The Baby Blas

Brody Dutka

Mercer University

CSC 435: High Performance Computing

Dr. Pounds

April 13, 2022

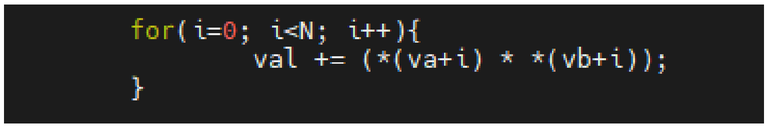
Introduction

The Baby Blas is a linear algebra library containing six routines. The different routines include the dot product between vectors, the tensor product between vectors, the product between a matrix and a vector, matrix multiplication, a direct linear solver (for equation Ax = b), and an iterative linear solver. The library, however, contains three different versions of each routine. Those are a serial version, a version using POSIX threads, and a third version using OpenMP. The parallel versions of the code also holds all of the parallelism within the function, so the user can use the parallel functions within serial code. The code has also been written in C, but can be called in C or Fortran. We will be comparing the different routines to see which versions work the most efficiently for different situations.

For our study, the library was benchmarked using a system with up to 48 cores from an Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz. When compiling all of our routines, we used the GCC compiler and the O3 compiler flag. The O3 flag provided additional optimizations than the O2 flag, and since we are dealing mainly with loops and math operations, accuracy was not lost from using O3.

Dot

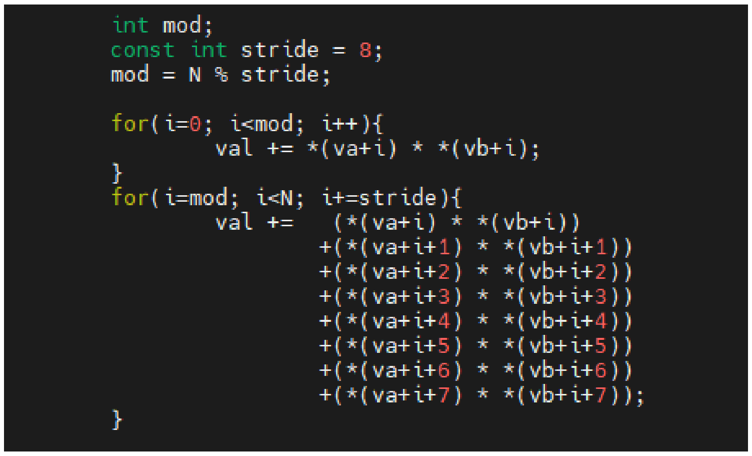
The first routine in the Baby Blas is dot.c, which computes the scalar dot product between two vectors. The algorithm used is adapted from "Matrix Computations" by Goloub and Van Loan (1996).



```
for(i=0; i<N; i++){
        val += (*(va+i) * *(vb+i));
}
```

Figure 1: dot.c code snippet of base dot product algorithm.

In order to optimize the dot product function, the loops were unrolled with a size 8 stride. The loop unrolling should aid the program in the vectorization of operations and reduce the overall loop count of the code (Hager & Wellein 2011).



```
int mod;
const int stride = 8;
mod = N % stride;

for(i=0; i<mod; i++){
        val += *(va+i) * *(vb+i);
}
for(i=mod; i<N; i+=stride){
        val +=   (*(va+i) * *(vb+i))
                +(*(va+i+1) * *(vb+i+1))
                +(*(va+i+2) * *(vb+i+2))
                +(*(va+i+3) * *(vb+i+3))
                +(*(va+i+4) * *(vb+i+4))
                +(*(va+i+5) * *(vb+i+5))
                +(*(va+i+6) * *(vb+i+6))
                +(*(va+i+7) * *(vb+i+7));
}
```

Figure 2: dot.c code snippet of unrolled dot product algorithm.

Now that the routine is optimized, both dot product algorithms were benchmarked for their megaflop performance against problem sizes ranging from 1,000 to 80,000,000. The dot product produces $2N$ flops and runs in $O(n)$ time (Goloub and Van Loan 1996).

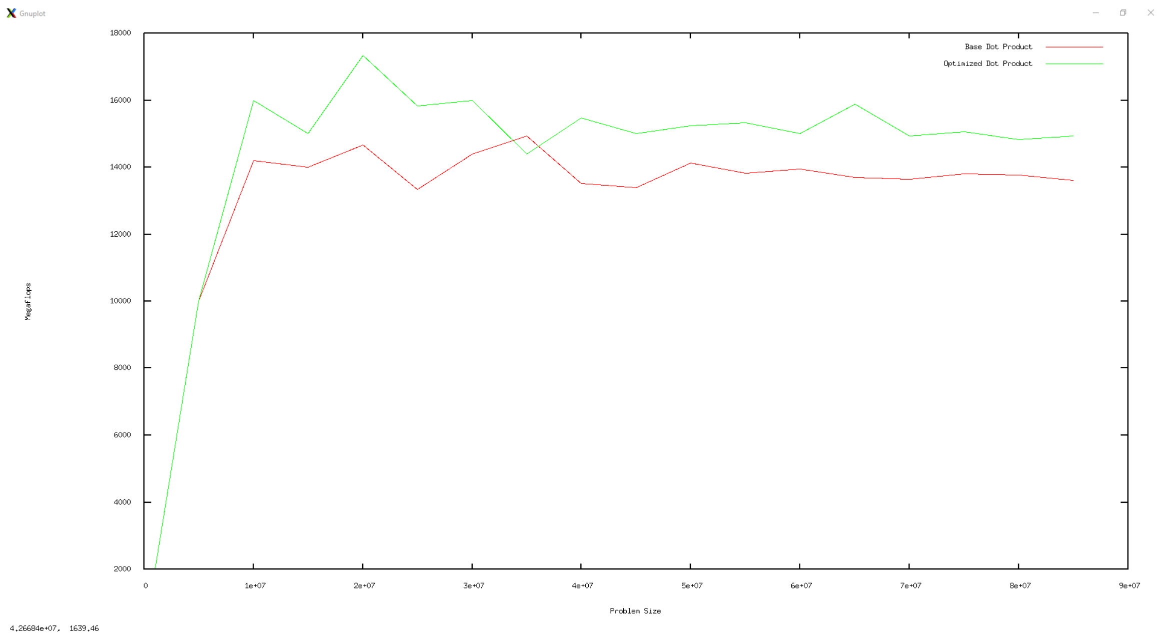
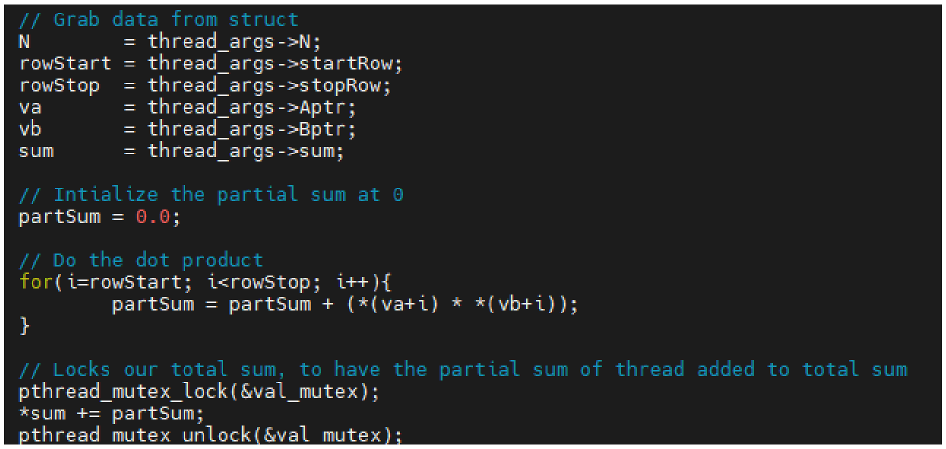


Figure 3: Shows problem size vs. megaflops for the serial dot.c routine.

The graph shows that the megaflop rate grows rapidly and relatively steadily, while the problem size goes up to 10,000,000. However, the rate at which the megaflops increase lowers significantly but reaches a peak of 1,733 megaflops at problem size 20,000,000 before declining to about 1,500 megaflops for problem sizes greater than 30,000,000. The plateau in megaflops versus problem size can be attributed to finding the cache limit of the system (Hager & Wellein 2011). In order to see a more prolonged megaflop growth, the cache capacity would need to increase.

For problem sizes up to about 700,000, the megaflops between the two algorithms were the same. After that point, the optimized version proved to be the faster option. Once the megaflops for both options leveled out, the optimized version sustained roughly 12% more megaflops than the unoptimized dot product.

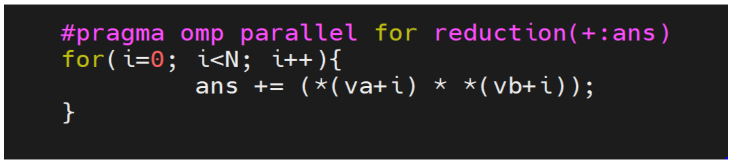
When rewriting the dot product for using Pthreads, the algorithm had to be changed slightly. In order to not see problems from the shared sum variable, each thread creates a partial sum of their respective vector partitions. Once the partial sums have been completed, a mutex is used to create the total sum from the partials, to protect against the race conditions (Nichols, Buttlar, and Farrel 1996).



```
// Grab data from struct
N        = thread_args->N;
rowStart = thread_args->startRow;
rowStop  = thread_args->stopRow;
va       = thread_args->Aptr;
vb       = thread_args->Bptr;
sum      = thread_args->sum;

// Intialize the partial sum at 0
partSum = 0.0;

// Do the dot product
for(i=rowStart; i<rowStop; i++){
        partSum = partSum + (*(va+i) * *(vb+i));
}

// Locks our total sum, to have the partial sum of thread added to total sum
pthread_mutex_lock(&val_mutex);
*sum += partSum;
pthread_mutex_unlock(&val_mutex);
```

Figure 4: Code snippet of thread worker function in dot.c for pthreads.

In the OpenMP iteration of the dot product, we used the for directive to parallelize the loop. In order to maintain accurate results, due to the shared memory, a reduction was used when summing up the final answer.



```
#pragma omp parallel for reduction(+:ans)
for( i=0;  i<N;  i++){
          ans += (*(va+i) * *(vb+i));
}
```

Figure 5: Code snippet of dot.c for OpenMP.

We tested our parallel code using the OpenMP version. We did tests over thread counts of 2, 4, 8, and 16. We also did our test over the range of problem sizes 0 to 10,000,000.

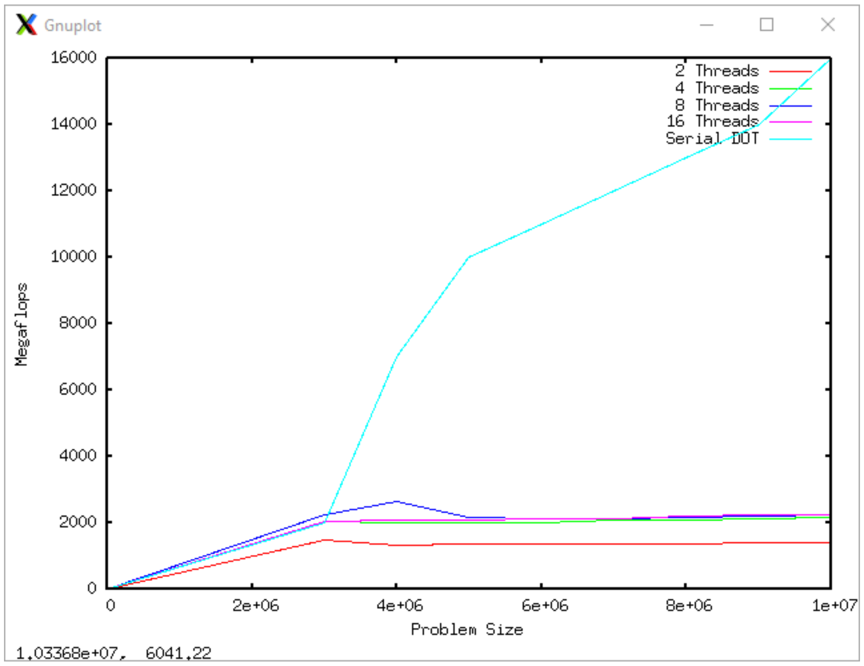
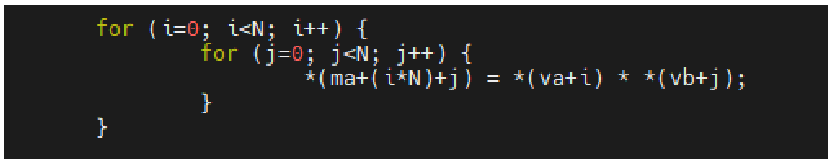


Figure 6: Plot consisting of megaflop vs. problem size, while using different thread sizes.

The serial version of the dot product greatly outperforms the parallel version at any thread count. For the OpenMP tests, the version with 8 threads performed the best, reaching a peak of 3,600. All of the threaded versions of the dot product besides 2 threads, which is lower, converge around 2,000 megaflops. At the same time this convergence occurs, we can still see the serial version rapidly increasing their performance. The serial dot product is believed to be more efficient than the parallel versions because of the cost of creating and setting up the threads used. Especially since the serial version is already blindingly fast.

VVM

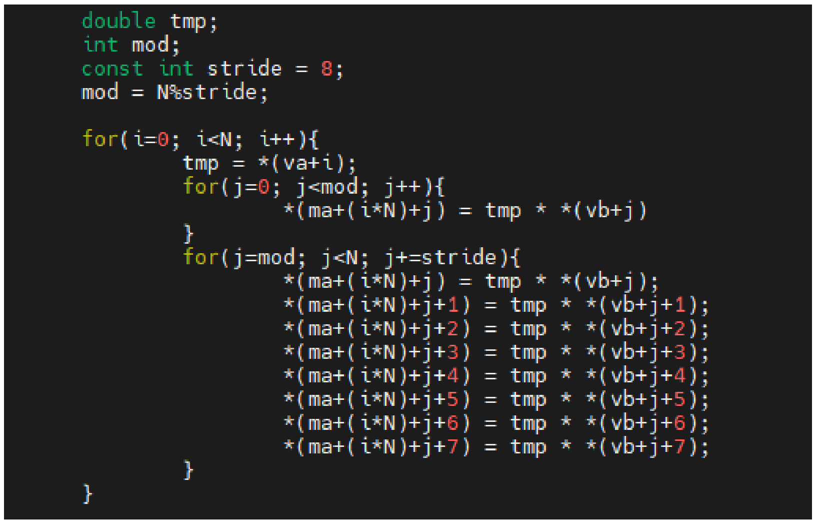
The second routine in the baby blas is vvm.c. This routine computes the tensor product between a column and a row vector. The algorithm used was also adapted from "Matrix Computations" by Goloub and Van Loan (1996).



```
for (i=0; i<N; i++) {
        for (j=0; j<N; j++) {
                *(ma+(i*N)+j) = *(va+i) * *(vb+j);
        }
}
```

Figure 7: vvm.c code snippet of base tensor product algorithm.

Two modifications were made to the base vvm algorithm. First, we unrolled the loop at a stride size of 8, which should help enable the vectorization of the inner loop of the program. The second adjustment uses a temporary variable to hold the repeated element used in the inner loop. As the inner loop crosses the row vector, the column vector stays the same. By assigning this to a variable, less time may be spent by not using a pointer for a static value (Hager & Wellein 2011).



```
double tmp;
int mod;
const int stride = 8;
mod = N%stride;

for(i=0; i<N; i++){
        tmp = *(va+i);
        for(j=0; j<mod; j++){
                *(ma+(i*N)+j) = tmp * *(vb+j)
        }
        for(j=mod; j<N; j+=stride){
                *(ma+(i*N)+j) = tmp * *(vb+j);
                *(ma+(i*N)+j+1) = tmp * *(vb+j+1);
                *(ma+(i*N)+j+2) = tmp * *(vb+j+2);
                *(ma+(i*N)+j+3) = tmp * *(vb+j+3);
                *(ma+(i*N)+j+4) = tmp * *(vb+j+4);
                *(ma+(i*N)+j+5) = tmp * *(vb+j+5);
                *(ma+(i*N)+j+6) = tmp * *(vb+j+6);
                *(ma+(i*N)+j+7) = tmp * *(vb+j+7);
        }
}
```

Figure 8: vvm.c code snippet of optimized tensor product algorithm.

Benchmarking was done on both of the algorithms in order to measure the megaflops at varying problem sizes. The vvm algorithm produces $2N^2$ flops and has a run-time of $O(n^2)$ (Goloub and Van Loan 1996).

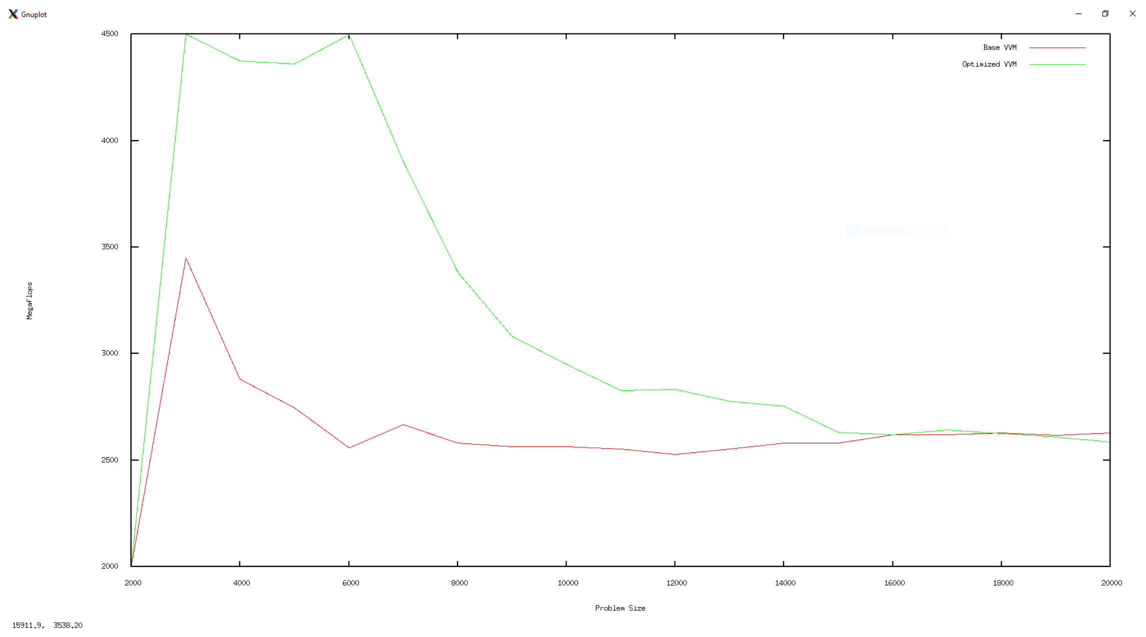
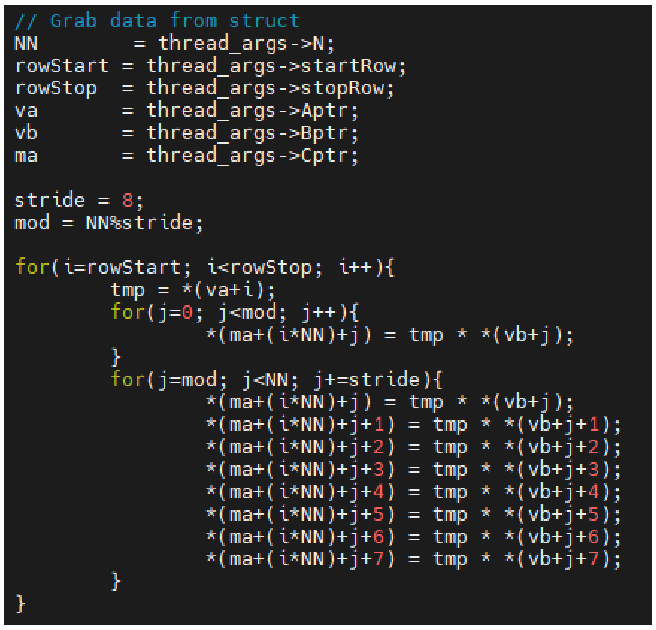


Figure 9: Shows problem size vs. megaflops for the serial vvm.c routine in the serial baby blas.

For the serial vvm routine, the cache limit seemed to be hit at a problem size of 3,000. The optimized version peaks at 4,500 megaflops and maintains this speed until a problem size of 7,000. The unoptimized version peaks at 3,500 megaflops before lowering to 4,500 megaflops at a problem size of 6,000. During the range of problem sizes between 3,000 to about 7,000–the optimized version sustains approximately a 30% increase in megaflops. After this, the megaflops decline until it converges with the unoptimized version of 2,200 megaflops at a problem size of 16,000.

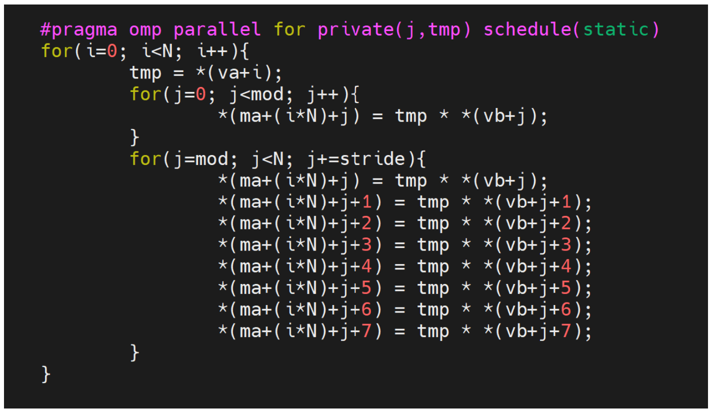
When converting our vvm algorithm to be used with pthreads, we started with the same optimized code found in Figure 6. The outer loop was divided among the threads being used to share the inner loop work. Due to the privatized outer loop, the inner loop was still unrolled since none of the threads will be writing to the exact same memory location. The temporary variable was also able to be kept for the same reason, as long as the temporary variable is private among the threads.



```
// Grab data from struct
NN        = thread_args->N;
rowStart = thread_args->startRow;
rowStop  = thread_args->stopRow;
va       = thread_args->Aptr;
vb       = thread_args->Bptr;
ma       = thread_args->Cptr;

stride = 8;
mod = NN%stride;

for(i=rowStart; i<rowStop; i++){
        tmp = *(va+i);
        for(j=0; j<mod; j++){
                *(ma+(i*NN)+j) = tmp * *(vb+j);
        }
        for(j=mod; j<NN; j+=stride){
                *(ma+(i*NN)+j) = tmp * *(vb+j);
                *(ma+(i*NN)+j+1) = tmp * *(vb+j+1);
                *(ma+(i*NN)+j+2) = tmp * *(vb+j+2);
                *(ma+(i*NN)+j+3) = tmp * *(vb+j+3);
                *(ma+(i*NN)+j+4) = tmp * *(vb+j+4);
                *(ma+(i*NN)+j+5) = tmp * *(vb+j+5);
                *(ma+(i*NN)+j+6) = tmp * *(vb+j+6);
                *(ma+(i*NN)+j+7) = tmp * *(vb+j+7);
        }
}
```

Figure 10: A code snippet from the pthreads vvm thread worker function.

For the OpenMP version of the vvm functions, we also decided to keep the loop unrolling in the inner loop and incorporate the temporary variable. Even though the outer loop is being parallelized, the inner loop should still be able to be unrolled without interference in the results. We ensured no problems within the inner loop by making the variable j private. The schedule directive was also used and set to static to quickly iterate through the loop in parallel.



```
#pragma omp parallel for private(j,tmp) schedule(static)
for(i=0; i<N; i++){
        tmp = *(va+i);
        for(j=0; j<mod; j++){
                *(ma+(i*N)+j) = tmp * *(vb+j);
        }
        for(j=mod; j<N; j+=stride){
                *(ma+(i*N)+j) = tmp * *(vb+j);
                *(ma+(i*N)+j+1) = tmp * *(vb+j+1);
                *(ma+(i*N)+j+2) = tmp * *(vb+j+2);
                *(ma+(i*N)+j+3) = tmp * *(vb+j+3);
                *(ma+(i*N)+j+4) = tmp * *(vb+j+4);
                *(ma+(i*N)+j+5) = tmp * *(vb+j+5);
                *(ma+(i*N)+j+6) = tmp * *(vb+j+6);
                *(ma+(i*N)+j+7) = tmp * *(vb+j+7);
        }
}
```

Figure 11: A code snippet from the OpenMP vvm routine.

We benchmarked our OpenMP version using varying thread sizes, this time using 2, 4, 8, 16, and 32 threads. We test our vvm routine over the problem sizes ranging from 0 to 20,000.

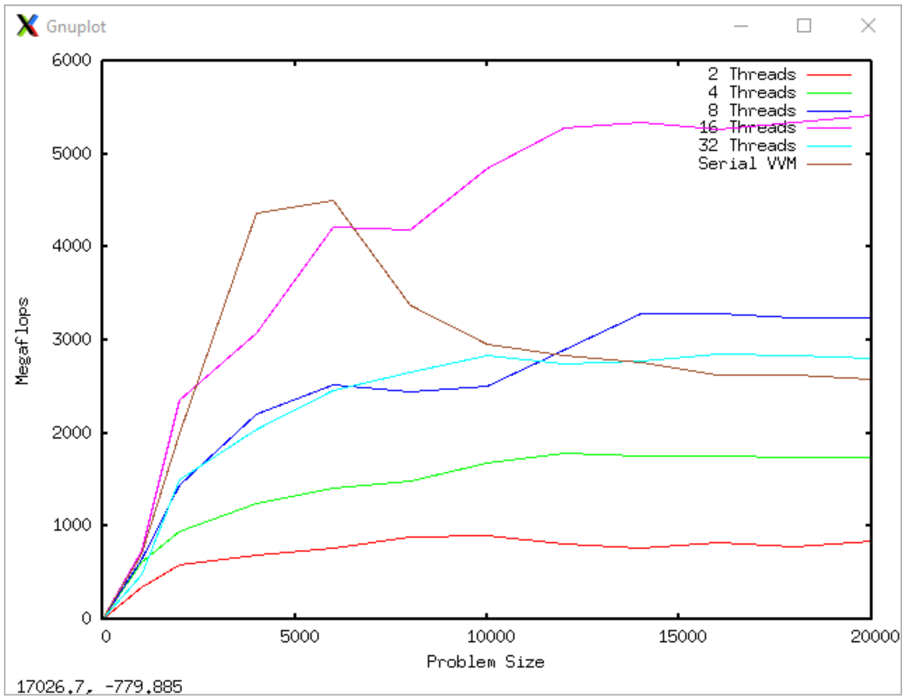
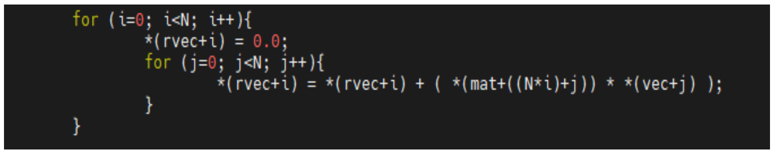


Figure 12: Shows problem size vs. megaflops for the OpenMP vvm.c routine using different thread counts.

At a problem size of about 10,000, we seem to start reaching our cache limit. The test that had the best overall performance is when we used 8 threads, this reaches a peak of roughly 5,500 megaflops, and converges at around the same speeds. The serial version is the next best performer, peaking at roughly 4,500 megaflops. When it converges though, it meets the tests at thread counts of 8 and 32.

MVV

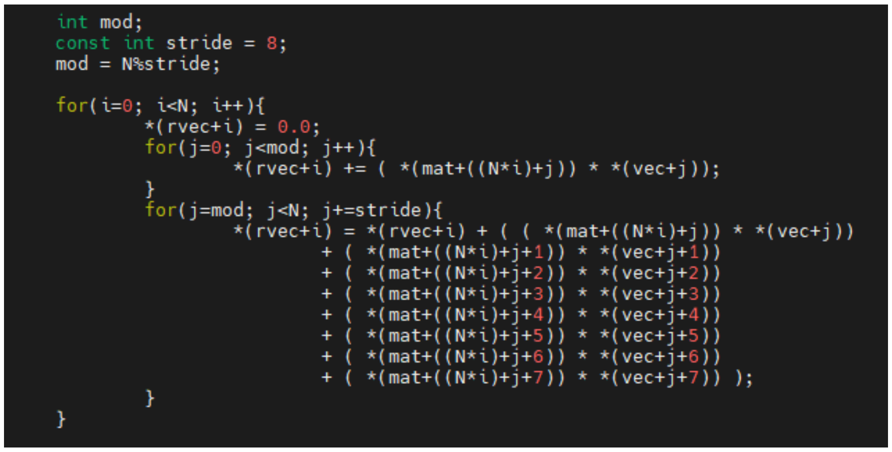
The mvv routine computes the vector product between a matrix and a vector. The code for the base algorithm is from “Matrix Computations” by Goloub and Van Loan (1996).



```
for (i=0; i<N; i++){
        *(rvec+i) = 0.0;
        for (j=0; j<N; j++){
                *(rvec+i) = *(rvec+i) + ( *(mat+((N*i)+j)) * *(vec+j) );
        }
}
```

Figure 13: Code snippet for base mvv algorithm.

The optimization made to the mvv code was, like the others, to unroll the inner loop. A size 8 stride was applied again to shorten the loop count and help the vectorization process.



```
int mod;
const int stride = 8;
mod = N%stride;

for(i=0; i<N; i++){
        *(rvec+i) = 0.0;
        for(j=0; j<mod; j++){
                *(rvec+i) += ( *(mat+((N*i)+j)) * *(vec+j));
        }
        for(j=mod; j<N; j+=stride){
                *(rvec+i) = *(rvec+i) + ( ( *(mat+((N*i)+j)) * *(vec+j))
                        + ( *(mat+((N*i)+j+1)) * *(vec+j+1))
                        + ( *(mat+((N*i)+j+2)) * *(vec+j+2))
                        + ( *(mat+((N*i)+j+3)) * *(vec+j+3))
                        + ( *(mat+((N*i)+j+4)) * *(vec+j+4))
                        + ( *(mat+((N*i)+j+5)) * *(vec+j+5))
                        + ( *(mat+((N*i)+j+6)) * *(vec+j+6))
                        + ( *(mat+((N*i)+j+7)) * *(vec+j+7)) );
        }
}
```

Figure 14: Code snippet from optimized mvv code.

To measure the speed, both algorithms were tested on problem sizes ranging from 2,000 to 20,000 for their megaflops. The mvv algorithm has $3N^2$ flops and has an $O(n^2)$ complexity (Goloub and Van Loan 1996).

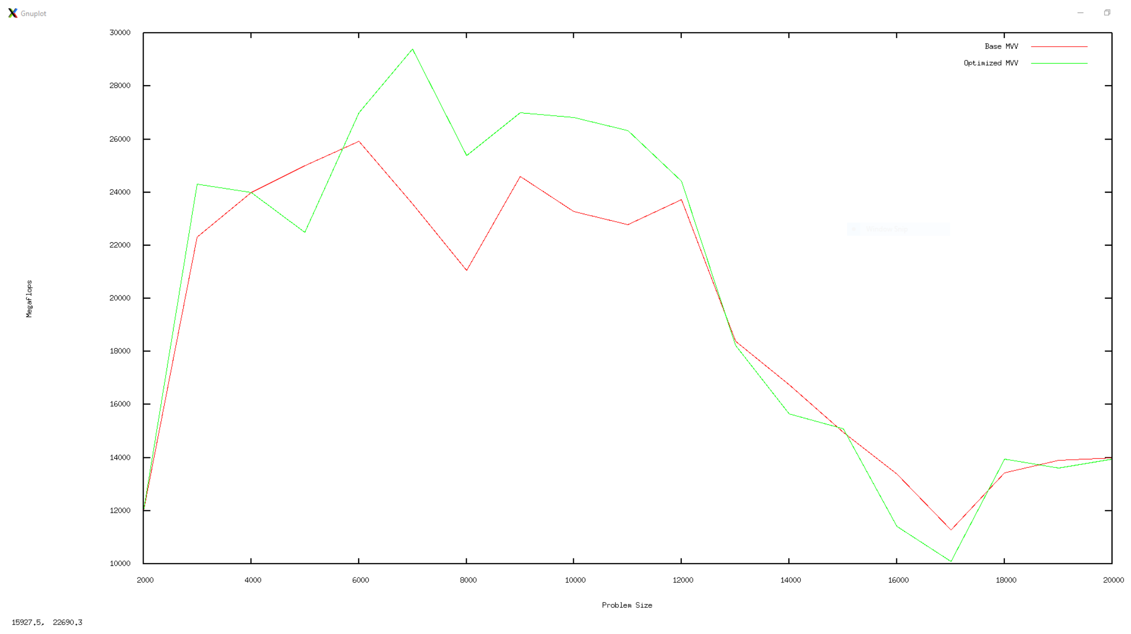
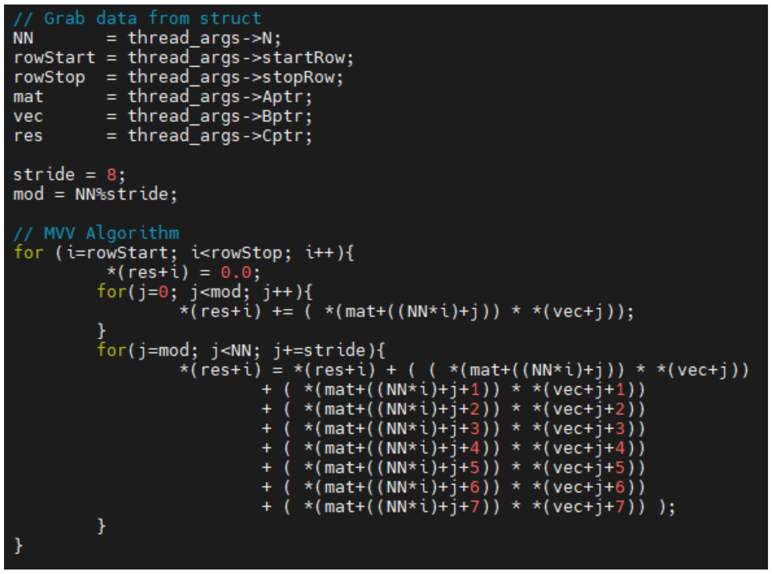


Figure 15: Shows problem size vs. megaflops for the serial mvv.c routine in the baby blas.

At a problem size of around 3,000 , the megaflop growth rate slows down as it nears the cache limit. Until about a problem size of 6,000 , the megaflops continued to grow slowly to the peak of 29,400 megaflops. From there, the megaflops of the program lowers until a problem size of 17,000 is hit. Then the megaflops slightly rise until an equilibrium is reached at 14,000 megaflops for larger sized problems.

The optimized version prevails in speed during the initial growth, but only slightly. While the problem size is between 6,000 and 12,000 , the optimized version of the mvv code produces approximately 19% more megaflops. However, both methods run at roughly the same speeds past this region.

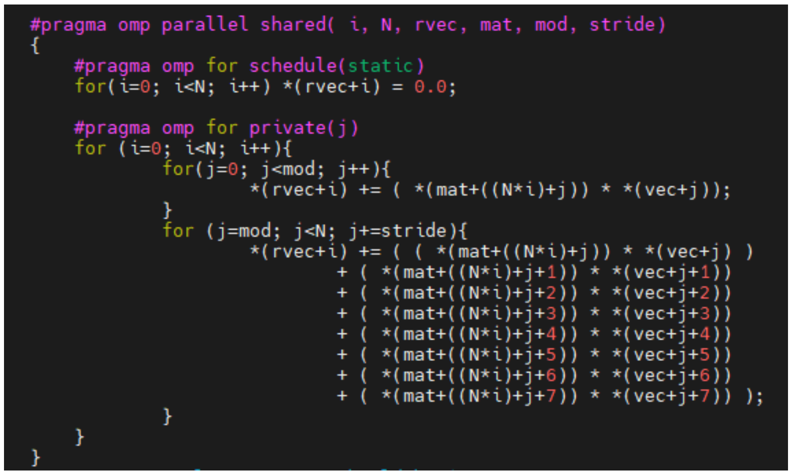
Likewise to the vvm, we can carry over the optimized mvv for our parallel code, seen in Figure 14. Since we parallelize the outer loop, the inner summations will not have a race condition (Nichols, Buttlar, and Farrel 1996). Thus, we can unroll the loop as well.



```
// Grab data from struct
NN       = thread_args->N;
rowStart = thread_args->startRow;
rowStop  = thread_args->stopRow;
mat      = thread_args->Aptr;
vec      = thread_args->Bptr;
res      = thread_args->Cptr;

stride = 8;
mod = NN%stride;

// MVV Algorithm
for (i=rowStart; i<rowStop; i++){
            *(res+i) = 0.0;
        for(j=0; j<mod; j++){
                *(res+i) += ( *(mat+((NN*i)+j)) * *(vec+j));
        }
        for(j=mod; j<NN; j+=stride){
                *(res+i) = *(res+i) + ( ( *(mat+((NN*i)+j)) * *(vec+j))
                        + ( *(mat+((NN*i)+j+1)) * *(vec+j+1))
                        + ( *(mat+((NN*i)+j+2)) * *(vec+j+2))
                        + ( *(mat+((NN*i)+j+3)) * *(vec+j+3))
                        + ( *(mat+((NN*i)+j+4)) * *(vec+j+4))
                        + ( *(mat+((NN*i)+j+5)) * *(vec+j+5))
                        + ( *(mat+((NN*i)+j+6)) * *(vec+j+6))
                        + ( *(mat+((NN*i)+j+7)) * *(vec+j+7)) );
        }
}
```

Figure 16: Code snippet from pthreads mvv thread worker function.

Our mvv code for OpenMP was slightly modified from the optimized serial version seen in Figure 14. However, where the results vector is usually initialized within the main loop, a separate loop was used to initialize the entire vector in parallel. Once this is done, the main mvv algorithm that has been unrolled is used. We know this is a safe implementation since the outer loop has been parallelized, which will prevent race conditions when writing to the results vector.



```
#pragma omp parallel shared( i, N, rvec, mat, mod, stride)
{
    #pragma omp for schedule(static)
    for(i=0; i<N; i++) *(rvec+i) = 0.0;

    #pragma omp for private(j)
    for (i=0; i<N; i++){
            for(j=0; j<mod; j++){
                    *(rvec+i) += ( *(mat+((N*i)+j)) * *(vec+j));
            }
            for (j=mod; j<N; j+=stride){
                    *(rvec+i) += ( ( *(mat+((N*i)+j)) * *(vec+j) )
                            + ( *(mat+((N*i)+j+1)) * *(vec+j+1))
                            + ( *(mat+((N*i)+j+2)) * *(vec+j+2))
                            + ( *(mat+((N*i)+j+3)) * *(vec+j+3))
                            + ( *(mat+((N*i)+j+4)) * *(vec+j+4))
                            + ( *(mat+((N*i)+j+5)) * *(vec+j+5))
                            + ( *(mat+((N*i)+j+6)) * *(vec+j+6))
                            + ( *(mat+((N*i)+j+7)) * *(vec+j+7)) );
            }
    }
}
```

Figure 17: Code snippet from OpenMP mvv algorithm.

We tested the OpenMP matrix vector multiplication using 2, 4, 8, 16, and 32 threads. We used problem sizes from 0 to 20,000 to run our benchmarks.

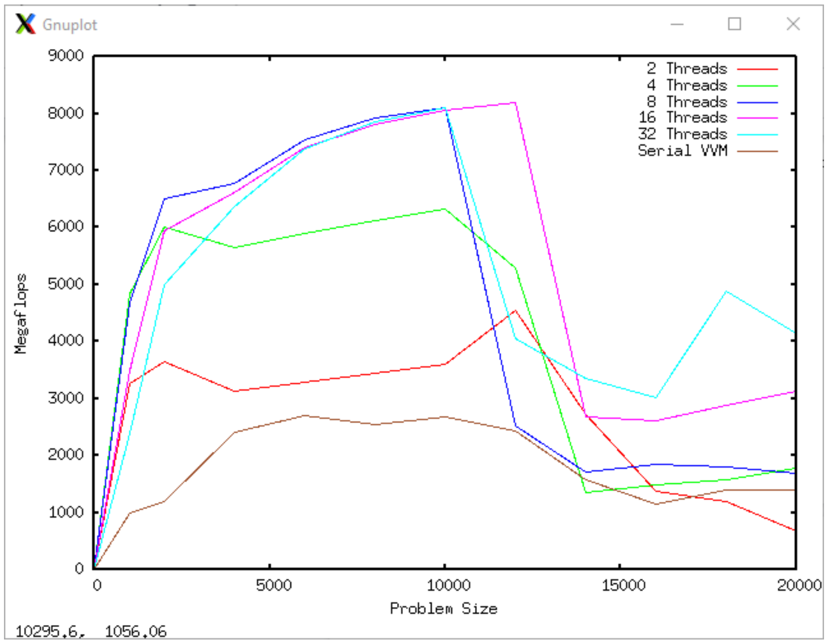
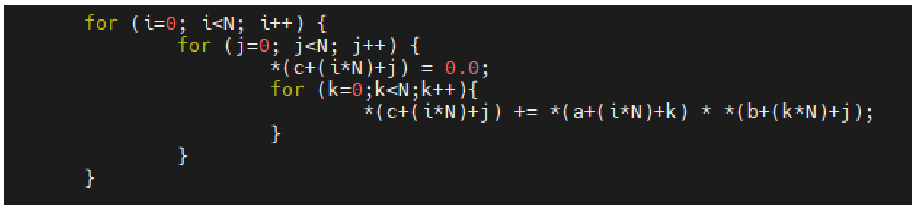


Figure 18: Shows problem size vs. megaflops for the OpenMP mvv routine using different thread counts.

The initial growth was roughly similar between the different runs. The thread counts of 8, 16, and 32 follow a similar path and reach a peak of roughly 8 gigabits. The thread size of 16 sustains this for longer before joining the rest. Once we get to larger sizes, and the megaflops start to converge, we see the 32 thread count on top at a little over 4 gigaflops. It is also important to note all of the parallel iterations ran faster initially, but the serial version converged with the thread sizes of 2, 4, and 8 once the problem size got large.

MMM

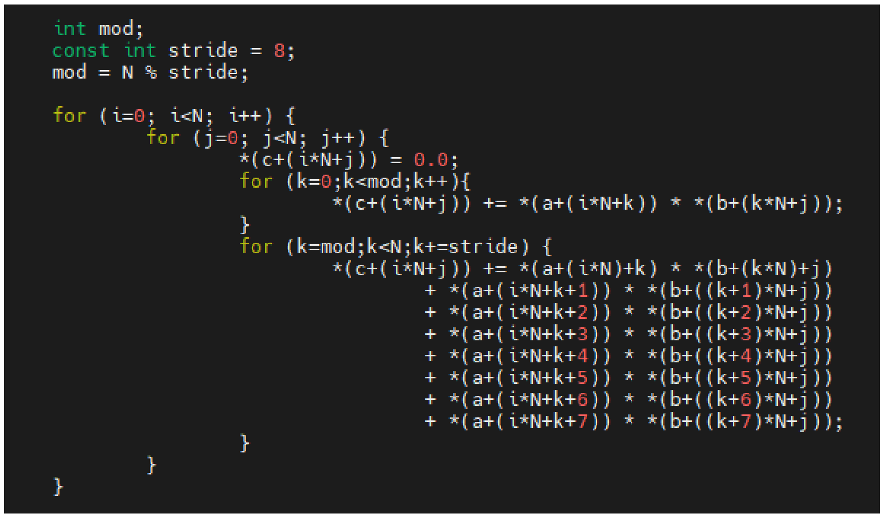
The last routine in the baby blas library before the two linear solvers is for matrix multiplication. The base algorithm used was also adapted from “Matrix Computations” by Goloub and Van Loan (1996).



```
for (i=0; i<N; i++) {
        for (j=0; j<N; j++) {
                *(c+(i*N)+j) = 0.0;
                for (k=0;k<N;k++){
                        *(c+(i*N)+j) += *(a+(i*N)+k) * *(b+(k*N)+j);
                }
        }
}
```

Figure 19: Code snippet from base mmm algorithm.

Since this is a rather basic algorithm like the others, the only optimization was to unroll the loop with a size 8 stride.



```
int mod;
const int stride = 8;
mod = N % stride;

for (i=0; i<N; i++) {
        for (j=0; j<N; j++) {
                *(c+(i*N+j)) = 0.0;
                for (k=0;k<mod;k++){
                        *(c+(i*N+j)) += *(a+(i*N+k)) * *(b+(k*N+j));
                }
                for (k=mod;k<N;k+=stride) {
                        *(c+(i*N+j)) += *(a+(i*N)+k) * *(b+(k*N)+j)
                                + *(a+(i*N+k+1)) * *(b+((k+1)*N+j))
                                + *(a+(i*N+k+2)) * *(b+((k+2)*N+j))
                                + *(a+(i*N+k+3)) * *(b+((k+3)*N+j))
                                + *(a+(i*N+k+4)) * *(b+((k+4)*N+j))
                                + *(a+(i*N+k+5)) * *(b+((k+5)*N+j))
                                + *(a+(i*N+k+6)) * *(b+((k+6)*N+j))
                                + *(a+(i*N+k+7)) * *(b+((k+7)*N+j));
                }
        }
}
```

Figure 20: Code from optimized mmm algorithm.

The two mmm versions were benchmarked from problem sizes of 0 to 8,000. The matrix multiplication algorithm produces $2N^3$ flops, and has a runtime of $O(n^3)$.

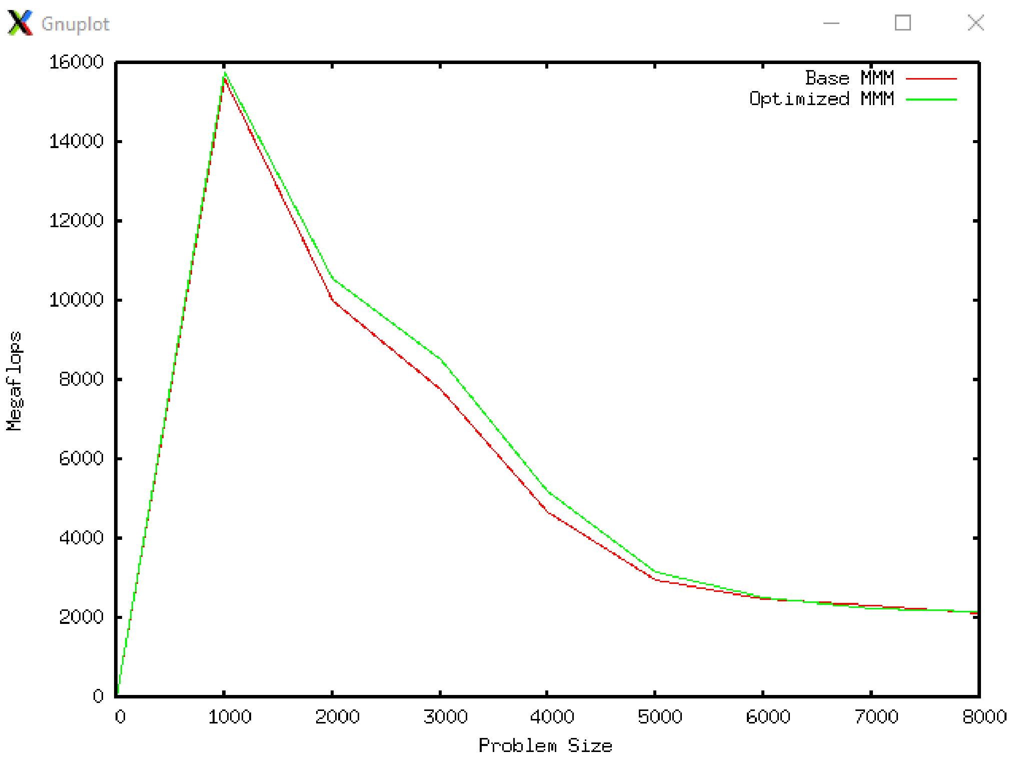
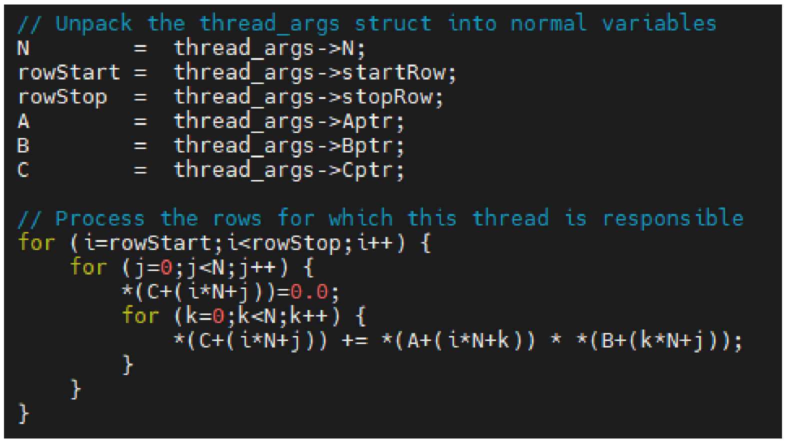


Figure 21: Shows problem size vs. megaflops for the serial mmm.c routine in the baby blas.

The benchmarking for matrix multiplication proved the cache limit is hit at a problem size of 1,000 and then declines in speed. Both programs hit a peak of almost 16,000 megaflops and converge at higher sample sizes to roughly 2,000 megaflops. The difference in performance seems negligible between the two versions. The optimized version is on top during the incline in megaflops, but by only about a single percent difference. Once the cache limit has been hit and the performance decreases, we see more separation between the two programs. Here, the optimized version sustains an 8% increase in megaflops during the time before convergence

To parallelize the function for pthreads, we divided the outer loop to partition the matrix that is being multiplied by its rows. The partitioning allows each thread to write to their section of the resultant matrix, eliminating any interference in data between threads.



```
// Unpack the thread_args struct into normal variables
N        =  thread_args->N;
rowStart =  thread_args->startRow;
rowStop  =  thread_args->stopRow;
A        =  thread_args->Aptr;
B        =  thread_args->Bptr;
C        =  thread_args->Cptr;

// Process the rows for which this thread is responsible
for (i=rowStart;i<rowStop;i++) {
    for (j=0;j<N;j++) {
        *(C+(i*N+j))=0.0;
        for (k=0;k<N;k++) {
            *(C+(i*N+j)) += *(A+(i*N+k)) * *(B+(k*N+j));
        }
    }
}
```

Figure 22: Code from Pthreads mmm thread worker function.

We tested out pthreads iteration of the matrix multiplication function over the thread sizes of 2, 4, 8, 16, and 32. We used a problem size from 0 to 6,000 in order to do this. We also included the serial version for comparison.

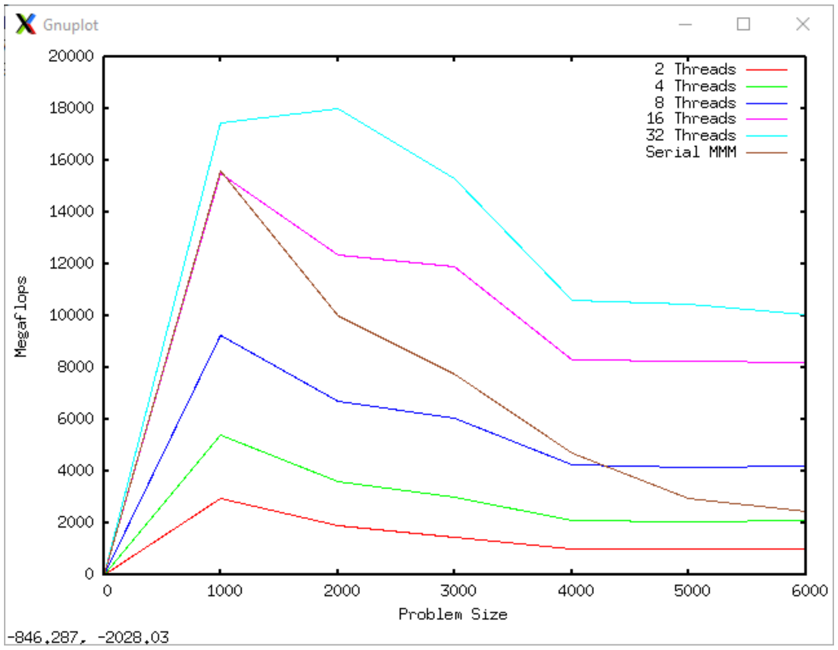
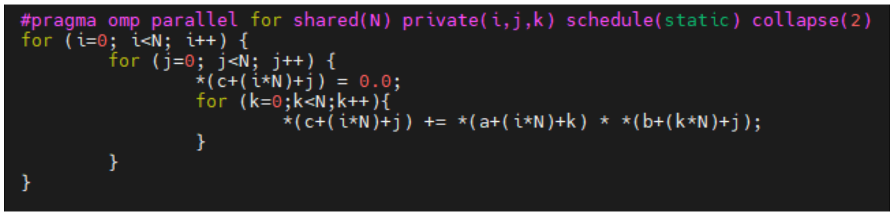


Figure 23: Shows problem size vs. megaflops for the Pthreads mmm routine using different thread counts.

The serial iteration holds its own against smaller thread sizes, and in the end converges with the 4 thread count. However, the threading scales well within our function and all counts greater than 8 produce more megaflops at larger sample sizes than the serial version. We can see our peak at about a matrix size of 2,000 reaching 18 gigaflops, while using 32 threads.

When writing the parallel version in OpenMP, we use the same strategy as pthreads. Here we divide up the outer loop while keeping the inside loops private to each thread. This partitions the first matrix in the desired way to parallelize the code while maintaining the accuracy of the serial version. Also, we can use the collapse directive since we have two nested loops without anything breaking them. This allows for a better breaking up of the loops by the OpenMP threads. The schedule call is also used. We chose a static schedule since the work among the threads should be relatively even, and stepping through the loops by blocks would be most efficient.



```
#pragma omp parallel for shared(N) private(i,j,k) schedule(static) collapse(2)
for (i=0; i<N; i++) {
        for (j=0; j<N; j++) {
                *(c+(i*N)+j) = 0.0;
                for (k=0;k<N;k++){
                        *(c+(i*N)+j) += *(a+(i*N)+k) * *(b+(k*N)+j);
                }
        }
}
```

Figure 24: Code from OpenMP mmm routine.

We benchmarked the OpenMP mmm routine at thread counts of 2, 4, 8, 16, and 32. We tested each of these at problem sizes ranging from 0 to 6,000. We also included the optimized serial code to compare the versions.

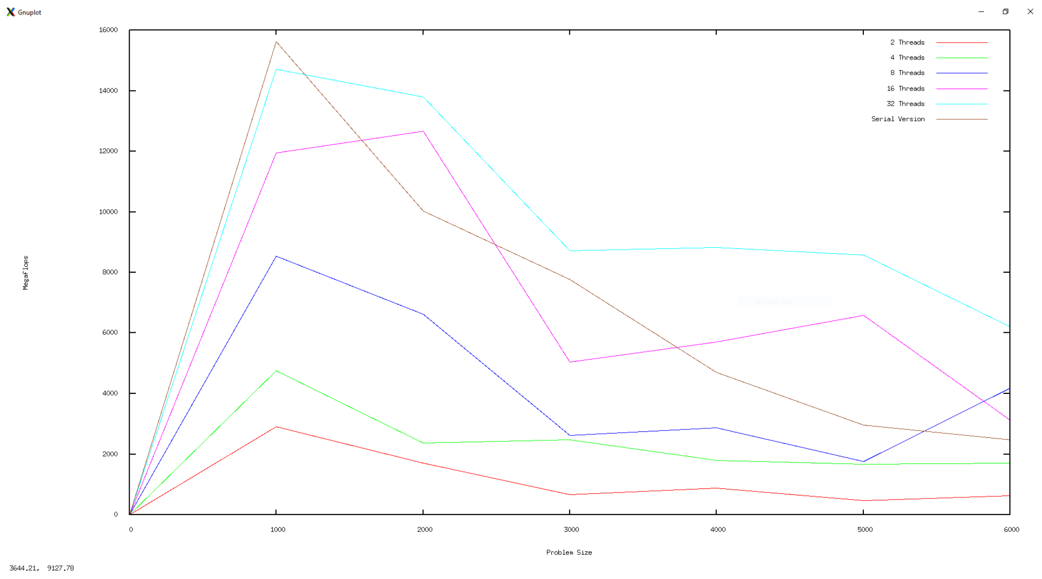


Figure 25: Shows problem size vs. megaflops for the OpenMP mmm routine using different thread counts.

As we saw in Figure 21, the peak and cache limit was at the problem size of 1,000. Here we see that all thread counts lower than 32 were not comparable to the serial version. The serial version hits the highest megaflop value, but the 32-thread version sustains about 14% more megaflops than the serial code. We also did not see a drop off in megaflops from increasing the threads, so we have not found a bottleneck to scalability at these thread counts and problem sizes. Further down the graph, the serial version converges closest with the 16-thread run. While those with fewer threads converged below their megaflop value, the larger thread counts converged at a higher megaflop rate from larger problem sizes.While those with less threads converged below their megaflop value, and the larger thread counts converged at a higher megaflop rate from larger problem sizes.

A speedup plot was made using the OpenMP mmm version. We tested a problem size of 5,000 using thread sizes ranging from 1 to 10. We used Amdahl's Law in order to fit our curve.

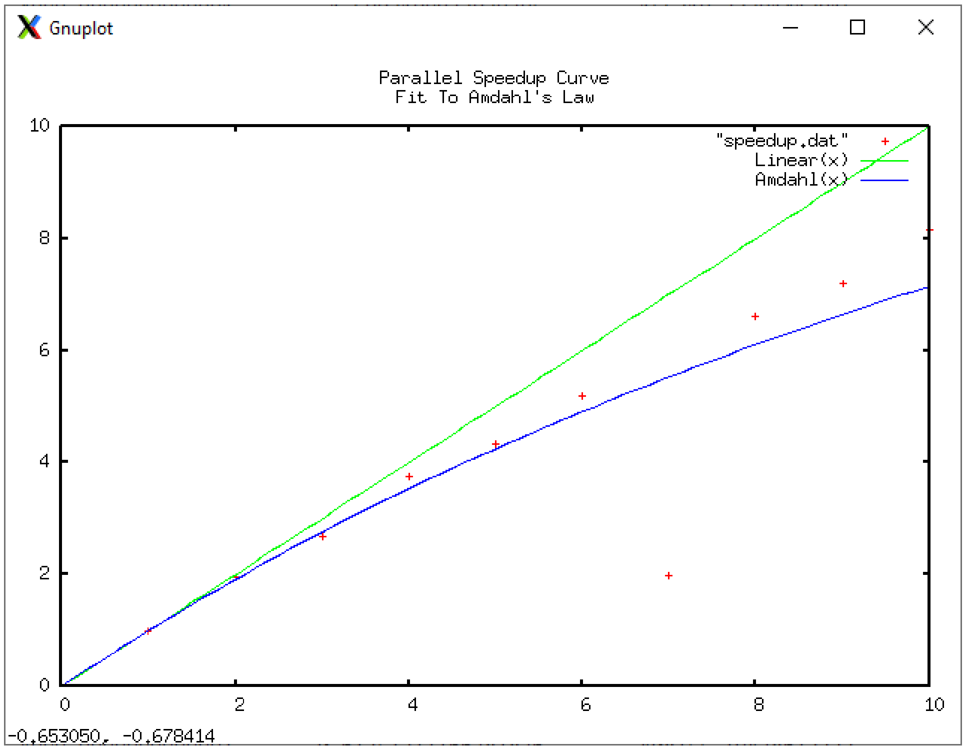
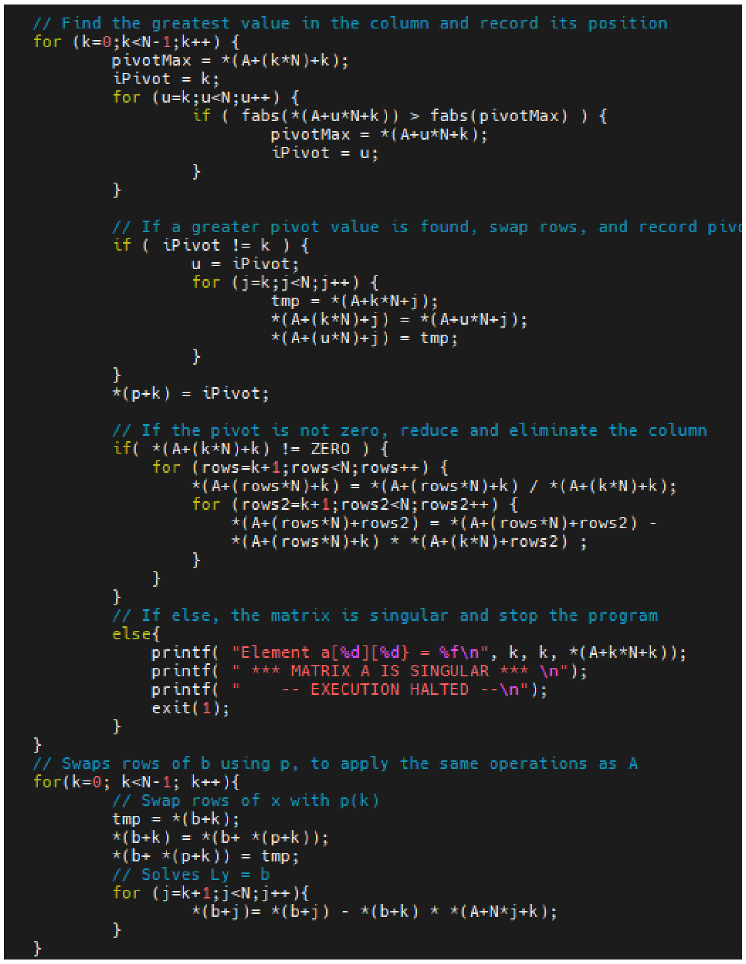


Figure 26: Speed up plot for our OpenMP mmm routine.

The speedup plot tells us we are seeing a fair amount of parallelism within our mmm function. That means the code does not serialize when run, and we are able to see an increase in performance across many thread sizes. Thi also tells us we are scaling well since the curve does not flatten out too severely as the threads go up.

DLS

The first linear solver in our library is the direct linear solver, or dls. First, our algorithm checks if the matrix is symmetric or not. If it is not, then the solver will do PLU factorization, which we adapted from Burden, Faires, and Burden (2015). We chose to use PLU, because it is more accurate than LU factorization, despite it being slightly slower.

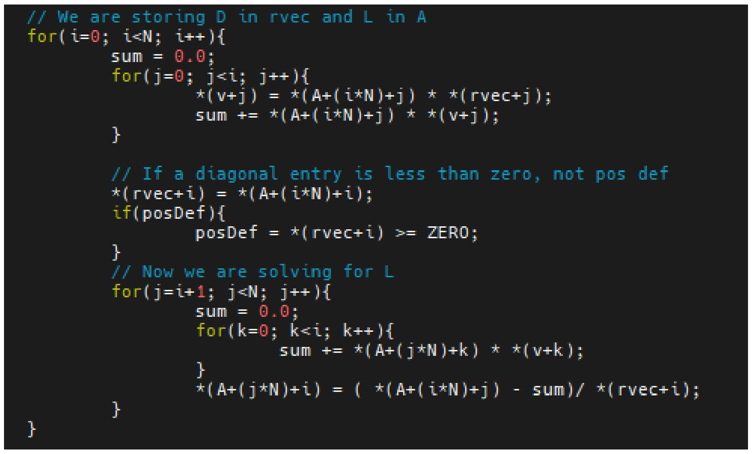


```
// Find the greatest value in the column and record its position
for (k=0;k<N-1;k++) {
        pivotMax = *(A+(k*N)+k);
        iPivot = k;
        for (u=k;u<N;u++) {
                if ( fabs(*(A+u*N+k)) > fabs(pivotMax) ) {
                        pivotMax = *(A+u*N+k);
                        iPivot = u;
                }
        }

        // If a greater pivot value is found, swap rows, and record piv
        if ( iPivot != k ) {
                u = iPivot;
                for (j=k;j<N;j++) {
                        tmp = *(A+k*N+j);
                        *(A+(k*N)+j) = *(A+u*N+j);
                        *(A+(u*N)+j) = tmp;
                }
        }
        *(p+k) = iPivot;

        // If the pivot is not zero, reduce and eliminate the column
        if( *(A+(k*N)+k) != ZERO ) {
            for (rows=k+1;rows<N;rows++) {
                *(A+(rows*N)+k) = *(A+(rows*N)+k) / *(A+(k*N)+k);
                for (rows2=k+1;rows2<N;rows2++) {
                    *(A+(rows*N)+rows2) = *(A+(rows*N)+rows2) -
                    *(A+(rows*N)+k) * *(A+(k*N)+rows2) ;
                }
            }
        }
        // If else, the matrix is singular and stop the program
        else{
            printf( "Element a[%d][%d} = %f\n", k, k, *(A+k*N+k));
            printf( " *** MATRIX A IS SINGULAR *** \n");
            printf( "    -- EXECUTION HALTED --\n");
            exit(1);
        }
}
// Swaps rows of b using p, to apply the same operations as A
for(k=0; k<N-1; k++){
        // Swap rows of x with p(k)
        tmp = *(b+k);
        *(b+k) = *(b+ *(p+k));
        *(b+ *(p+k)) = tmp;
        // Solves Ly = b
        for (j=k+1;j<N;j++){
                *(b+j)= *(b+j) - *(b+k) * *(A+N*j+k);
        }
}
```

Figure 27: Code snippet of PLU factorization in dls routine.

During the factorization if the matrix is found to be singular, the solver will stop. If it is not singular, we use back substitution to solve for the x vector after the factorization.

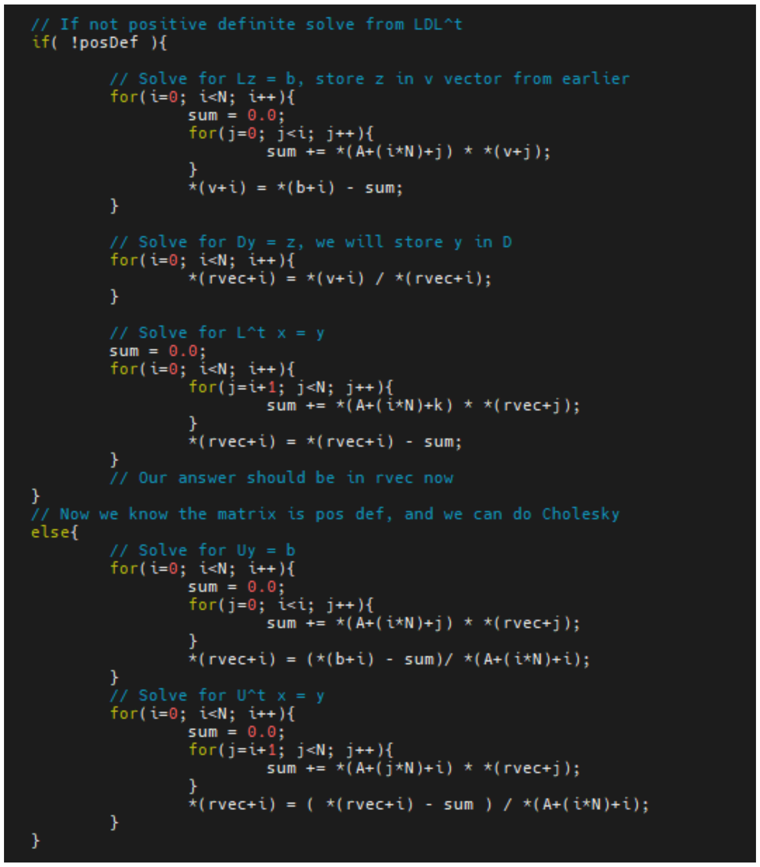
If the matrix A is symmetric, then we can do an LDL^t factorization. This is more efficient because we can assume that the upper and lower triangulars are the same.



```
// We are storing D in rvec and L in A
for(i=0; i<N; i++){
        sum = 0.0;
        for(j=0; j<i; j++){
                *(v+j) = *(A+(i*N)+j) * *(rvec+j);
                sum += *(A+(i*N)+j) * *(v+j);
        }

        // If a diagonal entry is less than zero, not pos def
        *(rvec+i) = *(A+(i*N)+i);
        if(posDef){
                posDef = *(rvec+i) >= ZERO;
        }
        // Now we are solving for L
        for(j=i+1; j<N; j++){
                sum = 0.0;
                for(k=0; k<i; k++){
                        sum += *(A+(j*N)+k) * *(v+k);
                }
                *(A+(j*N)+i) = ( *(A+(i*N)+j) - sum)/ *(rvec+i);
        }
}
```

Figure 28: Code snippet of LDL^t factorization in the dls function.

As we do our factorization, we check if any entries in the D vector are negative. If an entry is negative, then our matrix is not positive definite, and we proceed solving through back substitution. However, if the matrix is positive definite, we can use the Cholesky method to solve, from Burden, Fairies, and Burden (2015).



```c
// If not positive definite solve from LDL^t
if( !posDef ){

        // Solve for Lz = b, store z in v vector from earlier
        for(i=0; i<N; i++){
                sum = 0.0;
                for(j=0; j<i; j++){
                        sum += *(A+(i*N)+j) * *(v+j);
                }
                *(v+i) = *(b+i) - sum;
        }

        // Solve for Dy = z, we will store y in D
        for(i=0; i<N; i++){
                *(rvec+i) = *(v+i) / *(rvec+i);
        }

        // Solve for L^t x = y
        sum = 0.0;
        for(i=0; i<N; i++){
                for(j=i+1; j<N; j++){
                        sum += *(A+(i*N)+k) * *(rvec+j);
                }
                *(rvec+i) = *(rvec+i) - sum;
        }
        // Our answer should be in rvec now
}
// Now we know the matrix is pos def, and we can do Cholesky
else{
        // Solve for Uy = b
        for(i=0; i<N; i++){
                sum = 0.0;
                for(j=0; i<i; j++){
                        sum += *(A+(i*N)+j) * *(rvec+j);
                }
                *(rvec+i) = (*(b+i) - sum)/ *(A+(i*N)+i);
        }
        // Solve for U^t x = y
        for(i=0; i<N; i++){
                sum = 0.0;
                for(j=i+1; j<N; j++){
                        sum += *(A+(j*N)+i) * *(rvec+j);
                }
                *(rvec+i) = ( *(rvec+i) - sum ) / *(A+(i*N)+i);
        }
}
```

Figure 29: Code snippet of when solving symmetric matrices.

We benchmarked the serial direct linear solver for its megaflops over different problem sizes. The range of problem sizes used is from 0 to 6,000.

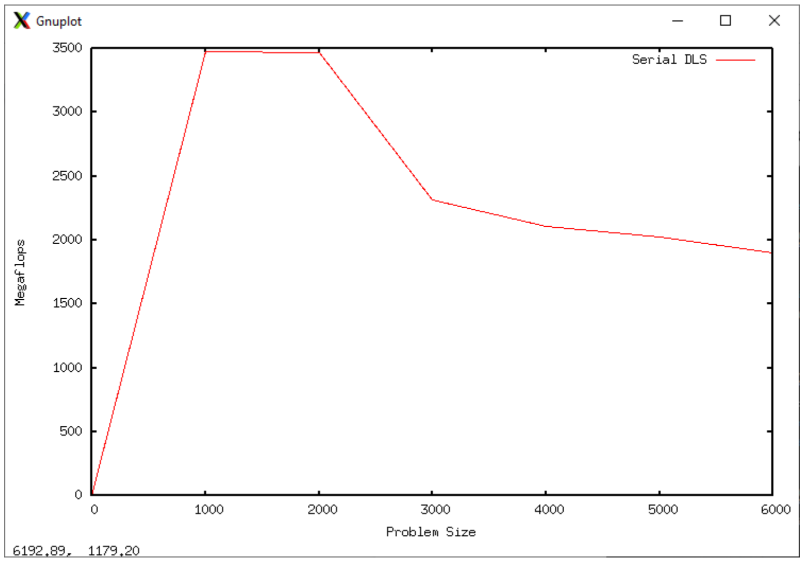
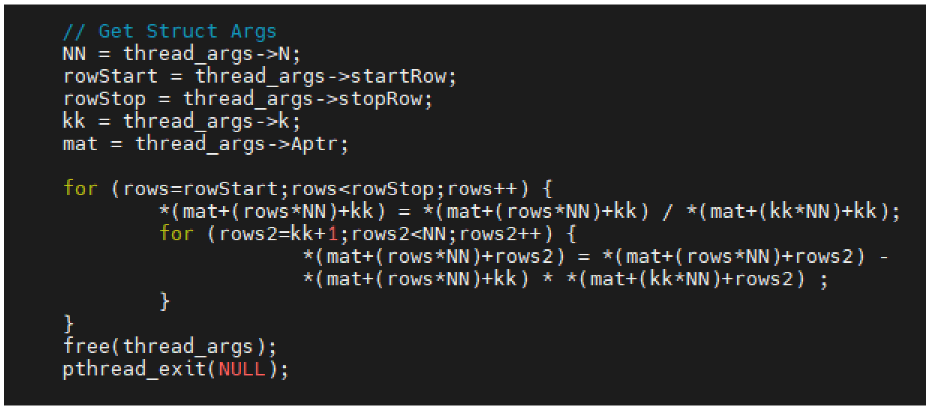


Figure 30: Plot of megaflops vs problem size using the serial dls function.

We can see from the graph that the cache limit hit around the problem size of 1,000 to 2,000. We also get a peak megaflops value of 3500 during this range. After the problem size of 2,000, the megaflop rate drops until it converges at roughly 2,000 megaflops for higher problem sizes.

The direct linear solver has many instances of data dependent operations, making us limited on what we can parallelize. The reduction and elimination phase found in Figure 27 was our target for parallelism. When using Pthreads, we divided the work in the outer loop among the threads. This should protect us from race conditions because all of the instances of writing to a variable are dependent on the outer loop. Therefore, we should expect consistent results while parallelizing the section.



```
// Get Struct Args
NN = thread_args->N;
rowStart = thread_args->startRow;
rowStop = thread_args->stopRow;
kk = thread_args->k;
mat = thread_args->Aptr;

for (rows=rowStart;rows<rowStop;rows++) {
        *(mat+(rows*NN)+kk) = *(mat+(rows*NN)+kk) / *(mat+(kk*NN)+kk);
        for (rows2=kk+1;rows2<NN;rows2++) {
                    *(mat+(rows*NN)+rows2) = *(mat+(rows*NN)+rows2) -
                    *(mat+(rows*NN)+kk) * *(mat+(kk*NN)+rows2) ;
        }
}
free(thread_args);
pthread_exit(NULL);
```

Figure 31: Code snippet inside pthreads dls worker functions.

We tested our OpenMP code across several different threads like the others. We did this using a range of problem sizes, which was from 0 to 6,000.

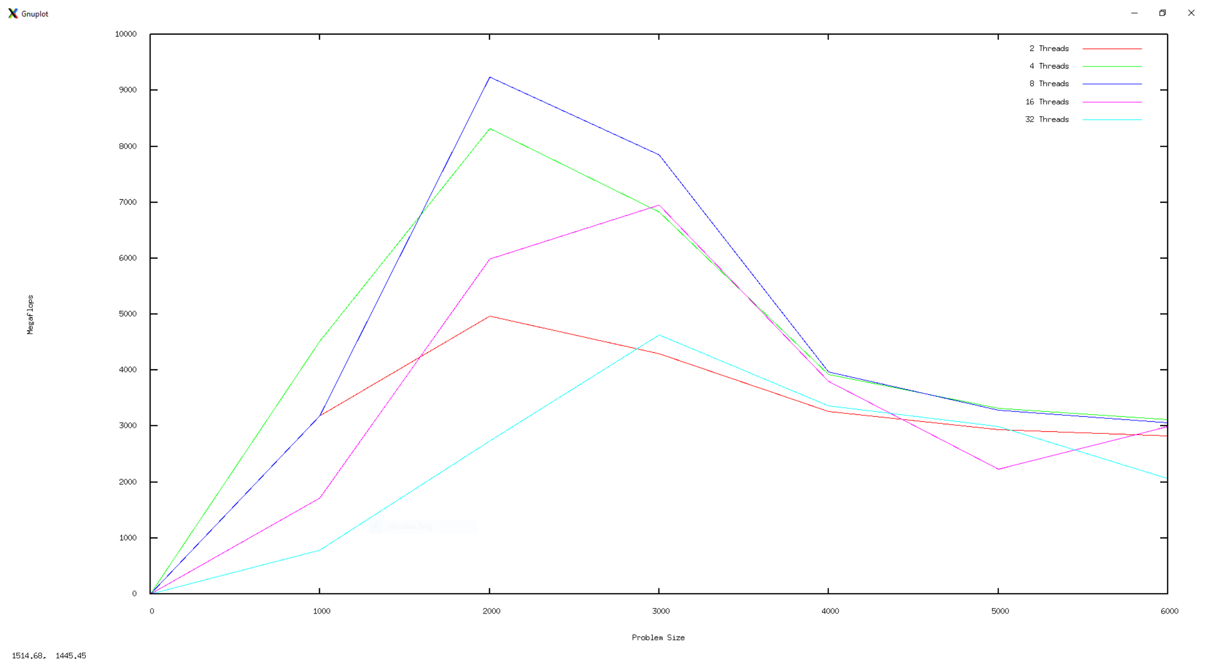
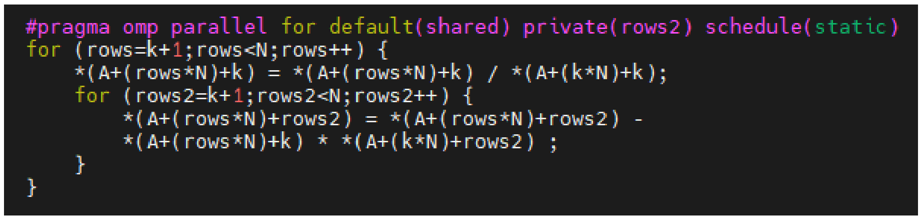


Figure 32: Plot of megaflops vs problem size, consisting of pthreads dls runs at different thread counts.

Our pthreads test reaches 9 gigaflops when running at 8 threads, at the problem size of 2,000. As the megaflops start to converge, all of the different thread counts meet at roughly 3 gigaflops.

The same section of code was chosen to be parallelized in OpenMP. The outside loop is declared as shared, to make sure it is separated among the threads when using the for directive. Then we make sure to privatize the inner loop to prevent any intermingling among threads when reducing the matrix. We also use a static schedule, which works well in our case since we are doing repeated math operations that can be relatively evenly shared.



```
#pragma omp parallel for default(shared) private(rows2) schedule(static)
for (rows=k+1;rows<N;rows++) {
    *(A+(rows*N)+k) = *(A+(rows*N)+k) / *(A+(k*N)+k);
    for (rows2=k+1;rows2<N;rows2++) {
        *(A+(rows*N)+rows2) = *(A+(rows*N)+rows2) -
        *(A+(rows*N)+k) * *(A+(k*N)+rows2) ;
    }
}
```

Figure 33: Code snippet from the OpenMP parallel portion of the dls.

We benchmarked the parallel iteration of our program using the OpenMP variant. We used thread sizes of 2, 4, 8, 16, and 32 over a problem size range of 0 to 6,000. We also included our serial benchmark in the graph for comparison.

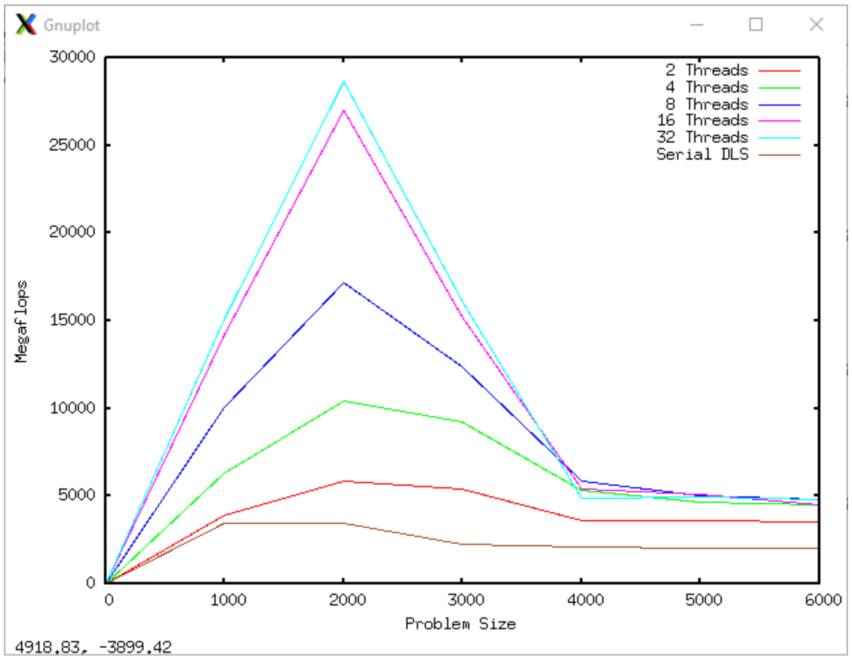


Figure 34: Plot of megaflops vs problem size, consisting of the OpenMP dls runs at different thread counts.

The plot of our OpenMP dls function shows from the problem size of 0 to about 4,000 the threads are scaling well. At 2,000 the peak of about 29 gigaflops is hit with the 32 thread run. However, after this section all of the runs besides the serial and 2 threads run converge at about 5 gigaflops while the other two fall below.

We took our OpenMP dls function and created a speedup plot using Amdahl's Law. We tested this from thread sizes of 1 to 10 at a sample size of 5,000.

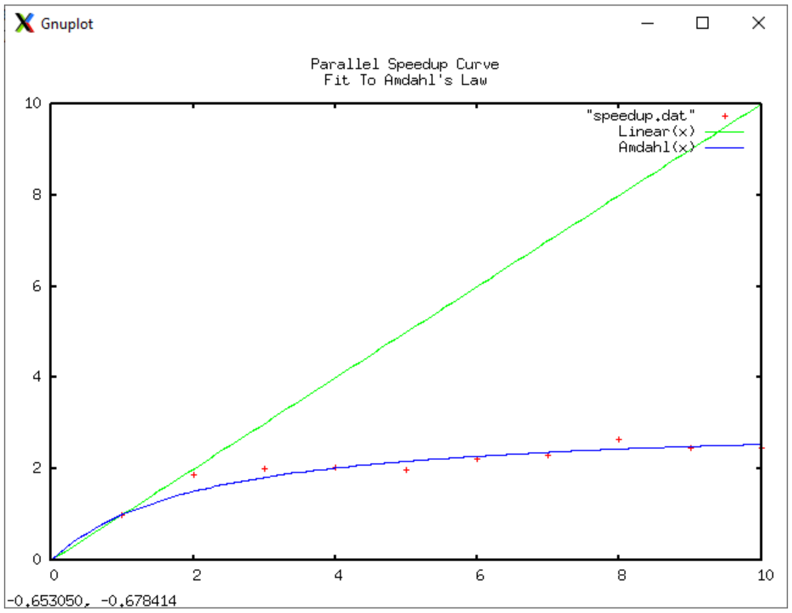
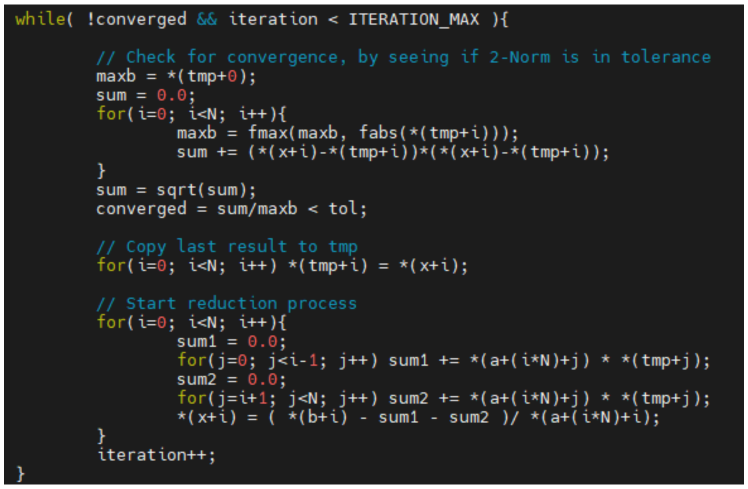


Figure 35: Speedup plot of OpenMP dls function.

The speedup plot is telling us that some parallelism is reached, but the majority of the program is running in serial. We can tell because we only see roughly a double in speedup when all the way at 10 threads.

ILS

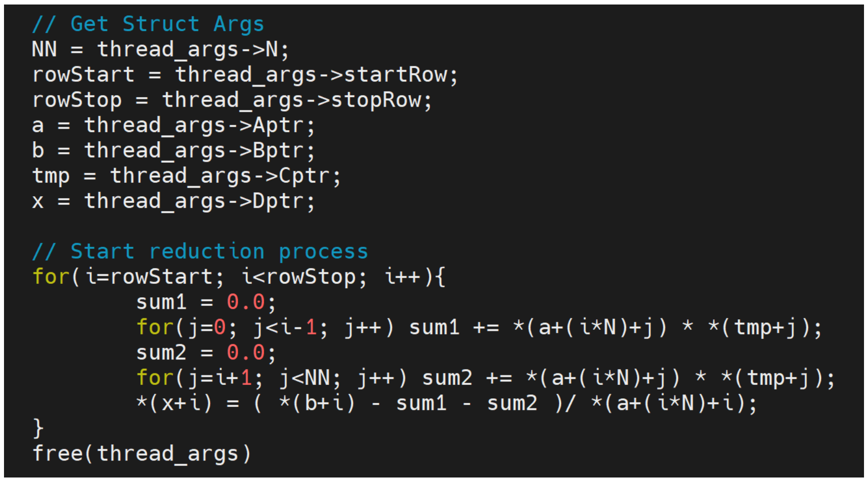
The ils function is an iterative linear solver. The function was designed to solve Ax=b through the Jacobi iterative method, as long as the matrix doesn't have zeros along the diagonal. The algorithm used for the jacobi method was adapted from "Matrix Computations" by Goloub and Van Loan (1996). We don't write over the initial matrix here because if the Jacobi method does not converge within a set amount of iterations, then the direct linear solver is called.



```c
while( !converged && iteration < ITERATION_MAX ){

        // Check for convergence, by seeing if 2-Norm is in tolerance
        maxb = *(tmp+0);
        sum = 0.0;
        for(i=0; i<N; i++){
                maxb = fmax(maxb, fabs(*(tmp+i)));
                sum += (*(x+i)-*(tmp+i))*(*(x+i)-*(tmp+i));
        }
        sum = sqrt(sum);
        converged = sum/maxb < tol;

        // Copy last result to tmp
        for(i=0; i<N; i++) *(tmp+i) = *(x+i);

        // Start reduction process
        for(i=0; i<N; i++){
                sum1 = 0.0;
                for(j=0; j<i-1; j++) sum1 += *(a+(i*N)+j) * *(tmp+j);
                sum2 = 0.0;
                for(j=i+1; j<N; j++) sum2 += *(a+(i*N)+j) * *(tmp+j);
                *(x+i) = ( *(b+i) - sum1 - sum2 )/ *(a+(i*N)+i);
        }
        iteration++;
}
```

Figure 36: Code from ils function of the Jacobi iterative solver.

The reduction process of the algorithm was the target for parallelism. In pthreads, the outer loop was split up between the threads. The sum variables were made private among the threads to allow each to work independently. We also know that race conditions should not occur, since the only shared variable being written to, is dependent on the parallelized outer loop.



```c
// Get Struct Args
NN = thread_args->N;
rowStart = thread_args->startRow;
rowStop = thread_args->stopRow;
a = thread_args->Aptr;
b = thread_args->Bptr;
tmp = thread_args->Cptr;
x = thread_args->Dptr;

// Start reduction process
for(i=rowStart; i<rowStop; i++){
        sum1 = 0.0;
        for(j=0; j<i-1; j++) sum1 += *(a+(i*N)+j) * *(tmp+j);
        sum2 = 0.0;
        for(j=i+1; j<NN; j++) sum2 += *(a+(i*N)+j) * *(tmp+j);
        *(x+i) = ( *(b+i) - sum1 - sum2 )/ *(a+(i*N)+i);
}
free(thread_args)
```

Figure 37: Code from pthreads ils worker function.

Benchmarking took place for our pthreads iterative solver in the same way as the direct linear solver was done.

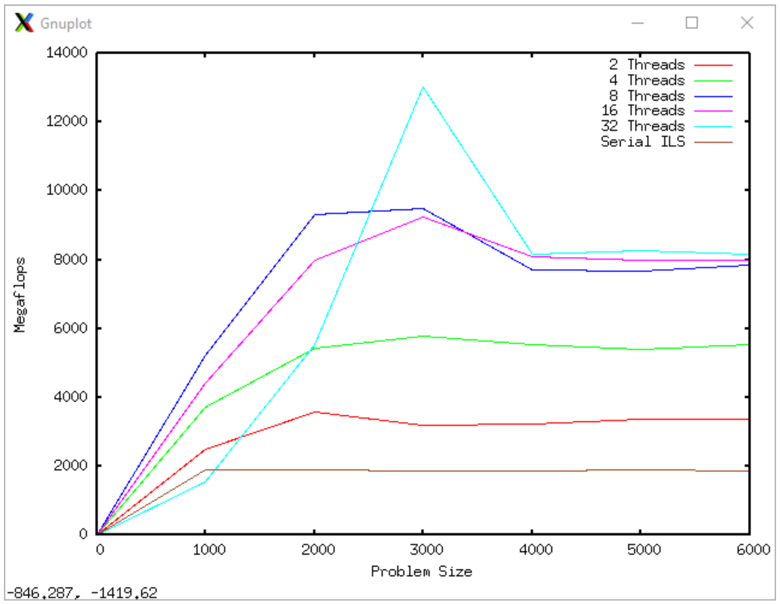
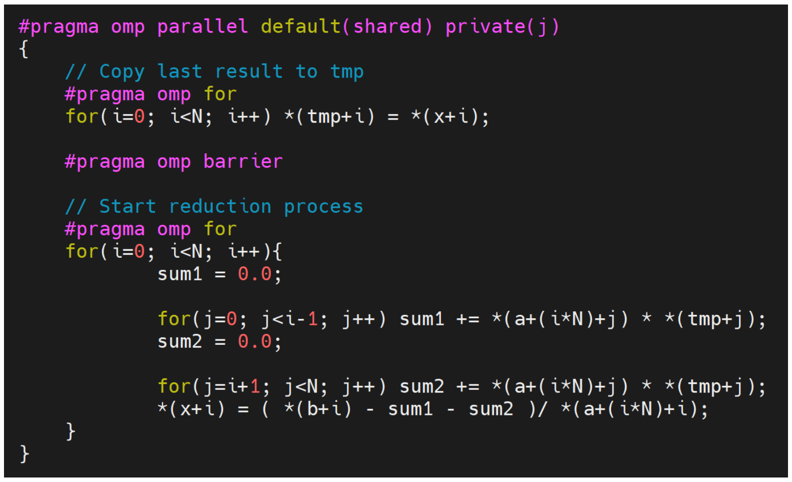


Figure 38: Plot of pthreads ils function using different thread counts.

We see strong growth in megaflops at smaller sample sizes from thread sizes of 4, 8, and 16. At a problem size of 3,000 the 32 thread run reaches the overall peak of 13 gigaflops. After the peak the 8, 16, and 32 sized runs converge at roughly 8 gigaflops. Whereas the other runs fall short, with the serial version being the slowest at 2 gigaflops.

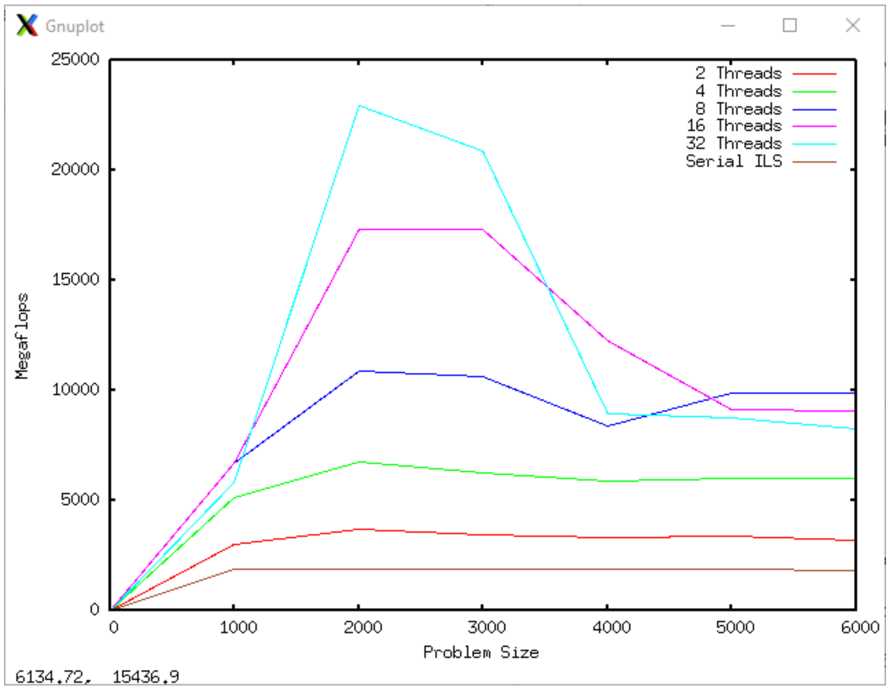
We update the same section of code when we use OpenMP for our ils. We don't use any reductions on the sums because we do not want each thread to contribute to the same variable. With this set up, we should expect similar results to our pthread code.



```
#pragma omp parallel default(shared) private(j)
{
    // Copy last result to tmp
    #pragma omp for
    for(i=0; i<N; i++) *(tmp+i) = *(x+i);

    #pragma omp barrier

    // Start reduction process
    #pragma omp for
    for(i=0; i<N; i++){
            sum1 = 0.0;

            for(j=0; j<i-1; j++) sum1 += *(a+(i*N)+j) * *(tmp+j);
            sum2 = 0.0;

            for(j=i+1; j<N; j++) sum2 += *(a+(i*N)+j) * *(tmp+j);
            *(x+i) = ( *(b+i) - sum1 - sum2 )/ *(a+(i*N)+i);
    }
}
```

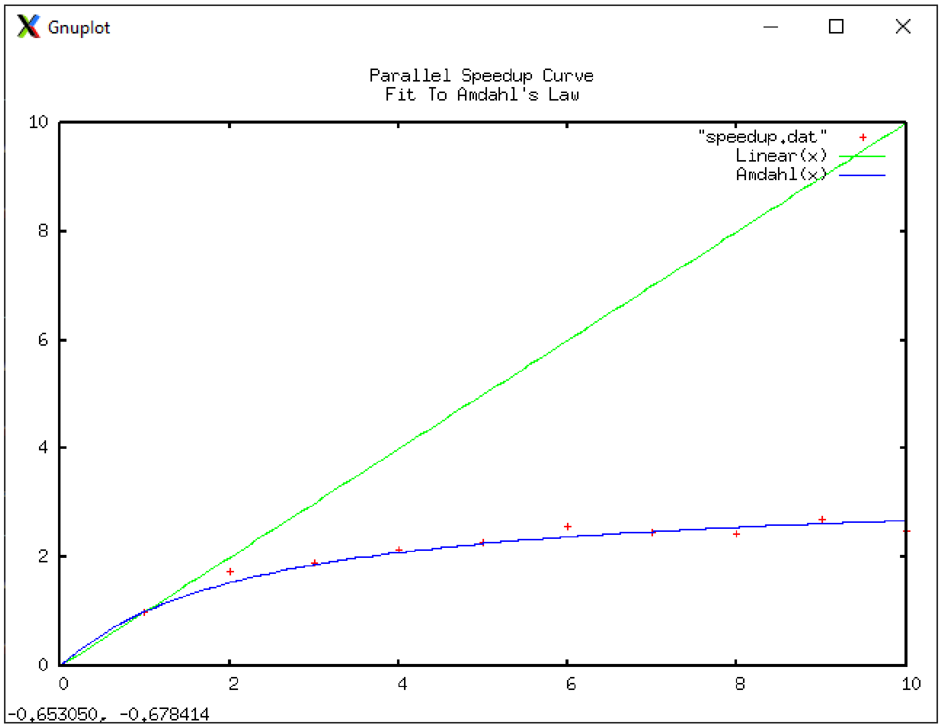
Figure 38: OpenMP portion of ils function.

We benchmarked the iterative solver in OpenMP with thread sizes of 2, 4, 8, 16, and 32. We tested all of these across a range from 0 to 6,000. We also included the serial iterative solver for comparison.



We can see the cache limit being hit at a problem size of roughly 2,000. Notice, as we use more threads during the range of 2,000 to 4,000, there is a correlation to megaflops and thread count. With only 4 threads in use, we see a double in performance when compared to our serial version. As the problem sizes get larger, the thread counts of 32, 16, and 8 converge at roughly 1 gigaflop. The lower thread counts also converge, but at a lower megaflop value.

Amdahl's Law was used to build a speedup plot for our OpenMP iterative linear solver. We used a problem size of 5,000 when collecting this data and a thread range of 1 through 10.



From the plot, we can infer that the bulk of the program has remained serialized. This is expected since a majority of the computations within our routine are data dependent, which forces serialization. We also only parallized a small portion, and due to the iterative nature, have to incur thread overhead more often. By the end, we still find the program doubled in speed.

Discussion

From the graphs of the parallel versions of our library, we can see there is generally an optimal amount of threads to most efficiently compute a problem. Various issues can be the cause of serialization that can limit the scalability of the routine. More generally, most parallel problems communicate between threads, which will always result in some serialization. For our purposes, we saw bottlenecks due to many of the operations utilizing shared resources, which has a serialized execution. For our smaller routines, the overhead of setting up and creating individual threads can limit our scalability because of the already short runtime. In our more complex routines, like the dls or ils, the mutual dependencies needed throughout severely limits our parallelization.

When deciding whether a piece of code needs to be parallelized it is good to take the different scalability limitations into account. Programs that have short runtimes are not worth parallelizing, because the overhead from the threads can take longer than the code itself. Also, areas that need to be done in a specific order can oftentimes be left in serial because of the communication of data and use of mutual dependencies. Code that consists of many discrete tasks, would be a good candidate to parallelize. In a more practical sense, code that would not be used more than once, can generally afford a longer runtime, and would not need to be parallelized. Unless of course, the runtime is outside of the frame it's needed, then it would be worth the effort. Also, code that consists of many discrete tasks, would be a good candidate to parallelize.

In our library, a majority of our routines consist of more fine grained parallelism than coarse grained. Most of the instructions given to the threads are at the loop level. The matrix vector multiplication is a good example of this. The threads occupy the outer loop, thus partitioning the matrix into smaller instruction sets for each thread. Once the thread has iterated its partition it is then not needed anymore, so we should not see many cache misses occur. In the dot product, we see more medium to coarse grained parallelism. Here, the vectors are generally large, and each thread takes a large portion to create a partial summation. Then a *big fat mutex* is used to get our total scalar. Waiting for the threads can cause us to lose a few cycles as well.

In our library, we saw that the pthread and openmp results were relatively similar. The OpenMP API was helpful in trying to parallelize larger blocks of code. Utilizing the tools were much easier than pthreads, and I found myself testing where I thought parallelism would work with it since the set up was simpler. It also made it easier to synchronize threads when trying to parallelize a larger program. Pthreads requires much more work to set up, but I believe it uses less resources because of this. Pthreads seems to be strong in fine grained parallelism, although it takes more work. Being able to explicitly break apart loops or sections of code to be sent to the threads was much easier in this method, and allowed me to have complete control over the actions of the threads.

Conclusion

The baby blas library's routines were found to have optimal cases for each version. Each routine has proved to have its most efficient call on a case by case basis. Below, we detail the best situation to use each.

For the following routines, when utilizing in parallel, the pthreads version proves to be the best option. This is due to pthreads tending to be best for the fine grained parallelism used. The dot product is the most efficient when used serially, and there's not much of a point of utilizing the parallel versions. Moving to the vvm routine, the serial routine proves to be best at problem sizes up to 6,000. However, at larger sizes I would recommend using the parallel version using only up to 16 threads (if available). When using the mvv routine most efficiently, you should use up to as many threads as you can. The scalability of the mvv allows for the best prolonged speed at higher threads. Matrix multiplication proved to have a lot of parallelism within it, i would use the pthreads routine with 16 threads or more. If that amount of cpu power is not available then the serial version would be the most efficient option.

For the two linear solvers, I recommend using the OpenMP iteration when calling in parallel. This is because of the more coarse grained parallelism being utilized here. The direct linear solver proved best to always be called in parallel. At lower sample sizes the thread counts matter more, but at larger sizes I recommend only to use up to 8 threads. The iterative linear solver was in the same vain, where at smaller sizes the discrepancies between thread counts is large. However, at large sample sizes you would not need to use more than 8 threads.

References

Golub, G. H., & Van_Loan, C. F. (2013). *Matrix computations*. John Hopkins University press.

Burden, R. L., Faires, J. D., & Burden, A. M. (2016). *Numerical analysis*. Cengage Learning.

Hager, G., & Wellein, G. (2010). *Introduction to high performance computing for scientists and engineers (Chapman & Hall/Crc Computational Science series)*. CRC Press.

Nichols, B. (n.d.). *Pthreads programming*. O'Reilly.