Strassen’s Matrix Multiplication Algorithm in Cilk++

Bradley C. Kuszmaul
Massachusetts Institute of Technology
Computer Science and Artificial Intelligence Laboratory
(MIT CSAIL)
32 Vassar St, 32-g766; Cambridge, MA 02139
bradley@mit.edu

September 11, 2009

Abstract

For the first challenge (Matrix Multiplication using Strassen’s Algorithm) of Phase 2 of the 2009 Intel Threading Challenge I implemented Strassen’s algorithm in Cilk++. I built versions that use both GotoBLAS and MKL to implement the base case of the recursion. I measured an effective performance of 102.5 GFLOPS on a $12000 \times 12000$ matrix multiply on a dual-socket Intel Xeon 2.53GHz E5540 (8 cores total, 16 HT (SMT) threads total) with 12GiB RAM.

1 Introduction

The 2009 Intel Threading Challenge challenges coders to implement solutions to a sequence of problems. The challenge is organized in two phases, one starting in the late spring to early summer and one in late summer into fall. The first problem in this year’s second Phase is to perform matrix multiplication using Strassen’s algorithm. I implemented my solution in Cilk++, developed by Cilk Arts. Cilk Arts was recently acquired by Intel.

Cilk++ introduces a few new keywords to the C++ language:

• `cilk_spawn` indicates that a function can be called in parallel to the caller.
• `cilk_sync` indicates that a thread must wait for all its spawned children to complete.
• `cilk_for` can be used in place of `for` to build a parallel loop. The `cilk_for` syntax can be viewed as syntactic sugar for a divide-and-conquer recursive program.

A Cilk++ program on one core typically runs within a few percent of the speed of the corresponding serial C++ program.

The Cilk++ system also includes a race-detection and profiling tool called Cilkscreen. Given a program and an input, the race detector finds all the data races that can occur in any possible execution of the program. The profiling tool measures the work (total number of instructions executed) and the span (total number of instructions on the critical path of the computation), and calculates the parallelism (work divided by span) and estimates the speedup on machines with various numbers of cores.

My approach when implementing a Cilk++ program is to

1. write the fastest correct serial program I can,
2. introduce Cilk++ keywords to make the program parallel,
3. use the Cilkscreen race detector to find and correct any data races, and
4. use the Cilkscreen performance profiler to predict performance, looking for ways to reduce the span to increase the parallelism.

---

Since Cilk++ allows me to factor the problem into the steps of implementing serial correctness, implementing serial performance, implementing parallel correctness, and implementing parallel performance, it can be fairly straightforward to write high-performance parallel programs.

The rest of this paper is organized as follows. Section 2 describes the problem. Section 3 describes my implementation and performance. Section 4 explains how to use my code. I submitted both source and executable code, and Section 5 shows how to rebuild the executables from source.

2 Problem Description

The problem, as stated on Intel’s web site, is as follows.

Problem: Write a threaded code to multiply two random matrices using Strassen’s Algorithm. The application will generate two matrices $A(M, P)$ and $B(P, N)$, multiply them together using

1. a sequential method\(^3\) and then
2. via Strassen’s Algorithm resulting in $C(M, N)$.

The application should then compare the results of the two multiplications to ensure that the Strassen’s results match the sequential computations.

The input to the application comes from the command line. The input will be 3 integers describing the sizes of the matrices to be used: $M$, $N$, and $P$.

Code restrictions: A very simple, serial version of the application, written in C, will be available here.\(^4\) This source file should be used as a starting point. Your entry should keep the body of the main function, the matrix generation function, the sequential multiplication code, and the function to compare the two matrix product results. Changes needed for implementation in a different language are permitted. You are also allowed to change the memory allocation and other code to thread the Strassen’s computations. (It would be a good idea to document the changes and the reason for such changes in your solution write-up.) After the needed changes, your submitted solution must use some form of Strassen’s Algorithm to compute the second matrix multiplication result.

Timing: The time for execution of the Strassen’s Algorithm will be used for scoring. Each submission should include timing code to measure and print this time to `stdout`. If not, the total execution time will be used.

3 Implementation and Performance

I measured the performance of the following codes.

1. The simple triply-nested loop ("3-loop") implementation provided by the contest judges. I measured both the original serial version and version parallelized with Cilk ("Cilkified"). This code uses the "sequential" or "cubic" algorithm.
2. The recursive Strassen code provided by the judges. The base case of the recursion for this code is a 3-loop. I measured both serial and Cilkified versions of this code.
3. The Intel Math Kernel Library (MKL) implementation of DGEMM (both single threaded and multithreaded).
4. The GotoBLAS implementation of DGEMM (both single threaded and multithreaded)
5. A recursive divide-and-conquer (D&C) implementation (both serial and Cilkified). This implementation is the same number of floating point operations as the 3-loop. I built three versions, variously using
   1. a triply nested loop as the base case of the recursion,
   2. single-threaded MKL as the base case, and
   3. single-threaded GotoBLAS as the base case.

\(^3\)A more careful statement of the problem might use “cubic method” instead of “sequential method”, since the cubic method can be run in parallel, and Strassen can be run serially.
\(^4\)From the contest judges.
4. An implementation of recursive Strassen (both serial and Cilkified) that I wrote. As for D&C I built three versions, variously using
   1. a triply nested loop as the base case,
   2. single-threaded MKL as the base case, and
   3. single-threaded GotoBLAS as the base case.

The floating-point performance for these various algorithms for a variety of matrix sizes is shown in Figure 1. Rather than displaying the run time, which is naturally shorter for small matrices, and longer for large matrices, I am displaying the rate in billions of floating point operations per second (GFLOPS). The cubic algorithm uses $NMP$ floating point additions and $NMP$ additions. (By being a little careful one could get that down to $NM(P-1)$ additions.) If the matrices are square, i.e., $N = M = P$, then the number of floating point operations is $2N^3$. For all of the following measurements I used square matrices.

For all of these measurements, the floating point performance is calculated by assuming that there are $2n^3$ floating point operations. For Strassen, which actually computes fewer floating point operations, that can yield an “effective floating point performance” which is faster than the peak performance of the hardware.

There is quite a bit of noise in these measurements. If I measure the same program twice in a row, there can easily be a 10% variation in performance, or more. I measured each data point at least 3 times, and took the fastest speed I saw. Some of the speed differences of Figure 1 likely to be noise. I measured GotoBLAS and MKL many times, however, and it appears that those differences are real, however.

All measurements were made on Dell PowerEdge T410, which comprises a dual-socket Intel Xeon 2.53GHz E5540 with 8 cores total and 16 HT (SMT) threads total. The machine has 12GiB (6 x 2GiB) of 1333MHz Dual Ranked RDIMMs memory. Each socket has 8MiB cache. The system is running Ubuntu 9.04 running Linux kernel 2.6.29-TPS+perfctr.\(^5\)

I measured elapsed time using gettimeofday. The gettimeofday has 1\(\mu\)s precision and about 1\(\mu\)s accuracy, and costs about 1\(\mu\)s to call. Thus any time measurements of less than, say, ten microseconds should be taken with a grain of salt. The judges’ reference implementation uses clock(), which, on Linux, returns the CPU time used rather than the elapsed time, which is not what the problem description seems to call for (total execution time).

\(^5\)Our machine’s kernel uses perfctr by Mikael Pettersson http://user.it.uu.se/~mikpe/linux/perfctr/. Our kernel also includes Silas Boyd-Wickizer’s TPS patches for providing different memory maps for different threads, but my codes don’t take advantage of that OS feature, so it is probably irrelevant to this contest.
I also modified the top-level loop to
1. do better error checking when parsing the arguments;
2. print effective megaflops as well as elapsed time;
3. use the same the clock and performance measurement code (rather than repeating the code it for different measurements);
4. use CRLF instead of LF for end-of-line characters (I’m on Linux, not Windows);
5. include Akki’s corrections to initialize the output matrix $C$ to zero properly;
6. use $\text{fabs}()$ instead of $\text{abs}()$ to compare the difference to a threshold;
7. to compute the base case properly (if any of the dimensions hit 1 while $MNP > \text{GRAIN}$ then the recursion stops (previously invoking the reference code with arguments such as $N = 1$, $M = 1024$, and $P = 1024$ would cause a segfault);
8. to compute the product of dimensions $MNP$ in 64-bit to prevent overflow when handling matrices bigger than 2048 on a side; and
9. set the grain size to 512\(^3\) (which is much larger than in the original code.)

All of the Cilk versions were run against Cilkscreen and no data races were detected.

The rest of this section describes each of the codes, how I Cilkified it, and its performance.

### 3.1 Reference 3-loop code

For the reference 3-loop, I measured both serial and Cilkified versions of this code. I measured this for both gcc and the Intel cc compiler (icc), but I only present gcc information, since the icc didn’t seem to do much better.

As Figure 1 shows, the performance mostly below 1 GFLOPS (billion floating point operations per second). The Cilk version got some speedup for both versions of the code, but for large matrices, the performance fell to a fraction of a GFLOPS.

In my submission, the file reference/StrassenMMmult.cpp is the serial version of the reference code, which includes both 3-loop and Strassen code. File reference/StrassenMMmultcilk.cilk is the Cilkified version.

To Cilkify the 3-loop code I simply parallelized the outer loop.

```c
    cilk_for (int i = 0; i < m; i++)
        for (int j = 0; j < n; j++)
            {
                C[i][j] = 0.0;
                for (int k = 0; k < p; k++)
                    C[i][j] += A[i][k]*B[k][j];
            }
```

### 3.2 Reference Strassen

For the reference Strassen code, I also measured both serial and Cilkified versions. As Figure 1 shows, the Strassen code generally did better than the 3-loop code, but still only got up to 3.6 GFLOPS.

The basic Strassen algorithm as implemented by the judges is shown in Figure 2. To Cilkify the reference Strassen code, I did the following:

1. Move all the pre-multiplication additions to the front, and spawn them all.
2. Put a $\text{cilk\_sync}$ after those additions.
3. Spawn all the recursive calls.
4. Put a $\text{cilk\_sync}$ after the recursive calls.
5. Cilkify the outermost loop of the final additions.

resulting in the algorithm shown in Figure 3. This code has a lot of parallelism.
\begin{align*}
S_1 &= A_{11} + A_{22} \\
S_2 &= B_{11} + B_{22} \\
P_1 &= S_1 \cdot S_2 \\
S_3 &= A_{21} + A_{22} \\
P_2 &= S_3 \cdot B_{11} \\
S_4 &= B_{12} - B_{22} \\
P_3 &= A_{11} \cdot S_4 \\
S_5 &= B_{21} - B_{11} \\
P_4 &= A_{22} \cdot S_5 \\
S_6 &= A_{11} + A_{12} \\
P_5 &= S_6 \cdot B_{22} \\
S_7 &= A_{21} - A_{11} \\
P_6 &= S_7 \cdot S_8 \\
S_8 &= B_{11} + B_{12} \\
P_7 &= A_{11} \cdot S_8 \\
S_9 &= A_{12} - A_{22} \\
S_{10} &= B_{21} + B_{22} \\
P_8 &= S_9 \cdot S_{10} \\
C_{11} &= P_1 + P_4 - P_5 + P_7 \\
C_{12} &= P_3 + P_5 \\
C_{21} &= P_2 + P_4 \\
C_{22} &= P_1 - P_2 + P_3 + P_6
\end{align*}

Figure 2: Basic Strassen algorithm, as implemented by the judges.

3.3 MKL

I measured the performance of single-threaded and multithreaded MKL.

For the MKL and all of the following codes, I changed the data layout to be a simple row-major. (The code provided by the judges uses an array of pointers to doubles.) This change is required by the DGEMM interface. Since this is the standard interface for matrix multiplication, it seems reasonable to use it. I did not make an effort to write a highly-optimized base case myself, since MKL and GotoBLAS are both available. (The contest explicitly encourages us to use Intel tools, and I wanted to measure GotoBLAS while I was at it.)

As Figure 1 shows, the threaded MKL code gets 76.3 GFLOPS on a $8192 \times 8192 \times 8192$ matrix multiply, which leaves the reference implementations in the dust. Even on small $256^3$ matrices, MKL gets 10.6 GFLOPS. It’s going to take a little more work to make Strassen competitive.

I used MKL to implement the base case of my recursive Cilk programs, so it’s interesting to notice that single-threaded MKL essentially reaches peak performance for matrices that are at least $512^3$. $256^3$ is just too small.

To use the MKL, I changed the code to handle each matrix as a single array of floats stored in row-major order. I used the standard BLAS3 DGEMM procedure provided by MKL. The arguments to DGEMM are

- $m$;
- $n$;
- $p$;
- $\alpha$, a real number
- $A$, a pointer a $m \times p$ matrix represented a array of doubles;
- $A_N$, the row stride of $A$ (the number of doubles between $A_{i,j}$ and $A_{i+1,j}$);
- $B$, a pointer to $p \times n$ matrix;
- $B_N$, the row stride of $B$;
- $\beta$, a real number;
\begin{verbatim}
cilk_spawn S_1 = A_{11} + A_{22}
cilk_spawn S_2 = B_{11} + B_{22}
cilk_spawn S_3 = A_{21} + A_{22}
cilk_spawn S_4 = B_{12} - B_{22}
cilk_spawn S_5 = B_{21} - B_{11}
cilk_spawn S_6 = A_{11} + A_{12}
cilk_spawn S_7 = A_{21} - A_{11}
cilk_spawn S_8 = B_{11} + B_{12}
cilk_spawn S_9 = A_{12} - A_{22}
cilk_spawn S_{10} = B_{21} + B_{22}
cilk_sync;
cilk_spawn P_1 = S_1 \cdot S_2,
cilk_spawn P_2 = S_3 \cdot B_{11}
cilk_spawn P_3 = A_{11} \cdot S_4,
cilk_spawn P_4 = A_{22} \cdot S_5
cilk_spawn P_5 = S_6 \cdot B_{22}
cilk_spawn P_6 = S_7 \cdot S_8
cilk_spawn P_7 = S_9 \cdot S_{10}
cilk_sync;
cilk_for i, j \text{ (for appropriate values of } i, j) \begin{align*}
(C_{11})_{i,j} &= (P_1)_{i,j} + (P_3)_{i,j} - (P_5)_{i,j} + (P_7)_{i,j} \\
(C_{12})_{i,j} &= (P_3)_{i,j} + (P_5)_{i,j} \\
(C_{21})_{i,j} &= (P_2)_{i,j} + (P_4)_{i,j} \\
(C_{22})_{i,j} &= (P_1)_{i,j} - (P_2)_{i,j} + (P_3)_{i,j} + (P_6)_{i,j}
\end{align*}
\end{verbatim}

Figure 3: Cilkified basic Strassen algorithm.

- \( C \), a pointer to a \( m \times n \) matrix; and
- \( C_N \), the row stride of \( C \).

The DGEMM procedure computes
\[
C \leftarrow \alpha (A \times B) + \beta C.
\]

For simplicity I’m ignoring some of the actual DGEMM parameters that allow one to specify that rows are in column-major order, or that the matrices should be transposed before being operated on.

I encapsulated the BLAS implementation specifics into a separately compiled C code, called `cblas-mkl.c`, `cblas-goto.c`, and `cblas-3loop.c` to implement the MKL interface, the GotoBLAS interface, and the 3loop interface respectively. As a result, the difference between the various Cilk implementations is which cblas binary is linked in.

### 3.4 GotoBLAS

I also tried using GotoBLAS, since I’ve heard that it is competitive with MKL. As Figure 1 shows, single-threaded GotoBLAS is a little faster for small matrices, but still doesn’t really get to peak until about \( 512^3 \), and then is about the same is MKL. The multithreaded GotoBLAS seems to stay significantly ahead of multithreaded MKL up to about \( 8192^3 \), where they are about the same.

Interestingly, in multithreaded operation both MKL and GotoBLAS only use 8 cores on my dual-socket Nehalem system, which has 16 hardware threads. They both play games with processor affinity. When I forced them to run 16 threads on a single DGEMM call, the performance dropped dramatically, so 8 threads seems like a good idea.

Since MKL or GotoBLAS both seem good for a base case. I tried both of them. To compile GotoBLAS for use with Cilk++, I used
which turns off all threading
I found for Cilk codes running only 8 worker threads it helps to set the processor affinity of the threads in a way that none of them end up on the same core. I used taskset to do it. A typical execution would be

```
$ make USE_THREAD=0
```

In contrast for the 3-loop implementation, using all 16 HT threads can help, since the floating point hardware isn’t the bottleneck, and HT threads are good for reducing the impact of cache misses. The 3-loop codes are still slow, however.

### 3.5 Recursive D&C

I built a recursive divide-and-conquer (D&C) program that uses $O(n^3)$ floating point operations. A recursive D&C matrix multiply should get good cache performance, and demonstrates that I can use the DGEMM libraries to get good performance. I built three versions, using a base case of any matrix multiply less than or equal to $512^3$.

1. The base case is 3-loop.
2. The base case is MKL.
3. The base case is GotoBLAS.

As Figure 1 shows, the 3-loop base case is not so good and both MKL and GotoBLAS get pretty good performance.

One interesting aspect of the D&C code with GotoBLAS is that it doesn’t perform well for small matrices. But if I run it twice, the second time (labeled “2nd run”) does much better. It appears that GotoBLAS suffers a startup cost of around 10 milliseconds when used with Cilk++, which must be overcome. For example if I run strassen on a 1024x1024 matrix, it runs very slowly, but if I run it again, it runs much faster. The problem seems to be that a 1024x1024 matrix runs in only 36ms, so the startup cost is causing trouble. On the 2nd run, D&C with GotoBLAS seems to be generally better than D&C with MKL.

The recursive D&C code is organized as shown in Figure 4. The code is organized so that if the matrices are small, it calls DGEMM (which can either be 3-loop, MKL, or GotoBLAS). Otherwise there are 8 recursive calls. If we view the matrices as divided into quadrants we have:

$$
\begin{pmatrix}
C_{11} & C_{12} \\
C_{21} & C_{22}
\end{pmatrix} =
\begin{pmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{pmatrix} \cdot
\begin{pmatrix}
B_{11} & B_{12} \\
B_{21} & B_{22}
\end{pmatrix}.
$$

I implement that recursively as

```
cilk sync;
C_{11} += A_{11} \cdot B_{11};
C_{12} += A_{11} \cdot B_{12};
C_{21} += A_{21} \cdot B_{11};
C_{22} += A_{21} \cdot B_{12};
```

Since the first four write into disjoint locations I can perform them in parallel without a data race. Then I can perform the second four in parallel without a data race. Thus, the D&C code can run in place with no memory.

The code is written using C-style row-major ordered arrays, as shown in Figure 4. If the matrices are small, then at Line 5, the function uses DGEMM. Otherwise, the function performs 8 recursive multiplications. Address calculations are performed using standard row-major array index calculations. For example, if $A$ is a matrix with row stride $P$, then we can calculate the address of $A_{i,j}$ as $A + j + iP$. It turns out that $A + j + iP$ points at the matrix of size $(M - i + 1) \times (P - j + 1)$

$$
A_{i:m,j:p} = 
\begin{bmatrix}
A_{i,j} & \cdots & A_{i,p} \\
\vdots & \ddots & \vdots \\
A_{n,j} & \cdots & A_{n,p}
\end{bmatrix}.
$$
DCMM\((m,n,p,A,B,C,N,P)\)
1 \(\triangleright\) \(A\) is an \(m \times p\) matrix with row stride \(P\).
2 \(\triangleright\) \(B\) is a \(p \times n\) matrix with row stride \(N\).
3 \(\triangleright\) \(C\) is an \(m \times n\) matrix with row stride \(N\).
4 if the matrices are small
5 then \(\text{DGEMM}(m,n,p,1.0.A,P,B,N,1.0.C,N)\);
6 else
7 \(m_2 \leftarrow [m/2] ;\)
8 \(n_2 \leftarrow [n/2] ;\)
9 \(p_2 \leftarrow [p/2] ;\)
10 cilk_spawn \(\text{DCMM}(m_2, n_2, p_2, A, B, C, N, P)\);
11 cilk_spawn \(\text{DCMM}(m_2, n-n_2, p_2, A, B+n_2, C+n_2, N, P)\);
12 cilk_spawn \(\text{DCMM}(m-m_2, n_2, p_2, A+m_2P, B, C+m_2N, N, P)\);
13 cilk_spawn \(\text{DCMM}(m-m_2, n-n_2, p_2, A+m_2P, B+n_2, C+n_2+m_2N, N, P)\);
14 cilk_sync;
15 cilk_spawn \(\text{DCMM}(m_2, n_2, p-p_2, A+p_2, B+p_2N, C, N, P)\);
16 cilk_spawn \(\text{DCMM}(m_2, n-n_2, p-p_2, A+p_2, B+p_2N+n_2, C+n_2, N, P)\);
17 cilk_spawn \(\text{DCMM}(m-m_2, n_2, p-p_2, A+p_2+m_2P, B+p_2N, C+m_2N, N, P)\);
18 cilk_spawn \(\text{DCMM}(m-m_2, n-n_2, p-p_2, A+p_2+m_2P, B+p_2N+n_2, C+n_2+m_2N, N, P)\);

\(\textbf{Figure 4:}\) A single-threaded divide-and-conquer matrix multiply function.

Doing the work in place with a \texttt{cilk\_sync} in the middle reduces the parallelism. How much parallelism does this in-place trick cost? The parallelism can be analyzed as follows. The \textit{work} of both algorithms satisfies this recurrence:

\[
W(N) = 8W(N/2)
\]

which has solution

\[
W(N) = O(N^3),
\]

where \(N\) is the side length of a square matrix. In fact the work is \(W(N) = N^2(2N-1)\) floating point operations.

The \textit{span} (also known as the \textit{critical path}) of the computation, if we allocated extra memory and did all 8 recursions in parallel is satisfied by the following recurrence:

\[
S(N) = S(n/2) + O(\log N).
\]

The first term, \(S(n/2)\) is the span of the recursive calls. Since all eight recursive calls run in parallel, the span is accounted for by a single call. The second term, \(O(\log N)\), is accounted for by the cost of adding the results together, which can be done in parallel. This recurrence has solution

\[
S(N) = O(\log^2 N).
\]

The parallelism of the fully parallel (not in-place) version is thus

\[
\]

For even small \(N\), this is a lot of parallelism. For example, for \(N = 4\), the parallelism is about 30. For \(N = 100\) the parallelism is tens of thousands.

Since we only have 8 cores, we can trade some of that parallelism for using less memory.

The in-place algorithm has the same work, but now the span is

\[
S(N) = 2S(N/2)
\]
\[ \begin{align*}
P_1 &= (A_{11} + A_{22}) \cdot (B_{11} + B_{22}), \\
P_2 &= (A_{21} + A_{22}) \cdot B_{11} \\
P_3 &= A_{11} \cdot (B_{12} - B_{22}) \\
P_4 &= A_{22} \cdot (B_{21} - B_{11}) \\
P_5 &= (A_{11} + A_{12}) \cdot B_{22} \\
P_6 &= (A_{21} - A_{11}) \cdot (B_{11} + B_{12}) \\
P_7 &= (A_{12} - A_{22}) \cdot (B_{21} + B_{22}) \\
C_{11} &= P_1 + P_4 - P_3 + P_7 \\
C_{12} &= P_3 + P_5 \\
C_{21} &= P_2 + P_4 \\
C_{22} &= P_1 - P_2 + P_3 + P_6
\end{align*} \]

**Figure 5:** The Strassen algorithm of Figure 2 rewritten to be more compact.

since there are two recursive calls of size \(N/2\). This recurrence has solution

\[ S(N) = O(N), \]

which seems like a much longer span. The parallelism is

\[ P(N) = W(N)/S(N) = O(n^2). \]

Thus, even though we gave up parallelism, there are still thousands-of-fold parallelism even for reasonably small matrices.

To do the analysis really correctly, I must account for the base case being bigger than 1. Thus we have

\[ S(N) = \begin{cases} 
O(N^3) & \text{if } N \leq 512, \text{ and} \\
2S(N/2) & \text{otherwise.}
\end{cases} \]

Plugging in some values for \(N\) we get the following parallelism statistics for divide-and-conquer if the base case is 512 elements.

<table>
<thead>
<tr>
<th>(N)</th>
<th>Work</th>
<th>Span</th>
<th>Parallelism</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>512^3</td>
<td>512^3</td>
<td>1</td>
</tr>
<tr>
<td>1024</td>
<td>1024^3</td>
<td>2 \cdot 512^3</td>
<td>4</td>
</tr>
<tr>
<td>2048</td>
<td>2048^3</td>
<td>4 \cdot 512^3</td>
<td>16</td>
</tr>
<tr>
<td>4096</td>
<td>4096^3</td>
<td>8 \cdot 512^3</td>
<td>64</td>
</tr>
<tr>
<td>8192</td>
<td>8192^3</td>
<td>16 \cdot 512^3</td>
<td>128</td>
</tr>
</tbody>
</table>

These numbers explain why the D&C code doesn’t really get going until the side lengths are 2048 or beyond. If we go for smaller base cases we lose performance inside DGEMM, and larger base cases don’t have much parallelism. Fortunately, we can multiply large enough matrices to get good parallelism with large enough base cases.

GotoBLAS has some other advantages over MKL. GotoBLAS is smaller and easier to install, and doesn’t cost money or have funky license mechanisms. MKL doesn’t work very well with Valgrind or Cilkscreen, which are my standard debugging tools, whereas GotoBLAS works. MKL’s installation script also gratuitously complains that MKL cannot run on a system with SELINUX, which is silly when performing a local install (for example in your home directory). I overcame the SELINUX problem by deleting the check in the MKL install script, rather than by disabling SELINUX.

### 3.6 Strassen

I implemented the standard Strassen code (not the Winograd version). That is, I used the same Strassen variant as in the judges’ code. The major differences between my code and the reference Strassen code are
cilk_spawn $P_1 = (A_{11} + A_{22}) \cdot (B_{11} + B_{22})$
cilk_spawn $C_{21} = (A_{21} + A_{22}) \cdot B_{11}$
cilk_spawn $C_{12} = A_{11} \cdot (B_{12} - B_{22})$
cilk_spawn $P_4 = A_{22} \cdot (B_{21} - B_{11})$
cilk_spawn $P_5 = (A_{11} + A_{12}) \cdot B_{22}$
cilk_spawn $C_{22} = (A_{21} - A_{11}) \cdot (B_{11} + B_{12})$
cilk_spawn $C_{11} = (A_{12} - A_{22}) \cdot (B_{21} + B_{22})$
cilk_sync;
cilk_spawn $C_{11} = P_1 + P_4 - P_5 + C_{11}$
cilk_spawn $C_{22} = P_1 - C_{21} + C_{12} + C_{22}$
cilk_sync;
cilk_for $i, j$ (for appropriate values of $i, j$)

$$(C_{21})_{i,j} \ += \ (P_4)_{i,j}$$
$$(C_{12})_{i,j} \ += \ (P_5)_{i,j}$$

Figure 6: Cilkified Strassen with fewer temporaries.

- Use row-major arrays,
- Use less memory without giving up any parallelism.

The algorithm rewritten slightly compared to Figure 2 is shown in Figure 5. To implement the Strassen code using less memory, I used $C$ for temporaries, and did the computation shown in Figure 6 instead.

The first cilk_sync is used to make sure the 7 recursive calls to matrix multiply are all finished. The second cilk_sync is needed to avoid a data race, since the last two steps overwrite the value of $C_{21}$ and $C_{12}$ needed by the two previous steps.

To further reduce memory, each of the recursive calls is performed in its own Cilk function that also computes its sum inputs. For example, to compute

$$P_1 = (A_{11} + A_{22}) \cdot (B_{11} + B_{22})$$

I wrote a separate Cilk function, as shown in Figure 7. This function, which is spawned from the main Strassen routine, allocates the two matrices for the sums $(A_{11} + A_{22}$ and $B_{11} + B_{22})$ spawns the recursive sums, syncs, calls the recursive matrix multiplication, and then frees the two temporaries. Unless another processor steals caller, then these temporary arrays will never be allocated at the same time. If the temporary arrays are needed at the same time, it is to get parallelism.

The space requirements can be analyzed as follows. For the serial program this code thus needs at most 5 temporary submatrices per level of the tree (one each for $P_1$, $P_4$, and $P_5$, plus two while computing one of the products. The space required thus satisfies recurrence

$$S(N) = 5(N/2)^2 + S(N/2).$$

This recurrence has solution

$$S(N) = O(N^2)$$

and solving for the constant factor shows that it takes about $5/3N^2$ temporary space to multiply square matrices $N$ on a side. In addition the program uses $4N^2$ space for the two input matrices and the two output versions which are checked against each other. That totals to $17/3N^2$ double values. For a $8192^3$ problem, each matrix is half a gigabyte, so the total space required is just under $3$GB. In contrast, the reference implementation requires 17 submatrices which means that it requires about $17/3N^2$ temporary space, requiring just under $5$GB. In practice my code can multiply matrices up to about $12,000$ on a side with $12$GB of RAM.

For the parallel code the space can be analyzed as follows. The outer level requires $4N^2$ space. The first level of the stack allocates 3 temporaries, for $3/4N^2$ more space. Then each of the 8 Cilk++ worker threads has a stack which
void strassen_P1 (double *P1, int P1N,
    const double *A11, int A11N,
    const double *A22, int A22N,
    const double *B11, int B11N,
    const double *B22, int B22N,
    long long m, long long n, long long p)
// P1 = (A11 + A22)*(B11+B22)
{
    double *sumA= (double*)malloc(m*p*sizeof(double));
    double *sumB= (double*)malloc(p*n*sizeof(double));
    cilk_spawn mmsum(A11, A11N,
        A22, A22N,
        sumA, p,
        m, p);
    cilk_spawn mmsum(B11, B11N,
        B22, B22N,
        sumB, p,
        p, n);
    cilk_sync;
    strassenMMult(m, n, p,
        sumA, p,
        sumB, p,
        P1, P1N);
    free(sumA);
    free(sumB);
}

Figure 7: The Cilk function to compute $P_1$.

could include the other 2 temporaries, and then $5/12N^2$ space for the recursive calls. So the total space is $97/12N^2$, which is just over twice the space required for an in-place algorithm.

I chose to implement Strassen’s algorithm instead of Winograd’s algorithm, even though Strassen requires a few more additions (15 instead of 18 additions). My intuition was the additions are cheap and easy to optimize. For example, the additions can be performed in relatively few loops. In fact, I didn’t even bother to use the BLAS code to do the matrix multiplications, even though it might have helped a little. Figure 8 shows the number of floating point operations when using the 3-loop, Strassen, Winograd, or a hybrid algorithm which uses 3-loop for the bottom matrices, and then Winograd for the top. The difference between best Strassen and the best Winograd is about 3.5% for 16384-element matrices, and less for smaller matrices.

Since I’m really using the base case at 512, I computed the number of floating point operation starting at 512 for a base case. As shown in Figure 9, the difference between Strassen and Winograd is hundredths of a percent.

Another problem with Winograd’s algorithm is that scheduling it to use little memory without giving up parallelism can be tricky. There is a schedule for Winograd that only requires two temporaries, but it’s a serial schedule. It may be possible to use few temporaries and run Winograd with parallelism, but I haven’t analyzed the space/performance tradeoff for Winograd’s algorithm. I’ve also heard that Winograd has relatively poor cache performance.

I concluded that Winograd’s algorithm is probably no faster than Strassen’s and decided not to implement it.

---

<table>
<thead>
<tr>
<th>$N$</th>
<th>$O(N^2)$</th>
<th>Strassen</th>
<th>Winograd</th>
<th>Best Strassen</th>
<th>Best Winograd</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>25</td>
<td>22</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>112</td>
<td>247</td>
<td>235</td>
<td>112</td>
<td>112</td>
</tr>
<tr>
<td>8</td>
<td>960</td>
<td>2017</td>
<td>1969</td>
<td>960</td>
<td>960</td>
</tr>
<tr>
<td>16</td>
<td>7936</td>
<td>15271</td>
<td>15079</td>
<td>7872</td>
<td>7680</td>
</tr>
<tr>
<td>32</td>
<td>64512</td>
<td>111505</td>
<td>110737</td>
<td>59712</td>
<td>57600</td>
</tr>
<tr>
<td>64</td>
<td>520192</td>
<td>798967</td>
<td>795895</td>
<td>436416</td>
<td>418560</td>
</tr>
<tr>
<td>128</td>
<td>4177920</td>
<td>5666497</td>
<td>565409</td>
<td>3128640</td>
<td>2991360</td>
</tr>
<tr>
<td>256</td>
<td>33488896</td>
<td>39960391</td>
<td>39911239</td>
<td>22195392</td>
<td>21185280</td>
</tr>
<tr>
<td>512</td>
<td>268173312</td>
<td>280902385</td>
<td>280705777</td>
<td>156547392</td>
<td>149280000</td>
</tr>
<tr>
<td>1024</td>
<td>2146435072</td>
<td>1971035287</td>
<td>1970248855</td>
<td>1100550336</td>
<td>1048892160</td>
</tr>
<tr>
<td>2048</td>
<td>17157674880</td>
<td>13816121377</td>
<td>13812975649</td>
<td>7722726720</td>
<td>7357973760</td>
</tr>
<tr>
<td>4096</td>
<td>137422176256</td>
<td>96788347111</td>
<td>96775764199</td>
<td>54134584512</td>
<td>51568730880</td>
</tr>
<tr>
<td>8192</td>
<td>1099444518912</td>
<td>677820419665</td>
<td>67770088017</td>
<td>379244081472</td>
<td>361232774400</td>
</tr>
<tr>
<td>16384</td>
<td>8795824586752</td>
<td>4745950897207</td>
<td>4745749570615</td>
<td>2655916529856</td>
<td>252963053760</td>
</tr>
</tbody>
</table>

Figure 8: The number of floating point operations required for different matrix multiplication algorithms.

<table>
<thead>
<tr>
<th>$N$</th>
<th>$O(N^2)$</th>
<th>Strassen</th>
<th>Winograd</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>268173312</td>
<td>268173312</td>
<td>268173312</td>
</tr>
<tr>
<td>1024</td>
<td>2146435072</td>
<td>1881931776</td>
<td>1881145344</td>
</tr>
<tr>
<td>2048</td>
<td>17157674880</td>
<td>13192396800</td>
<td>13189251072</td>
</tr>
<tr>
<td>4096</td>
<td>137422176256</td>
<td>92422275072</td>
<td>92409692160</td>
</tr>
<tr>
<td>8192</td>
<td>1099444518912</td>
<td>647257915392</td>
<td>647207583744</td>
</tr>
<tr>
<td>16384</td>
<td>8795824586752</td>
<td>4532013367296</td>
<td>4531812040704</td>
</tr>
</tbody>
</table>

Figure 9: The number of floating point operations required when the base case is a side length of 512 or less.

My code works on non-power-of-two-sized matrices. It does this by dividing by two, and then making corrections afterwards. For example when multiplying

\[
\begin{bmatrix}
  a & b \\
  c & d \\
  e & f
\end{bmatrix}
\cdot
\begin{bmatrix}
  h & i & j \\
  k & l & m
\end{bmatrix}
\]

we first compute this recursively

\[
\begin{bmatrix}
  a & b \\
  c & d
\end{bmatrix}
\cdot
\begin{bmatrix}
  h & i \\
  k & l
\end{bmatrix},
\]

and then we compute the extra terms

\[
C_{3,1} = eh + fk, \\
C_{3,2} = ei + fl, \text{ and} \\
C_{3,3} = ej + fm.
\]

Figure 10 shows the code for handling matrices with odd side lengths. It is $O(n^2)$ work to perform the corrections, it is parallelized, and it doesn’t seem to affect the performance.

## 4 How to Use My code

The following binaries are included.
void handle_border_cases (int m, int n, int p,
            const double *A, int AN,
            const double *B, int BN,
            double *C, int CN)
// deal with n,m,p being odd, in which case there are extra rows and columns at the end
// of A and B that need to be added into C. This work is only O(n^2). We can parallelize
// the outer loop to get the span down.
{
    if (m%2) {
        // an extra row in A and C. That extra row only affects the last row of C
        int i = m-1;
        cilk_for (int j=0; j<n; j++) {
            double sum = 0;
            for (int k=0; k<p; k++)
                sum += A[i*AN + k]*B[k*BN + j];
            C[i*CN + j] = sum;
        }
    }
    if (n%2) {
        // An extra column in B and C. That extra column only affects the last column of C
        cilk_for (int i=0; i<(m&~1); i++) {
            // Don't do the last odd column, since we just did it.
            int j = n-1;
            double sum = 0;
            for (int k=0; k<p; k++)
                sum += A[i*AN + k]*B[k*BN + j];
            C[i*CN + j] = sum;
        }
    }
    if (p%2) {
        // an extra column in A and an extra row in B. We have an outer product
        cilk_for (int i=0; i<(m&~1); i++) {
            // Don't do the last odd column since we did it.
            for (int j=0; j<(n&~1); j++) {
                int k = p-1;
                C[i*CN + j] += A[i*AN + k]*B[k*BN + j];
            }
        }
    }
}

Figure 10: Code to correct for non-power-of-two sized matrices.
1. reference/StrassenMMmult
   The reference implementation of 3-loop and Strassen, provided by the judges.
   Invoke as
   $ reference/StrassenMMmult 2048 2048 2048
   The cilkified version of reference/StrassenMMmult
2. reference/StrassenMMmultcilk
   Invoke as
   $ CILK_NPROC=8 reference/StrassenMMmultcilk 2048 2048 2048
3. strassen-3loop
   This is the D&C code as well as the Strassen code, linked against a base case using 3-loop.
   Invoke as
   $ CILK_NPROC=8 taskset 00ff ./strassen-3loop --fast 2048 2048 2048
   This can be understood as
   • CILK_NPROC=8
     Use 8 cores (this makes sense on a two-socket system with 8 cores, each core having to HT threads).
   • taskset 00ff
     Use processors 0 through 7 (since those are distinct cores).
   • ./strassen-3loop
     The command.
   • --fast
     Use the multithreaded BLAS to compute the ”canonical” answer. If you don’t include --fast then the
     reference implementation’s 3-loop version is used, and the multithreaded BLAS is checked against that
     implementation.
   • 2048 2048 2048
     The 3 parameters $M$, $N$, and $P$ defining the matrix sizes.
4. strassen-goto
   This is the D&C code and Strassen code linked against GotoBLAS.
   It is invoked the same way as strassen-3loop.
5. strassen-mkl
   This is the D&C code and Strassen code linked against the Intel MKL.
   It is invoked the same way as strassen-3loop.

5 How to Build My Code

You need to install Cilk++, MKL, and GotoBLAS.
• I installed Cilk++ in ./cilkarts/cilk/.
• I installed MKL in ./intel/Compiler/11.1/046.
• I installed GotoBLAS in ./GotoBLAS2.
Then do make to build everything.
This code statically links everything so that the binaries can be moved to another system. Since GotoBLAS
produces machine-specific libraries, if you are running on a machine other than Nehalem, you should recompile
GotoBLAS and then rebuild my codes.