Loren McGinnis
PA2

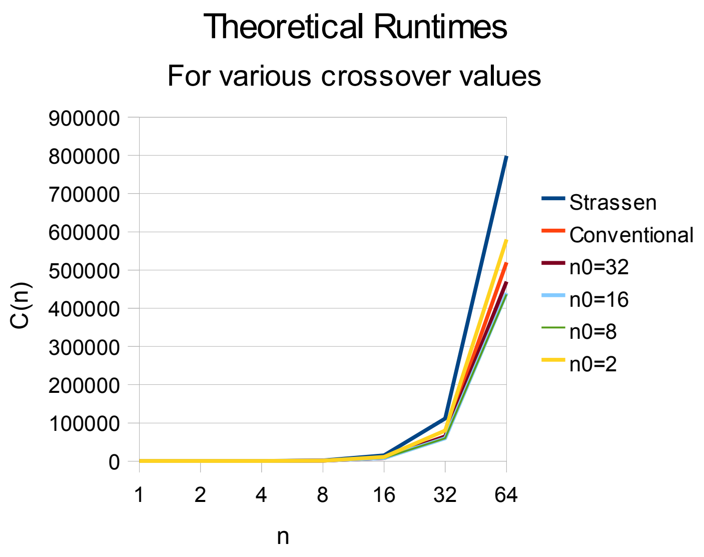
Matrix Multiplication-Finding the Crossover Point

Introduction

From a theoretical, asymptotic standpoint, it is clear that Strassen's algorithm for multiplying matrices runs faster than the conventional algorithm. In, practice, however, I soon realized that Strassen's does a lot of overhead work—adding and multiplying matrices together numerous times at each iteration—such that running it on smaller size matrices is inefficient, and using the conventional algorithm is actually much faster. I also discovered some other factors related to memory management that affected the comparative run-times of each algorithm. In particular, Strassen's algorithm requires allocating more space to store intermediate data. Additionally, the effects of caching, and how it directly relates to the representation of the matrices in memory, can either significantly improve or impair an algorithm's efficacy. Keeping these variables in mind, I attempted to implement the most efficient versions of both algorithms, and hybridized them for further improvement. In the hybrid algorithm, I ran Strassen's on the initial input, until the dimensions of the sub-matrices were less than or equal to a certain number—the crossover point—at which I ran the conventional algorithm. Find the optimal crossover point was a matter of experimentation, but the result was consistent across varying sizes of input.

Approach

I implemented all code for this assignment in C++. I used the clock() method to benchmark run-times, and did all performance testing on the NICE servers. In order for the data to be reliable, I was careful to run my program when no one else was fighting for system resources—the top command was very handy here—and the results seemed to turn out well. During testing, I also noticed that different forms of input (e.g. 0s and 1s vs -1s, 0s, and 1s) had no effect on runtime (although using floats/doubles made everything a constant factor slower), since the CPU executes arithmetic operations on a given data type in the same amount of time, regardless of the data values. Thus, I extend my analysis to all of these inputs and treat them identically.



Theoretical Analysis

Before actually coding the problem, I determined a theoretical value for the crossover point, based on the assumptions that all arithmetic operations had a computation cost of 1, and everything else was free. From this, the cost functions S(n) for Strassen's and C(n) for the conventional running on a matrix of size n by n, were the following:

$$C(n)=n^3-n^2 \quad S(n)=7S(n/2)+4.5n^2$$

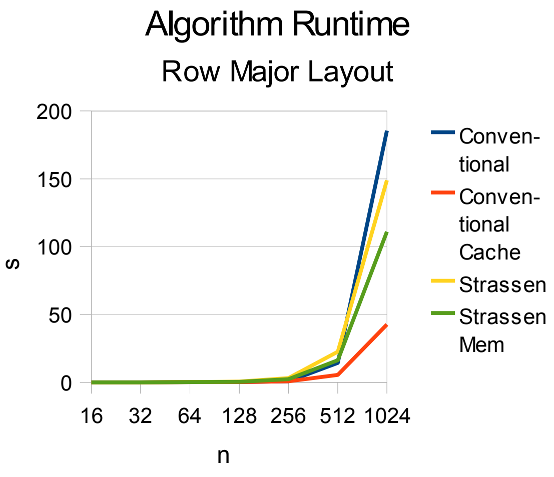
Using a spreadsheet, I calculated the cost for powers of 2, then, to simulate the crossover point at some value n_0, replaced $S(n_0)$ with $C(n_0)$. This was the result of graphing the data. The cost bottomed out between $n_0 = 8$ and $n_0 = 16$. In a more formulaic approach, assuming n_0 is the crossover point, we know that, if we're running Strassen's on a matrix of size n_0 , the cost of running conventional will equal the cost of running one more step of Strassen's, or:

$$C(n_0) = H(n_0)$$
$$n_0^3 - n_0^2 = 7C(n_0/2) + 4.5n_0^2 = 1.75n_0^3 + 2.75n_0^2$$
$$n_0 = 15$$

Implementation

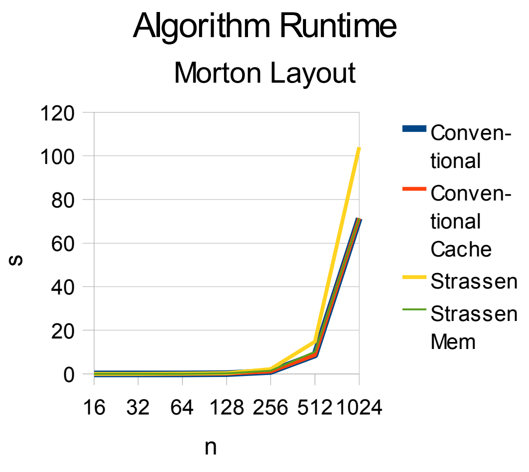
When I first began coding, I allocated n^2 space for three matrices, and used row-major format for the data, basically implementing my own pointer arithmetic to dereference an element by row and column. Even though I tried different approaches to data layout, row-major proved to be most efficient in the final hybrid algorithm. I wrote two versions of the conventional algorithm. The first was cache-ignorant, and calculated each element of the result matrix entirely before moving on to the next, similar to how one would do it by hand. The second implementation ran much faster, because it traversed all the matrices in a row-by-row order, exhibiting the spatial locality for cache-efficiency.

For Strassen's, I also started with a naïve implementation and improved upon it. While I was never copying data unnecessarily—my matrix implementation was smart about using the same data in memory for the sub-matrices—I was not careful about allocation, and called malloc() several times at each recursive step for the seven intermediate matrices required by the algorithm. Using clever ordering of operations, however, I was able in my efficient version of Strassen's to avoid allocating any memory at all except an initial allocation of $4n^2$ space. $3n^2$ was for the input and output matrices, and the fourth matrix became a placeholder for temporary data. Running the conventional algorithm with caching in mind provided the best improvement for values larger than 256 (there was almost no effect on smaller values, because the matrices all fit into the cache). The memory-efficient version of Strassen's was about 25% faster than the naïve version. The constant factor seemed a little surprising, and hard to explain, given the indeterminate behavior of malloc(), but the amount of total space that I allocated in the naïve approach was also a within a constant factor of the more efficient version.

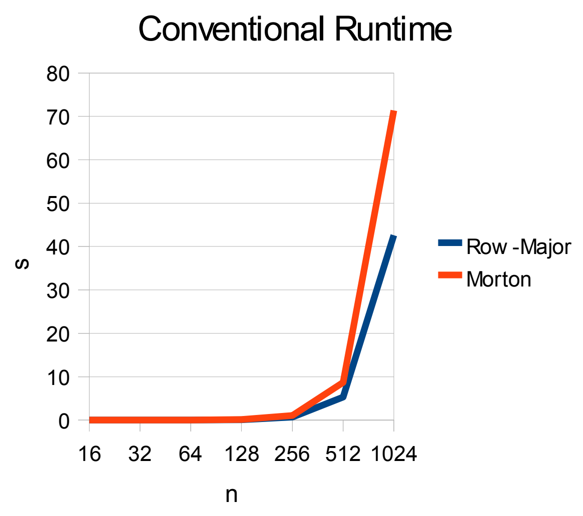


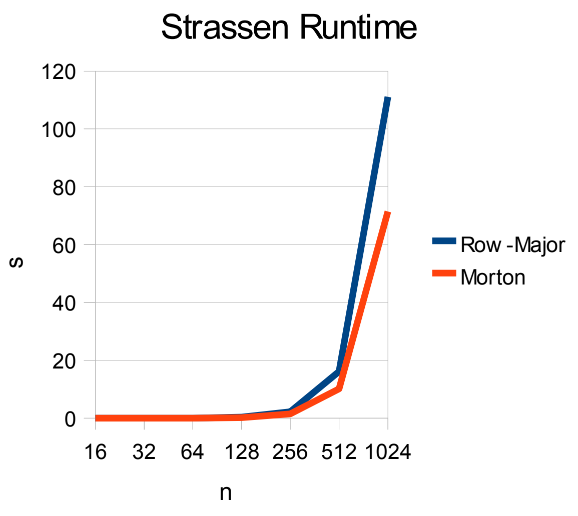
Morton representation

After seeing the dramatic and evident benefits of being cache-aware with the conventional algorithm, I attempted to make Strassen's more cache efficient by changing the layout of the data. In a Morton representation, the data is represented in such a way that for any given sub-matrix, the data remains contiguous. The issue with a row-major layout in Strassen's algorithm is that for very large initial matrices, the rows of a small matrix are very far apart, and in manipulated sub-matrices at lower levels of recursion causes a loss of spatial locality. A Morton representation



maintains spatial locality for small matrices, an ideal situation for Strassen's. Of course, the trade-off is that the conventional method loses locality, as the results show.

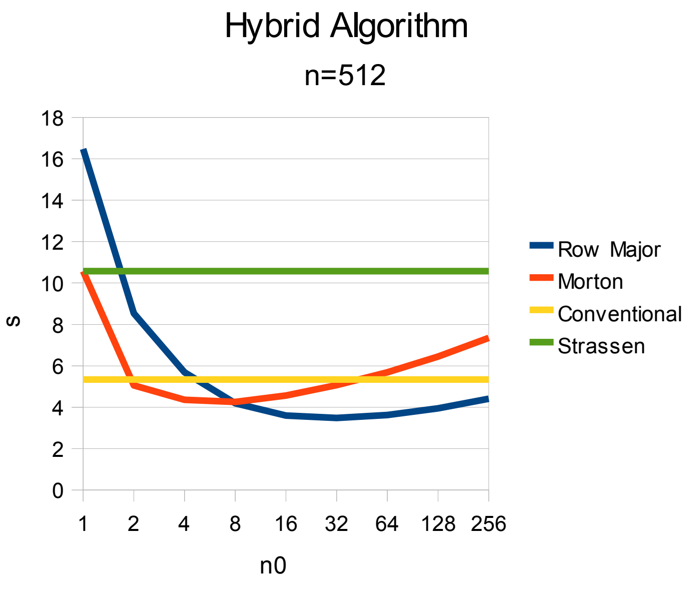


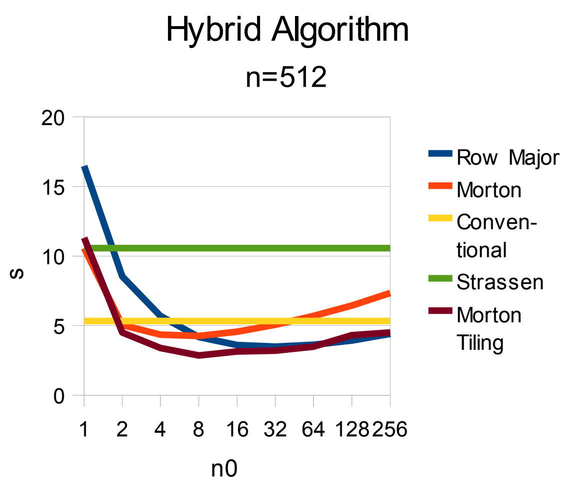


The Morton layout helped Strassen's about as much as it hurt the conventional (although, the naïve conventional approach improved dramatically—the very slow results from row-major are because it runs down the columns of one matrix, causing a cache miss every time; Morton mitigates this). However, Strassen's algorithm was still slower than row-major conventional-cache.

Finding the Crossover, Experimentally

After optimizing the two algorithms separately, I implemented a hybrid algorithm, that ran Strassen's until the dimension of the input matrices were smaller than a given value. The given value that minimizes runtime is the crossover point. Keeping the input size constant, I tested different crossover values, for both the Morton and row-major layouts. The fastest runtime occurred at n_0=32 with a row-major layout. I was disappointed that the Morton layout did not perform as well (although it did beat the fastest conventional time, and had a lower optimal n_0). This is probably because the conventional algorithm was optimized for row-major layout, and even though there were wide gaps between each row in the sub-matrices, it remained more efficient than with Morton. However, for very small matrices, there were many more matrix multiplications, and each multiplication was much simpler, so the overhead time from cache-misses was significant enough to make the Morton layout faster. To verify my data, I ran the same tests for larger values of n, and the graphs came out identically (albeit with a different scale on the y-axis), so the crossover point is unaffected by data size.

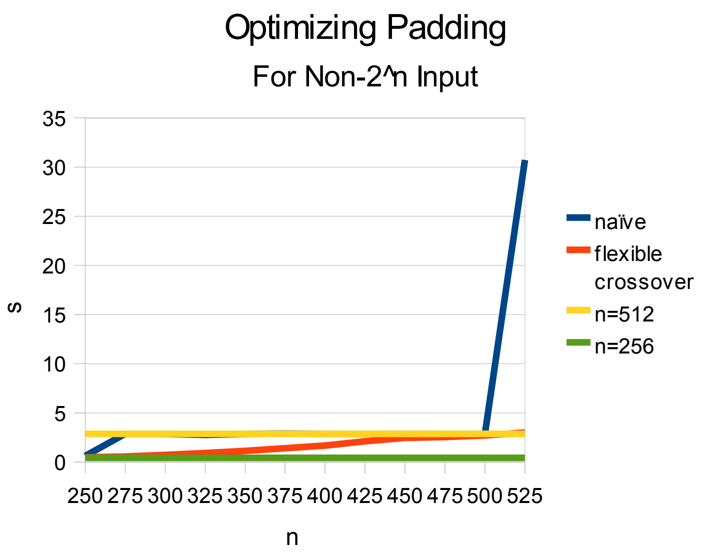




In order to get the best of both worlds, so that the Strassen's could benefit from the Morton layout, and the conventional could benefit from row-major for smaller matrices, I extended the same idea of hybridization to the layout of the data. I implemented a new layout, which was in Morton form down to the crossover point, at which the data was inserted into row-major order. The results of this were even better, and gave me a final n_0=8, as seen above.

Optimizing For Intermediate Dimensions

All of the previous tests have been done with inputs sizes of 2^n, since this makes the implementation simpler. To be able to multiply matrices of arbitrary dimension n, I needed to pad the input so that it could still be divided evenly by the algorithm. An easy way to do this is just pad one extra layer of zeros if n is odd. However, in the worst case, only one recursion would be possible, and the algorithm would be forced to make n/2 the crossover point. On the other hand, we could ensure that the matrix will recurse as far as needed by padding it to the next highest power of two. This will allow the algorithm to crossover at the desired spot.



The worse case, here, though, is when n is slightly more than a power of two. As the graph shows, when crossing n=256, there is a big jump in runtime, where the input is padded to 512, and thus takes just as long to compute. Similarly, just after 512 the runtime jumps to that of 1024. To improve this, I implemented a more flexible approach, which finds some value n_1 near n_0 such that

$$d = n_1 2^k - n > 0$$

and d is minimized. Considering the small runtime difference between n_0=16, n_0=32, and n_0=64, for row-major layout, picking another crossover in this range seemed reasonable. The input is padded to $n_1 2^k$, guaranteeing that the algorithm will recurse down to n_1, and n_1 becomes the new crossover point. The results of this optimization show the runtime growing more smoothly between powers of two.