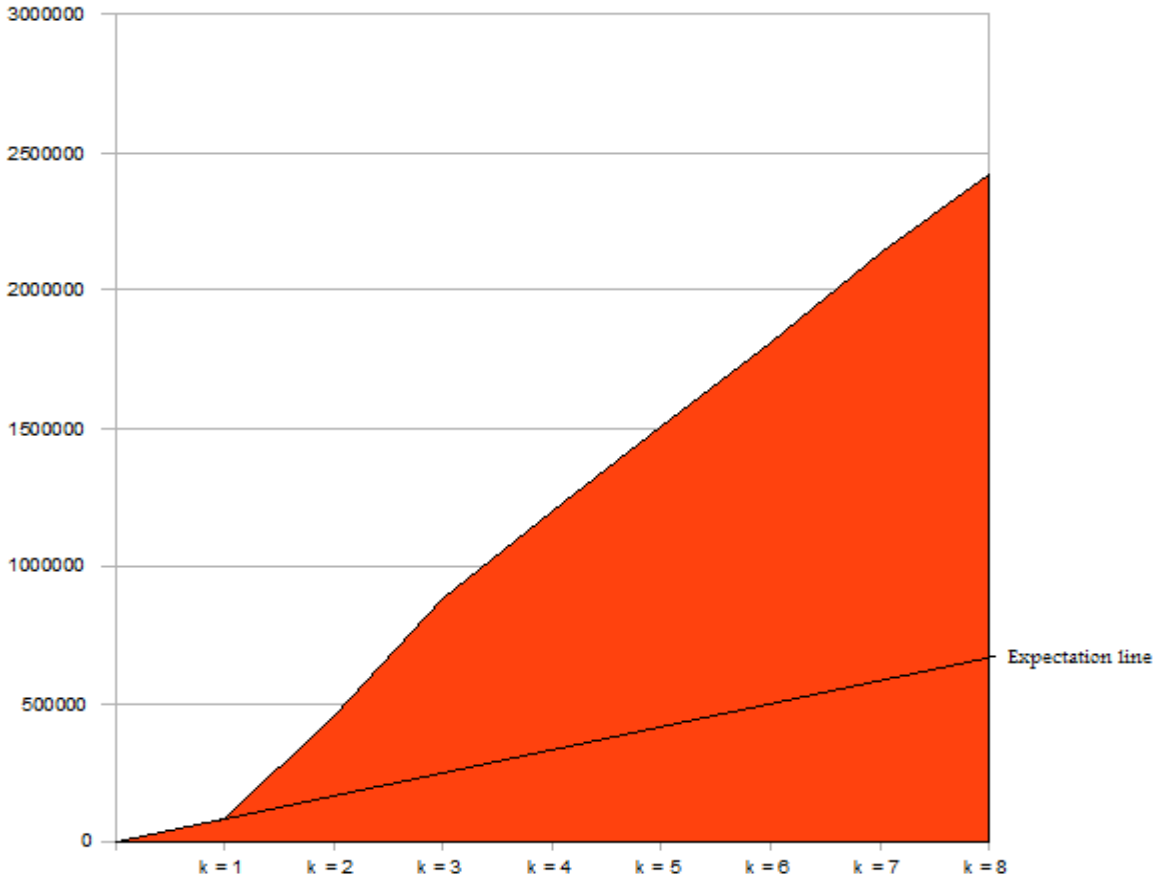
and collecting time results for each. In addition, I also used numProcs value of 4, in the case of the Intel Core2 Quad Q6700, since that CPU has 4 cores.

5.) Results and Observations

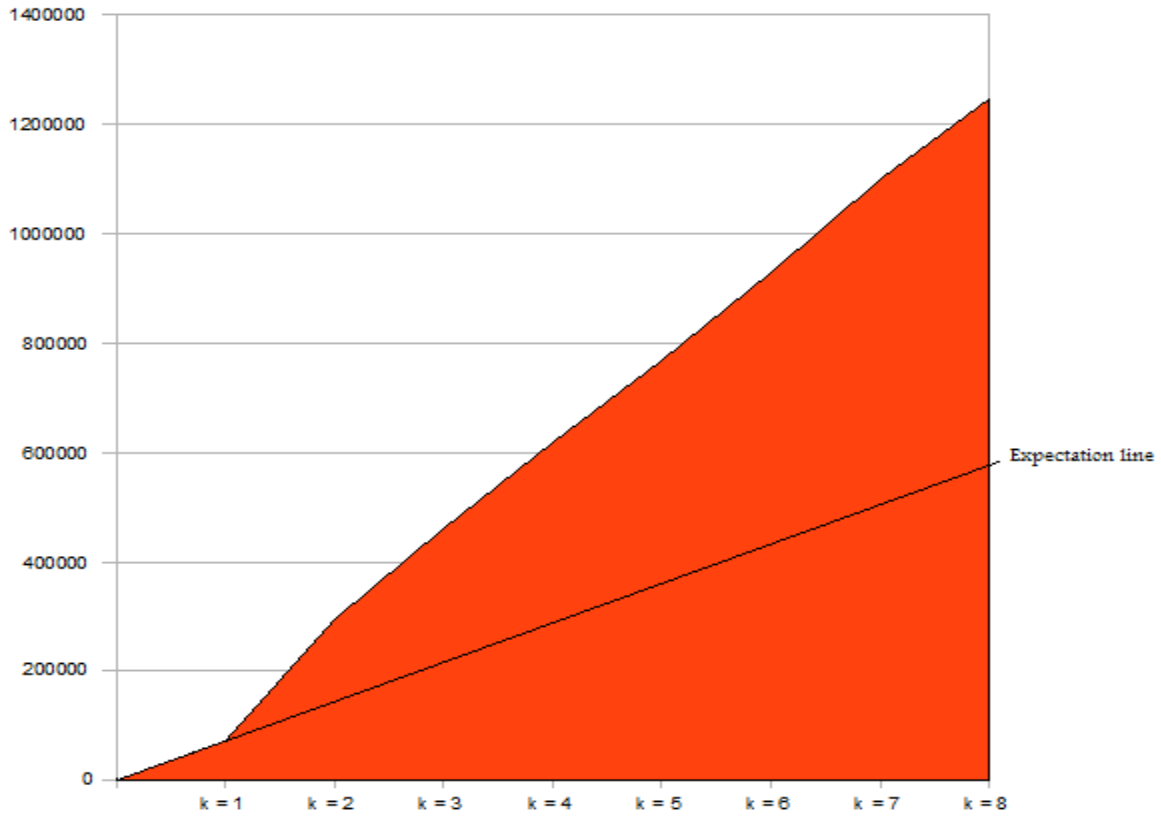
In all test cases, matrix multiplication calculation times were measured in milliseconds.

5.1) ParallelMatrixMultiply

5.1.1) Intel Atom N270



1 Thread. n = 1024. y-axis measured in milliseconds.



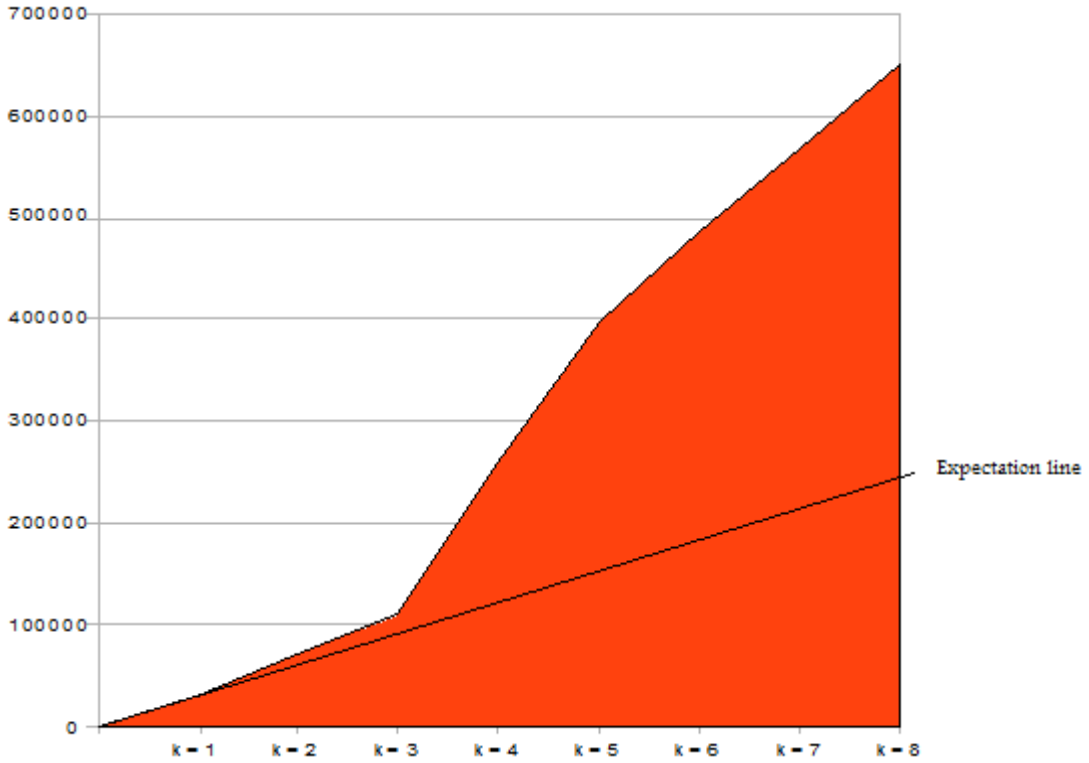
1 Thread. n = 1024. y-axis measured in milliseconds.

In both the 1 and 2 Thread cases, matrix multiplication calculation times fell on or above the "Expectation line" denoting the lower bound of my hypothesized calculation time requirement. Thus, performance was consistent with the initial hypothesis.

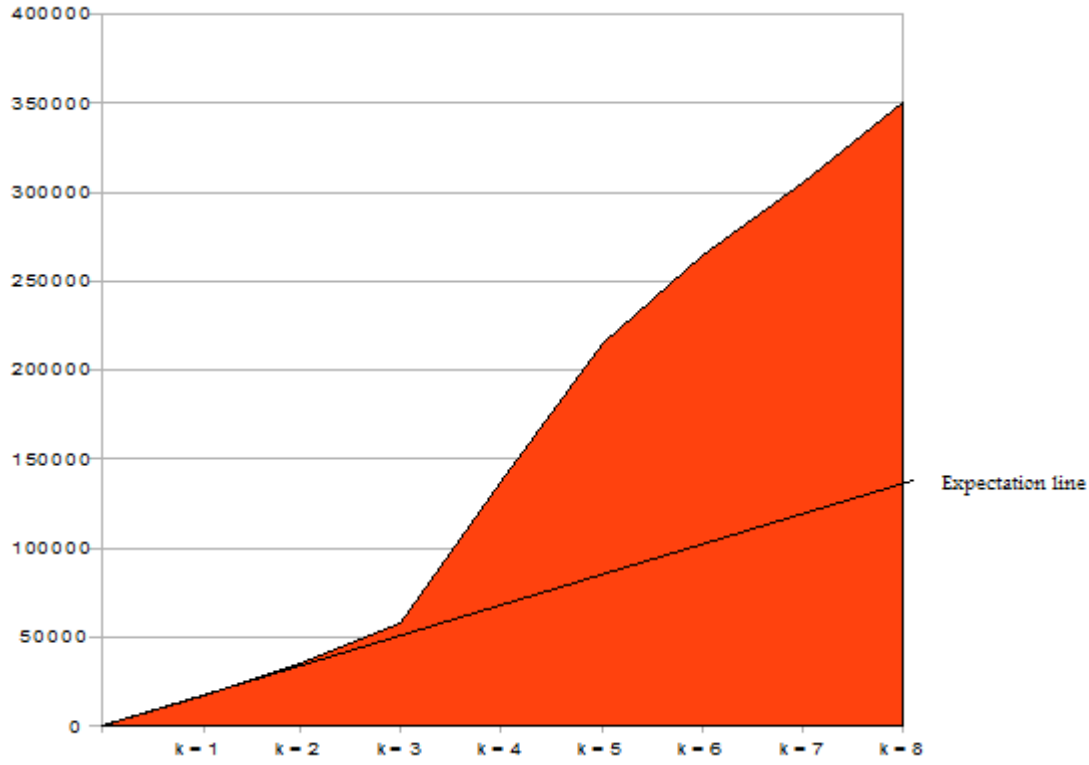
I found it interesting that, in general, use of two Threads nearly doubled the performance of ParallelMatrixMultiply on the Atom versus a single Thread. Since the Atom uses HyperThreading rather than an actual second core, I had expected a significantly smaller performance gain.

A drop in calculation rate becomes apparent in both cases when k is greater than 1. When k = 1, the memory footprint of a row read in matrix A, a column read in matrix B, and a row being written to in matrix C (the resulting matrix) would be a little over 32 KB, fitting into the Atom's 56 K L1 capacity. However, when k = 2, this footprint ends up exceeding 64 KB, thus resulting in the a Thread's calculation of an element in matrix C to exceed the capacity of the Atom's L1 cache, requiring use of slower L2 cache.

5.1.2) AMD Turion 64 X2 TL-60



1 Thread. n = 1024. y-axis measured in milliseconds.



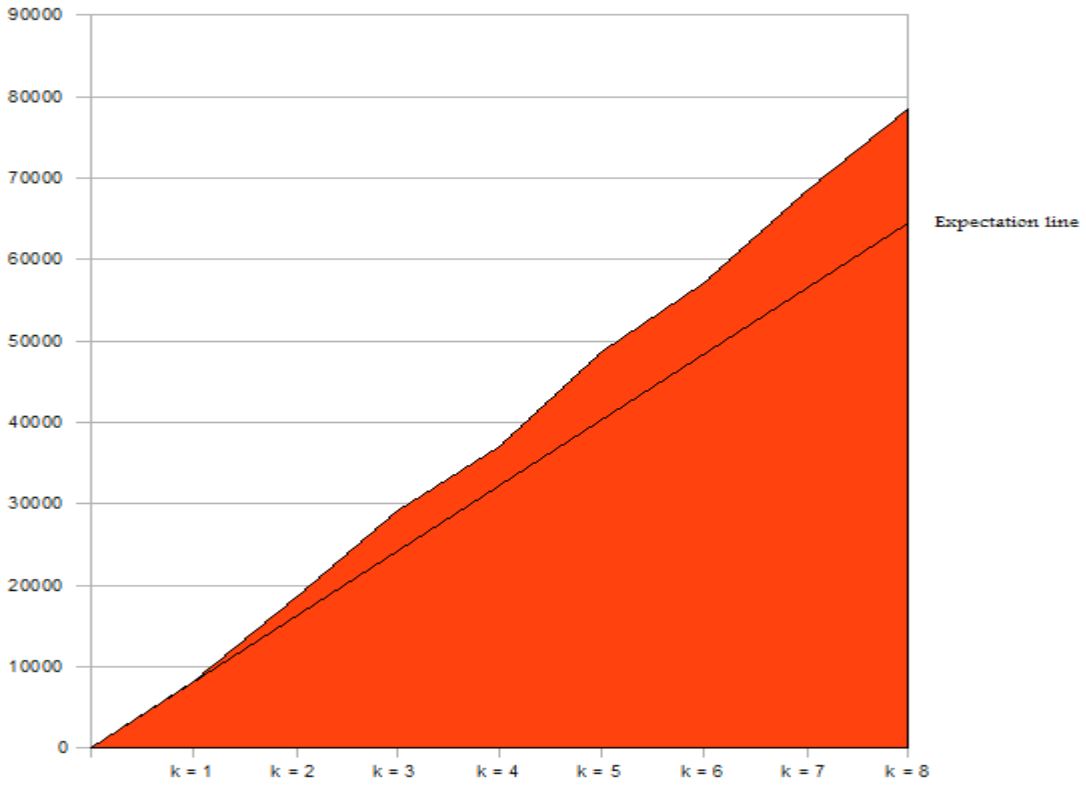
2 Threads. n = 1024. y-axis measured in milliseconds.

In both the 1 and 2 Thread cases, matrix multiplication calculation times fell on or above the "Expectation line." Thus, performance was consistent with the initial hypothesis.

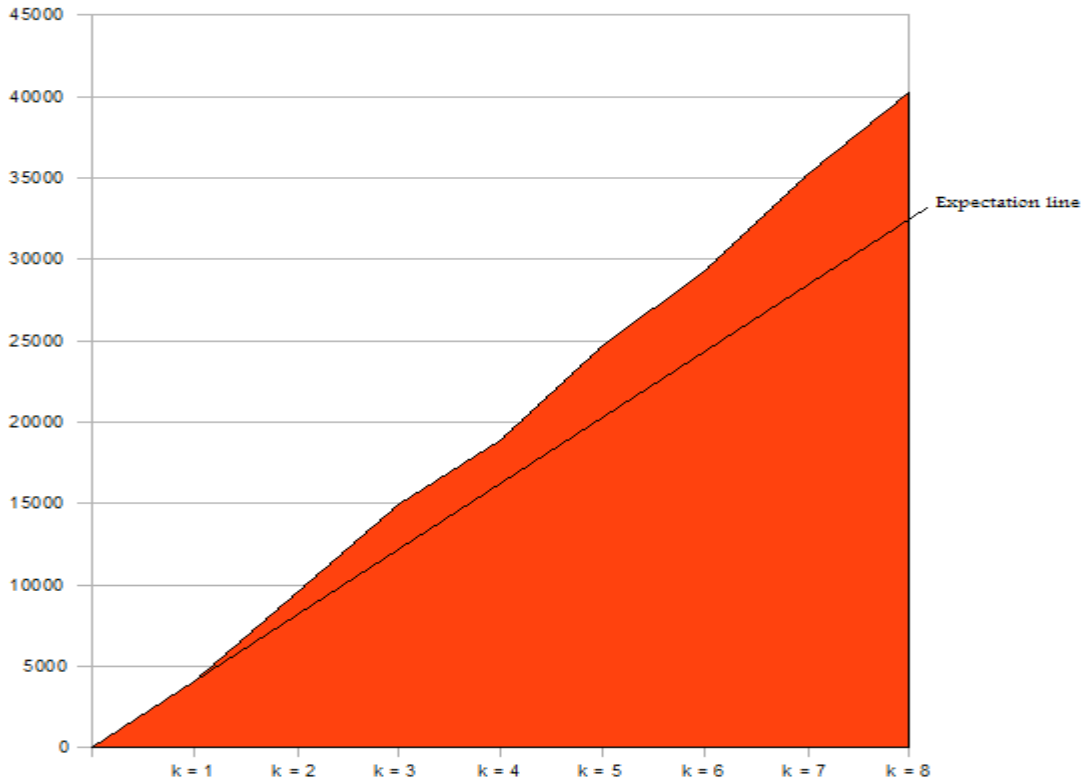
Additionally, utilizing 2 Threads on the dual-core Turion processor resulted in a near doubling of performance (halving of computation time) for all tested values of k, compared to results measured from restricting use to a single Thread.

A drop in calculation rate becomes apparent in both cases when k is greater than 3. This makes sense, since when k = 3, the memory footprint of a row read in matrix A, a column read in matrix B, and a row being written to in matrix C (the resulting matrix) would be a little over 96 KB, where as when k = 4, this footprint ends up exceeding 128 KB, thus resulting in the a Thread's calculation of an element in matrix C to exceed the capacity of the Turion's L1 cache, requiring use of slower L2 cache.

5.1.3) Intel Core2 Quad Q6700



1 Thread. n = 1024. y-axis measured in milliseconds.



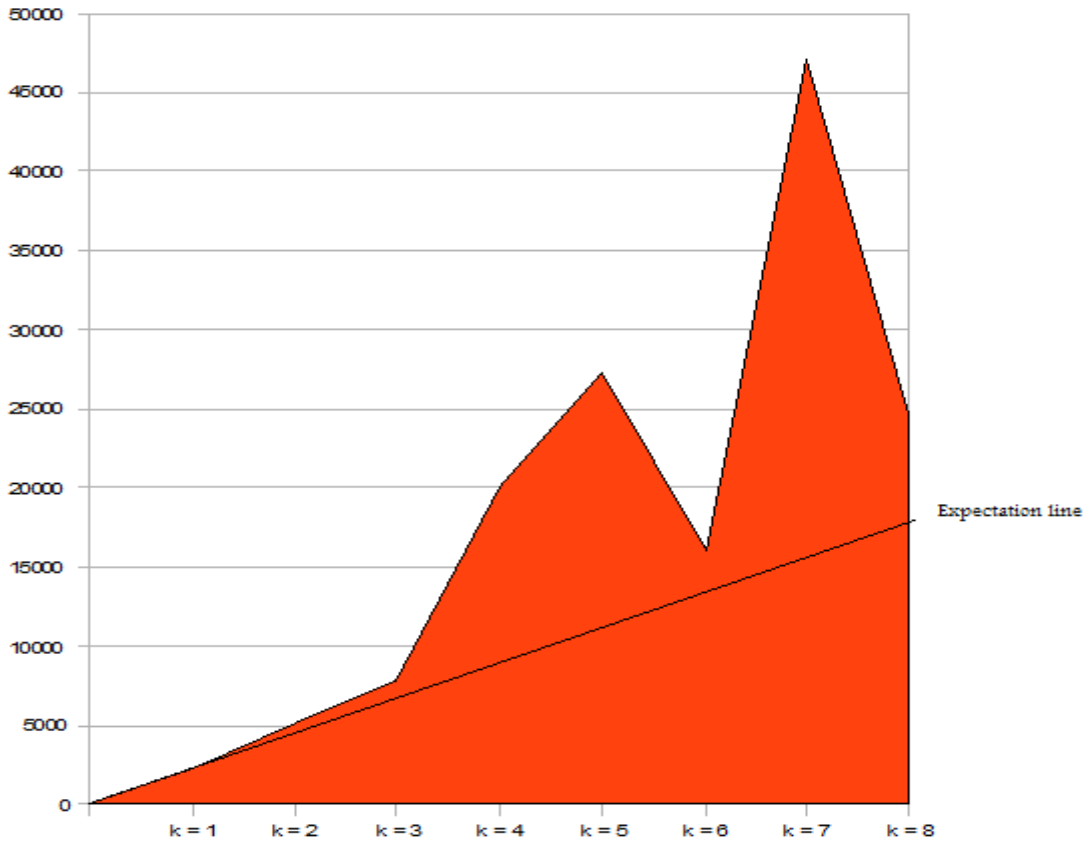
2 Threads. n = 1024. y-axis measured in milliseconds.

Again, in both the 1 and 2 Thread cases, matrix multiplication calculation times fell on or above the "Expectation line," thus falling into my hypothetical expectation.

Additionally, utilizing 2 Threads on the quad-core Intel processor resulted in a near doubling of performance (halving of computation time) for all tested values of k, compared to results measured from restricting use to a single Thread.

Not surprisingly, the Q6700 server-class processor greatly outperforms the Turion mobile-class processor. Given the Q6700's higher clock frequency and power consumption, this is to be expected.

What did surprise me was the near straight-line performance of the Q6700 at all tested values of k. It appears that the transition from using strictly L1 to using L2 cache did not cause significant calculation rate slowdown, the way it did for the Turion processor when k was equal to 4 or higher. This may be due to the Q6700 having faster L2 cache and a wider front-side bus, allowing quicker access to L2 memory.



4 Threads. n = 1024. y-axis measured in milliseconds.

The case where 4 Threads were used on the quad-core Intel processor was interesting in that, while results did end up falling above the Expectation line in accordance with the hypothesis, spiking patterns emerged, as shown in the chart above at k values of 4, 5, and 7.

This is probably attributed to the fact that, at k values higher than 3, the memory footprint exceeds a core's L1 cache capacity, requiring access to the L2 shared cache. While the Q6700 chip is a quad-core, it may be more accurate to describe the chip as being a set of two "dual-cores chips". Each "dual-core chip" shares 4 MB of L2 cache between its two cores, dynamically allocating some quantity of the L2 cache to each core. This dynamic allocation of cache entails some amount of overhead.

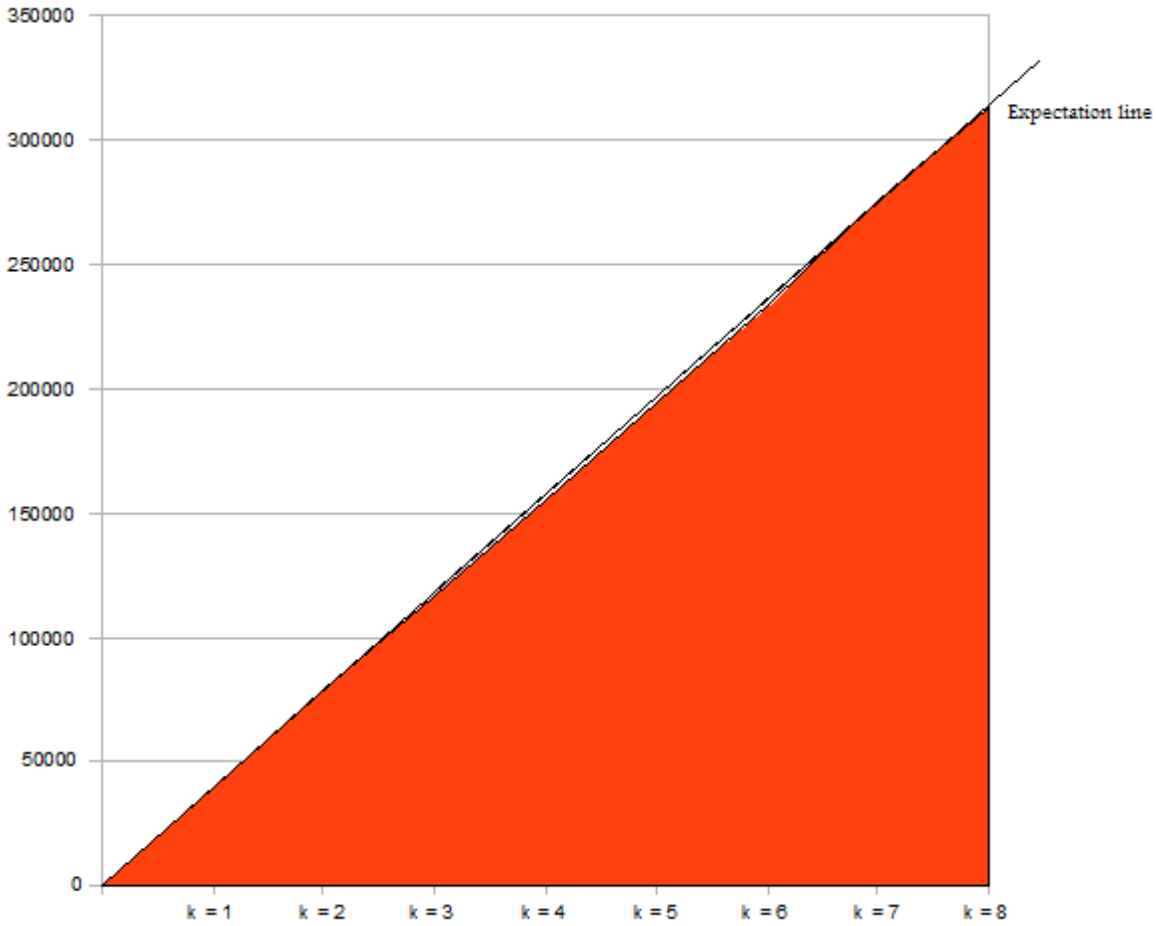
In the case of running ParallelMatrixMultiply restricted to 1 Thread, only one core from one of the effectively two "dual-core chips" is being taxed. Since no major task is being asked of the second core on said "dual-core chip," there is no need to re-allocate L2 cache resources. All L2 cache resources can be devoted to the single core in active use. No sharing of L2 cache is required.

In the case of 2 Threads, one Thread is likely being run on one "dual-core chip", while the other Thread is being run on the other "dual-core chip." As such, the core in use on each "dual-core chip" can be given that chip's entire allotment of L2 cache, without concern for dynamic allocation overhead. Again, no sharing of L2 cache is required.

However, in the case of 4 Threads, all cores are being utilized, and as such, each core will be contending for L2 cache resources. As such, L2 cache sharing occurs, adding that mechanism's overhead into the calculation times.

5.2.) RecursiveParallelMatrixMultiply

5.1.1) Intel Atom N270



n = 1024. y-axis measured in milliseconds.

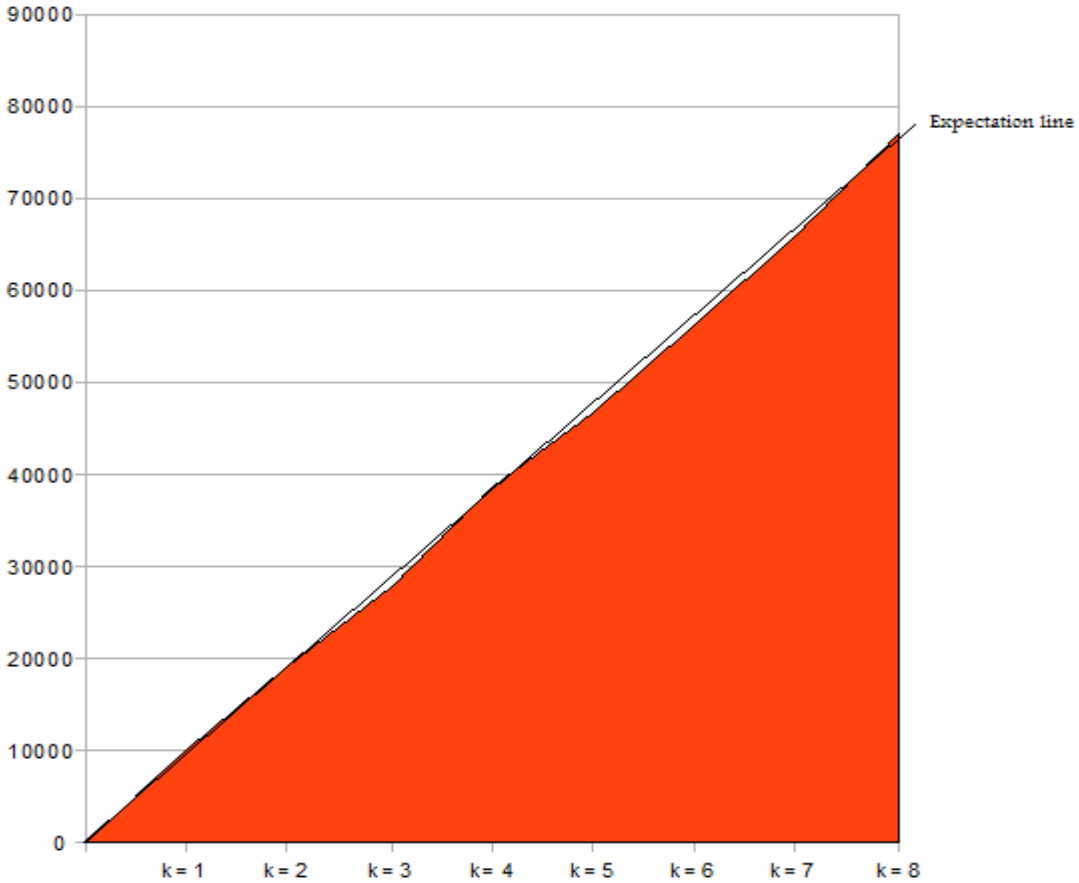
Performance for all values of k held tightly to the Expectation line. Since each matrix multiplication duration value used in the table is an average of five experimentally observed duration values, it is

possible that the slightly better than Expectation line results seen when k = 4, 5, and 6 are due to some margin of error, rather than actually performing better than what my hypothesized limit.

In comparison to the ParallelMatrixMultiply code, RecursiveParallelMatrixMultiply ran about 1 to 3 times faster.

The near-linear tightness of the results with the Expectation line is likely due to the matrices being divided into smaller sub-matrices, such that the smaller columns and rows being accessed in the sub-matrices during the ordinary matrix multiply operation (once the sub-matrices' dimensions are less than or equal to the threshold) fit within the Atom's dedicated L1 cache. Hence, accessing slower L2 cache is avoided.

5.1.2) AMD Turion 64 X2 TL-60



n = 1024. y-axis measured in milliseconds.

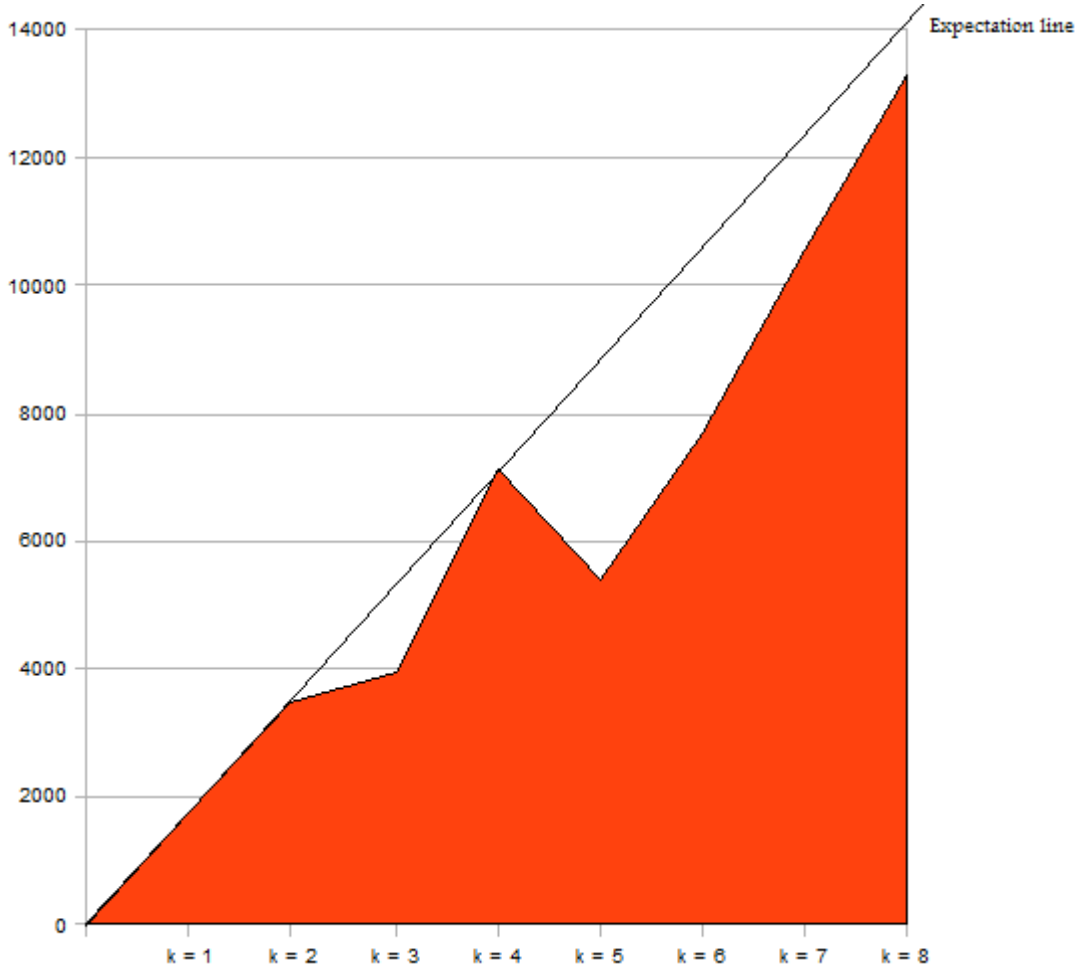
Performance for all values of k held tightly to the Expectation line. Similar to the situation with the Intel Atom N270 processor, the slightly better than Expectation line results seen when k = 3, 5, 6 and 7 are probably due to some margin of error, rather than actually

performing better than what my hypothesized limit.

In comparison to the ParallelMatrixMultiply code, RecursiveParallelMatrixMultiply ran about 1.5 to 3.5 times faster.

As with the Atom processor, the near-linear tightness of the results with the Expectation line is likely due to the matrices being divided into smaller sub-matrices, such that accessing slower L2 cache is avoided.

5.1.3) Intel Core2 Quad Q6700



n = 1024. y-axis measured in milliseconds.

The case of running RecursiveParallelMatrixMultiply on the Q6700 is particularly interesting, in that we finally got results where performance times fell below the Expectation line (when k = 3, 5, 6, 7 and 8).

As of the time of this writing, I have not been able to account for the better-than-hypothesis results. Considering the results from the

Atom and Turion testing, the Q6700's L1 cache is large enough such that I cannot say that the exhibited performance on the Q6700 is due to the shared L2 cache.

In comparison to the ParallelMatrixMultiply code, the Q6700 ran the RecursiveParallelMatrixMultiply code about 0.5 to 4 times faster.

6.) Conclusions

It was surprising to see that my hypothesis failed to account for the case where RecursiveParallelMatrixMultiply ran on the Intel Core2 Quad 6700. While I am was unable to determine the cause for this test case's results, I believe that this would make an interesting topic of further research. Given more time and access to additional equipment, I think it would be worthwhile to test on other processor configurations, including those with various denominations of cores, those with shared caching, and combinations of these two parameters. I would be particularly interested to see test results from an Intel Core i7 processor, which combines 4 physical cores with HyperThreading, allowing for 8 threads to be run simultaneously.

Based on the testing data gathered from three different classes of CPUs (low-power netbook, mid-ranged mobile, and server-class processors) and two different matrix multiplication programs, it is clear that the choice of algorithmic approaches and hardware, particularly in regards to cache, can have a large impact on how quickly computationally intense calculations can be completed.

On the dual-core (or HyperThreading-equivalent) processors, in situations where both the ordinary parallel matrix multiply and recursive parallel matrix multiply programs were largely made use of just L1 cache, the recursive version operated roughly twice as fast as its ordinary counterpart on the same hardware.

Obviously, having access to higher clock frequency, larger cache CPUs can improve computation performance. But, to achieve the best performance out of large-scale calculation tasks, algorithms that can minimize (or avoid all together) their dependence on slower level caches should be utilized. Accessing slower L2 cache can significantly reduce computational performance, easily doubling the time it takes work to complete. As such, I can see a demand for supercomputing applications that checks the hardware they are run on, and adjust their algorithmic approaches accordingly, in order to minimize runtimes, hence improving overall productivity.