ASEN 4057: Final Project

Benjamin Elsaesser[a] Michael Tzimourakas[b]
University of Colorado Boulder, ASEN 4057
Date: 5/4/18

[a]SID: 104665820
[b]SID: 104805696

I. Description of C Program to Improve

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

#define n 1000

double A[n][n];
double B[n][n];
double C[n][n];

void rand_matrix();

int main(int argc, char * argv[])
{
        // Update progress
        printf("Initializing matrices...\n");

        // Iterators
        int i, j, k;

        // Create random matrices
        rand_matrix();

        // Update progress
        printf("Computing matrix product...\n");

        // Begin clock
        clock_t begin, end;
        double time_spent;
        begin = clock();

        // Compute matrix product

        for (i = 0; i < n; i++)
        {
                for (j = 0; j < n; j++)
                {
                        C[i][j] = 0;
                        for (k = 0; k < n; k++)
                        {
                                C[i][j] += A[i][k]*B[k][j];
                        }
                }
        }

        // Stop clock
        end = clock();
        time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
        printf("Time spent computing matrix product: %lf\n",time_spent);

        return 0;
}
```

```
void rand_matrix()
{
        int i, j;
        srand ( time(NULL) );

        for (i = 0; i < n; i++)
        {
                for (j = 0; j < n; j++)
                {
                        A[i][j] = (double) rand() / RAND_MAX;
                        B[i][j] = (double) rand() / RAND_MAX;
                }
        }
}
```

The following code will be improved using different optimization principles. This code contains some many areas that can be improved on to optimize the code. In particular, this code does a poor job of reusing memory when in cache. Instead of reusing memory when reading the rows and columns, they are deleted and read into the memory again. This is one aspect that will be improved. Next, the code does a poor job of hiding memory latency. It must load variables from memory into registers which increases the computation time. The code will be designed to allow for simultaneous memory access and data prefetching. Lastly, the code does not minimize the ratio of memory access to FLOP's. The code will optimize the number of memory accessed per floating point operations so that it does not have to fetch memory so much. All of these improvement will vastly reduce the computation time and overall is better coding structure.

II. Part 1: Serial Code Optimization

A. Profile Report and Discussion of Performance of Original C Code

```
                                {1179953719542355272}
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
100.41      6.35     6.35                             main
  0.32      6.37     0.02        1    20.15    20.15  rand_matrix

 %         the percentage of the total running time of the
time       program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self      the number of seconds accounted for by this
seconds    function alone.  This is the major sort for this
           listing.

calls      the number of times this function was invoked, if
           this function is profiled, else blank.

 self      the average number of milliseconds spent in this
ms/call    function per call, if this function is profiled,
           else blank.

 total     the average number of milliseconds spent in this
ms/call    function and its descendents per call, if this
           function is profiled, else blank.

name       the name of the function.  This is the minor sort
           for this listing. The index shows the location of
           the function in the gprof listing. If the index is
           in parenthesis it shows where it would appear in
           the gprof listing if it were to be printed.
♀
Copyright (C) 2012-2015 Free Software Foundation, Inc.

Copying and distribution of this file, with or without modification,
are permitted in any medium without royalty provided the copyright
notice and this notice are preserved.
♀
                     Call graph (explanation follows)


granularity: each sample hit covers 2 byte(s) for 0.16% of 6.37 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    6.35    0.02                 main [1]
                0.02    0.00       1/1           rand_matrix [2]
-----------------------------------------------
                0.02    0.00       1/1           main [1]
[2]      0.3    0.02    0.00       1         rand_matrix [2]
-----------------------------------------------

 This table describes the call tree of the program, and was sorted by
 the total amount of time spent in each function and its children.

 Each entry in this table consists of several lines.  The line with the
 index number at the left hand margin lists the current function.
 The lines above it list the functions that called this function,
 and the lines below it list the functions this one called.
 This line lists:
                                     Page 1
```

Figure 1: Profile Report Created by Gprof

Figure 1 describes the profile created by C's gprof profiling ability. This profile contains a 'Flat profile' and a 'Call graph' for the original C code. Some key points are described in the profile. First is the computation time. To compute a (1000 x 1000) matrix multiplication, it takes 6.37 seconds listed in the both the Call Graph and Flat profile. Within this code, there is one function call, the random matrix generator. This call takes 0.02 seconds. The time is 20.15 ms/call for this matrix. These times are described and will be improved through the optimization techniques listed above.

B. Proposed Serial Code Improvements

The following methods will be implemented to improve the original C code. The first issue with the code that must be improved is it does a poor job of reusing memory when in cache. This causes the computer to re-access memory that could be reused locally. The memory is not allocated in the correct location and must be reaccessed later which takes time. This is improved by breaking up the multiplication of the matrix into smaller chunks. Making these chunks small enough allows for the computer to keep this memory in the cache rather than committing it to RAM. Keeping it in cache allows the computer to reuse these values, overall improving the code. This process works for a matrix of any size and allows the computer to be efficient in its use of cache.

Next, the code does a poor job of hiding memory latency. It takes time to load variables from memory into the registers and ultimately the Floating Point Unit and Arithmetic Logic Unit. This time can be reduced by reading in some initial variables, computing with those initial variables and reading in the next variables. Doing this process rather than computing all variables at once reduces the process time significantly.

Lastly, the final issue the code contains is the code does not minimize the ratio of memory access to FLOPs. This means it accesses memory at a high rate relative to each FLOP. For every add/multiply, there are three accesses to memory. Reducing the amount of time the memory is accessed reduces this ratio and will improve the computation speed. This can be done by changing the nesting of the for loop and computing areas simultaneously. This more efficient code computes multiple in the for loop at the same time instead of computing everything in the location of the loop over and over again. Utilizing this improvement reduces the calls to memory and increases the ability of the FLOPs of the computer.

Overall, implementing all of these optimization methods for computing a (1000 x 1000) matrix multiplication should improve the computation time greatly and overall create a more efficient code for the computer to process.

C. Timing Comparison of Original Code and Optimized C Code

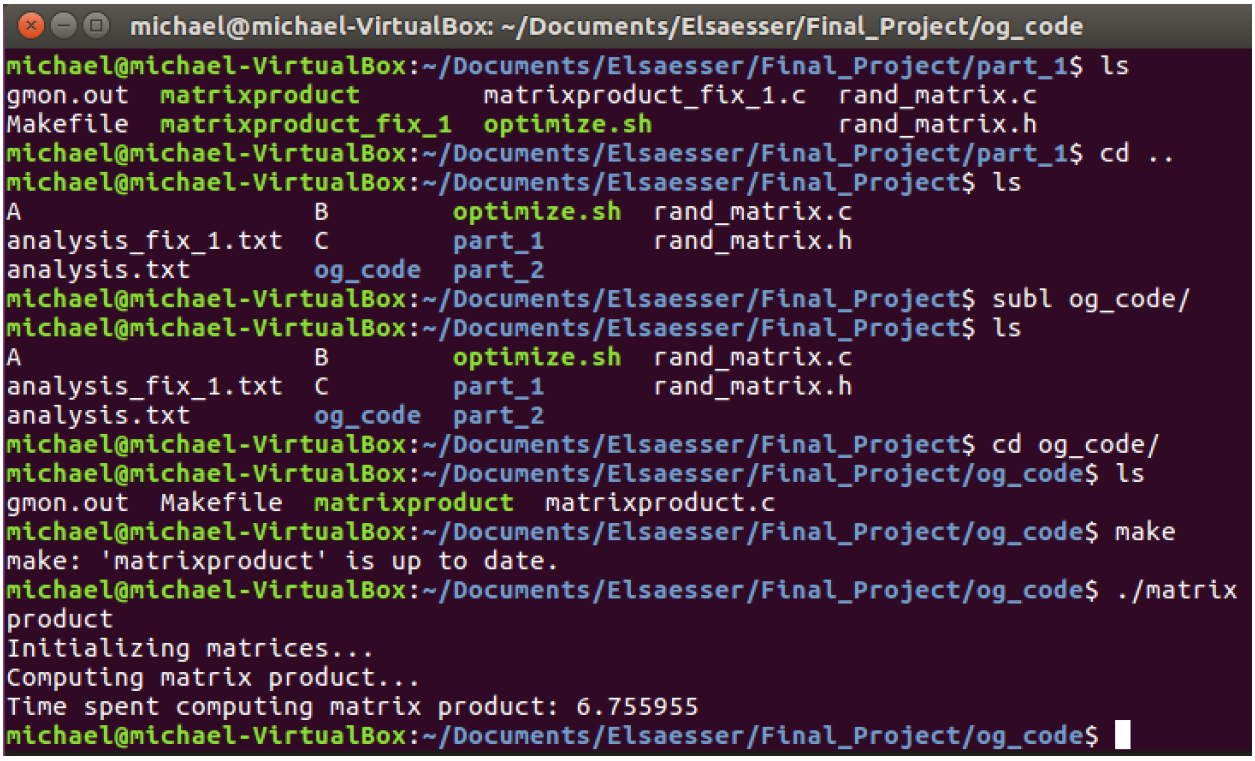

Figure 2: The computation time of original code

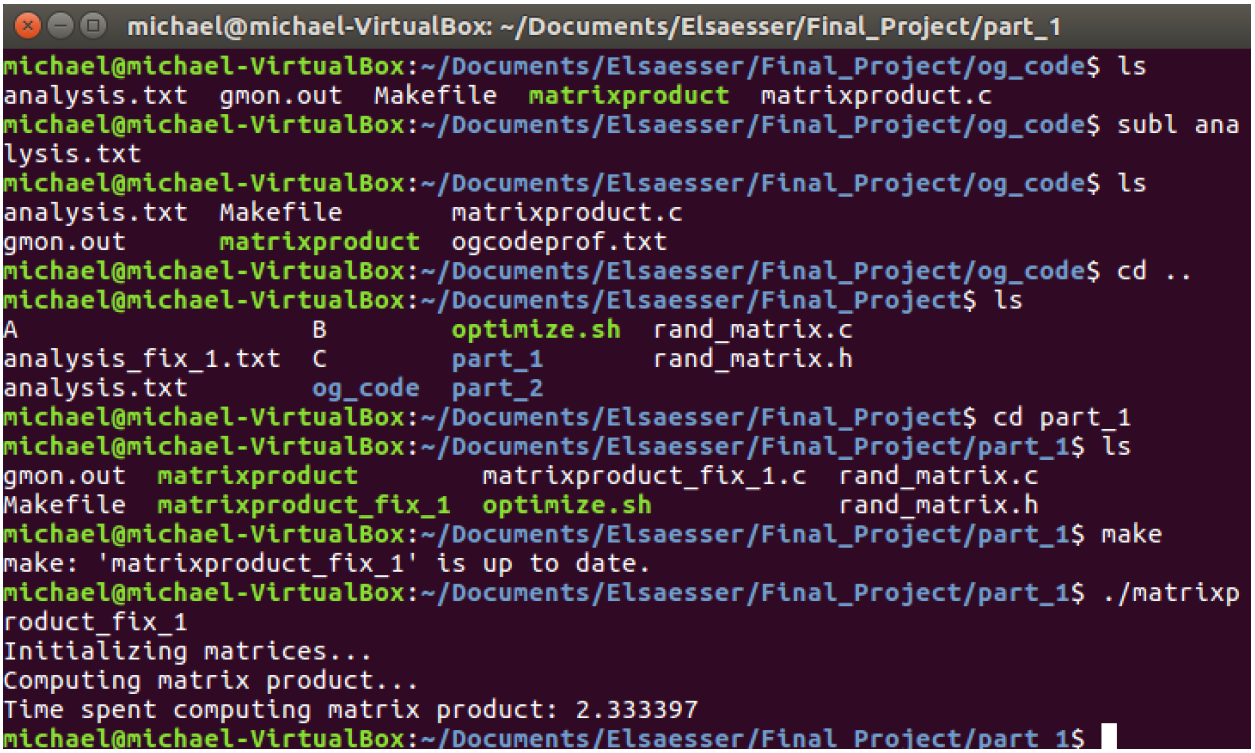

Figure 3: The computation time of improved code

The previous two figures compare the computation time of the original C code, and the improved C code. As you can see, implementing the serial code improvements vastly increased the computation speed of the matrix multiplication code. The code improved from 6.7559 seconds to 2.333 seconds. This is a 289.58 % increase in speed. Now this may not seem like much, but this is vastly better and overall is less intensive on the computer running the program. As the amount of computation the code increases, this difference in time will grow. These techniques can also be applied to different coding languages and many different types of programs.

D. Profile Report and Discussion of Performance of Optimized C Code

```
                                {1016698349015024068}.txt
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 99.89      2.28     2.28                             main
  0.44      2.29     0.01        1    10.03    10.03  rand_matrix

 %         the percentage of the total running time of the
time       program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self      the number of seconds accounted for by this
seconds    function alone.  This is the major sort for this
           listing.

calls      the number of times this function was invoked, if
           this function is profiled, else blank.

 self      the average number of milliseconds spent in this
ms/call    function per call, if this function is profiled,
           else blank.

 total     the average number of milliseconds spent in this
ms/call    function and its descendents per call, if this
           function is profiled, else blank.

name       the name of the function.  This is the minor sort
           for this listing. The index shows the location of
           the function in the gprof listing. If the index is
           in parenthesis it shows where it would appear in
           the gprof listing if it were to be printed.
♀
Copyright (C) 2012-2015 Free Software Foundation, Inc.

Copying and distribution of this file, with or without modification,
are permitted in any medium without royalty provided the copyright
notice and this notice are preserved.
♀
                     Call graph (explanation follows)


granularity: each sample hit covers 2 byte(s) for 0.44% of 2.29 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    2.28    0.01                 main [1]
                0.01    0.00       1/1           rand_matrix [2]
-----------------------------------------------
                0.01    0.00       1/1           main [1]
[2]      0.4    0.01    0.00       1         rand_matrix [2]
-----------------------------------------------

 This table describes the call tree of the program, and was sorted by
 the total amount of time spent in each function and its children.

 Each entry in this table consists of several lines.  The line with the
 index number at the left hand margin lists the current function.
 The lines above it list the functions that called this function,
 and the lines below it list the functions this one called.
 This line lists:
                                     Page 1
```

Figure 4: Profile Report Created by Gprof

The previous figure is the profile report created by the improved C code. This profile report really describes how much more efficient the new code is when compared to the old one. For this new code, the total time takes 2.29 seconds vs the original 6.37 seconds. This total time improvement is significantly more efficient. This improvement also applies to the children functions used. The 'rand_matrix' function call takes 0.01 seconds compared to the 0.02 seconds prior. This is half as long and is evidence the techniques used reduce the computation time. Now this may not seem like much, but when other programs that require many function calls are improved, the overall time will be reduced greatly. Lastly, the total ms/call time was improved to 10.03 ms/call vs the original 20.15 ms/call. This is less than half the computation time and vastly improves the code. The evidence that the serial code improvements have created a more efficient code are throughout this report and these techniques should be applied to other C codes.

III. Part 2: Parallelization of Optimized Code

A. Proposed Strategy for Code Parallelization

The second part of this project was to parallelize the optimized code created. Parallelizing the code means using multiple processors of the computer to perform multiple tasks at the same time, or in parallel. Using a library called Open MP, the processors the number of processors utilized can be specified. Each additional processor used increases the total possible number of FLOPS. However, there are a few bottlenecks that can actually make parallelized code run slower than the non-parallelized code. The first of which being the time used to communicate between the processor. The problem that we expect this code to suffer from is that of different processors attempting to access the same memory location at the same time. This is expected because our optimization approach involved using nested for loops to create discreet blocks in the matrices whose values can be solved for directly. The problem with this approach is that the same locations in the A and B matrices must be accessed in multiple iterations of the loops. For this reason the team decided to parallelize the outermost for loop in the blocking process. What this corresponds to is each processor computing a set number of rows (ib) in the C matrix.

B. Scalability Study on Virtual Machine with up to Eight Processes

In order to determine the scalability, or how well it can utilize extra processors, two 1000x1000 matrices were multiplied together using 1, 2, 4, and 8 processors. The time the matrix multiplication took in each of these cases was recorded, resulting in table 1. These results show that the parallelized code runs slower than the code using only one processor. Furthermore, the second fastest time was recorded using two processors and the third fastest time was with 8 processors, making the run with 4 processors the slowest. For the reasons outlined in section IIIA, it is expected that the non-parallelized would be the fastest. The strangest result out of these four is that the run with 8 processors is faster than the one with 4 processors. Running with different block sizes shows that the speed of using a different number of processors is heavily dependent on the block sized used in this case. This makes sense because the amount of overlap between the memory addresses accessed by each processor is very dependent on the geometry of how the matrices are blocked and the number of processors. In general, however, the time difference between running the program with 4 and 8 processors is less than 0.1 seconds. It is important to note that the scalability study and the profiling of the optimized code were run on separate computers, which is the cause for the discrepancy between the timing of the one processor run and the regular optimized code timing.

Number of Processors	Time to Run [s]
1	3.364761
2	3.411092
4	4.105691
8	4.076283

Table 1: Scalability Study on Optimized and Parallelized Code

C. Profile Report and Discussion of Performance on Virtual Machine

Below is a profile summary of the optimized and parallelized code using gprof. The results of profiling the parallelized code is not very useful. This is because the profiler only profiles one processor, which would not give a good representation of the entire code's performance. For this reason, the timing found in 1 is the best metric for code performance that we have.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 99.81      2.08     2.08                             main
  0.48      2.09     0.01        1    10.03    10.03  rand_matrix

 %         the percentage of the total running time of the
time       program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self      the number of seconds accounted for by this
seconds    function alone.  This is the major sort for this
           listing.

calls      the number of times this function was invoked, if
           this function is profiled, else blank.

 self      the average number of milliseconds spent in this
ms/call    function per call, if this function is profiled,
           else blank.

 total     the average number of milliseconds spent in this
ms/call    function and its descendents per call, if this
           function is profiled, else blank.

name       the name of the function.  This is the minor sort
           for this listing. The index shows the location of
           the function in the gprof listing. If the index is
           in parenthesis it shows where it would appear in
           the gprof listing if it were to be printed.
^L
Copyright (C) 2012-2015 Free Software Foundation, Inc.

Copying and distribution of this file, with or without modification,
are permitted in any medium without royalty provided the copyright
notice and this notice are preserved.
^L
                     Call graph (explanation follows)


granularity: each sample hit covers 2 byte(s) for 0.48% of 2.09 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    2.08    0.01                 main [1]
```

```
                0.01    0.00       1/1           rand_matrix [2]
-----------------------------------------------
                0.01    0.00       1/1           main [1]
[2]      0.5    0.01    0.00       1          rand_matrix [2]
-----------------------------------------------

 This table describes the call tree of the program, and was sorted by
 the total amount of time spent in each function and its children.
```

IV. Summary of Findings and Potential Future Improvements

This project proved the abilities of improving processes in C and utilizing external libraries to optimize code. Utilizing data structure techniques and implementing strategies to reduce the amount of work the computer must do resulted in vastly improving the original matrix multiplication code.

First, simply improving the structure of the code improved the time required to compute a (1000 x 1000) matrix multiplication example. Reusing memory in cache, hiding memory latency, and reducing the memory access to FLOPS ratio were all methods of improving the original code. These original methods of reducing the computation time proved to be the most effective in reducing the time of calculating the matrix multiplication. These methods reduced the run time from 6.37 seconds to 2.29 seconds. This is a 278.17 % improvement in computation speed and vastly improves the code.

Next, parallel processing was utilized in an attempt to further optimize the already optimized code. Utilizing the library Open MP, the computer dedicated more processors to the execution of the code. It turned out that parallelizing this code actually made it run slower than before, going from a 2.29 second run time to about 4 seconds depending on how many processors are used. This helps to show the importance of thinking through the architecture of your code and how parallelization will effect it before attempting to adapt the code to use multiple processors.

To improve this code, the optimization strategy should be changed to make parallelization more effective. This could be done in multiple ways, one way could be to use an external library like lapacke to multiply each row of the A matrix with each column of the B matrix to populate the C matrix. This would make it so that each loop is accessing a separate part of each matrix, getting rid of the memory overlap issue. Another solution to this problem would be to re-organize the blocking used here to get rid of the memory overlap issue.

V. Appendix

A. Optimized Code: matrixproduct_fix_1.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include "rand_matrix.h"

#define n 1000

double A[n][n];
double B[n][n];
double C[n][n];


int main(int argc, char * argv[])
```

```
{
        // Update progress
        printf("Initializing matrices...\n");

        // Iterators
        int i, j, k, ii, kk;
        // Block sizes
        int ib = 20;
        int kb = 50;

        // Variables for data prefetching
        double a00, a01, a10, a11, a20, a21, a30, a31;

        // Create random matrices
        rand_matrix();

        // Update progress
        printf("Computing matrix product...\n");

        // Begin clock
        clock_t begin, end;
        double time_spent;
        begin = clock();

        ///// Compute matrix product

        // Loop through first block
        for (ii = 0; ii < n; ii += ib)
        {
                // Loop through second block
                for (kk = 0; kk < n; kk += kb)
                {
                        // Loop through columns iterating by 2
                        for (j = 0; j < n; j += 2)
                        {
                                // Loop through rows iterating by 4
                                for (i = ii; i < ii + ib; i += 4)
                                {
                                        // Check if it's the first time looping through the
                                        // Set prefetching variables = 0 if true
                                        if (kk == 0)
                                                a00 = a01 = a10 = a11 = a20 = a21 = a30 = a

                                        // Else set prefetching variables equal to the 4x2
                                        // in the result matrix, C
                                        else
                                        {
                                                a00 = C[i][j];
                                                a01 = C[i][j+1];
                                                a10 = C[i+1][j];
                                                a11 = C[i+1][j+1];
                                                a20 = C[i+2][j];
                                                a21 = C[i+2][j+1];
                                                a30 = C[i+3][j];
                                                a31 = C[i+3][j+1];
```

```
                                        }

                                        // Loop through kb columns in the A matrix and rows
                                        // Uses the 4x2 block to multiply
                                        for (k = kk; k < kk + kb; k++)
                                        {
                                                a00 += A[i][k]*B[k][j];
                                                a01 += A[i][k]*B[k][j+1];
                                                a10 += A[i+1][k]*B[k][j];
                                                a11 += A[i+1][k]*B[k][j+1];
                                                a20 += A[i+2][k]*B[k][j];
                                                a21 += A[i+2][k]*B[k][j+1];
                                                a30 += A[i+3][k]*B[k][j];
                                                a31 += A[i+3][k]*B[k][j+1];
                                        }

                                        // Reassign the result matrix with the prefetched
                                        C[i][j] = a00;
                                C[i][j+1] = a01;
                                C[i+1][j] = a10;
                                C[i+1][j+1] = a11;
                                C[i+2][j] = a20;
                                C[i+2][j+1] = a21;
                                        C[i+3][j] = a30;
                                        C[i+3][j+1] = a31;
                                }
                        }
                }
        }

        // Stop clock
        end = clock();
        time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
        printf("Time spent computing matrix product: %lf\n",time_spent);

        return 0;
}
```

B. Parallelized Code: matrixproduct_mpi.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <omp.h>
#include "rand_matrix.h"

#define n 1000

double A[n][n];
double B[n][n];
double C[n][n];


int main(int argc, char * argv[])
{
```

```
// Update progress
printf("Initializing matrices...\n");

// Iterators
int i, j, k, ii, kk;
// Block sizes
int ib = 20;
int kb = 50;
int n_loops = n;

// Variables for data prefetching
double a00, a01, a10, a11, a20, a21, a30, a31;

// Parallization variables
int nprocs = atoi(argv[1]);

// Create random matrices
rand_matrix();

// Update progress
printf("Computing matrix product...\n");

// Begin clock
clock_t begin, end;
double time_spent;
begin = clock();

///// Compute matrix product
omp_set_num_threads(nprocs);
int sched = n_loops/(ib*nprocs);


// Loop through first block
#pragma omp parallel
{
        #pragma omp for private(i,j,k,kk,ii,a00,a01,a10,a11,a20,a21,a30,a31) sched
        for (ii = 0; ii < n_loops; ii += ib)
        {
                // Loop through second block
                for (kk = 0; kk < n_loops; kk += kb)
                {
                        // Loop through columns iterating by 2
                        for (j = 0; j < n_loops; j += 2)
                        {
                                // Loop through rows iterating by 4
                                for (i = ii; i < ii + ib; i += 4)
                                {
                                        // Check if it's the first time looping thr
                                        // Set prefetching variables = 0 if true
                                        if (kk == 0)
                                                a00 = a01 = a10 = a11 = a20 = a21 =

                                        // Else set prefetching variables equal to
                                        // in the result matrix, C
                                        else
```

```
                                        {
                                                a00 = C[i][j];
                                                a01 = C[i][j+1];
                                                a10 = C[i+1][j];
                                                a11 = C[i+1][j+1];
                                                a20 = C[i+2][j];
                                                a21 = C[i+2][j+1];
                                                a30 = C[i+3][j];
                                                a31 = C[i+3][j+1];
                                        }

                                        // Loop through kb columns in the A matrix
                                        // Uses the 4x2 block to multiply
                                        for (k = kk; k < kk + kb; k++)
                                        {
                                                a00 += A[i][k]*B[k][j];
                                                a01 += A[i][k]*B[k][j+1];
                                                a10 += A[i+1][k]*B[k][j];
                                                a11 += A[i+1][k]*B[k][j+1];
                                                a20 += A[i+2][k]*B[k][j];
                                                a21 += A[i+2][k]*B[k][j+1];
                                                a30 += A[i+3][k]*B[k][j];
                                                a31 += A[i+3][k]*B[k][j+1];
                                        }

                                        // Reassign the result matrix with the pre
                                        C[i][j] = a00;
                                        C[i][j+1] = a01;
                                C[i+1][j]= a10;
                                C[i+1][j+1] = a11;
                                C[i+2][j] = a20;
                                C[i+2][j+1] = a21;
                                        C[i+3][j] = a30;
                                        C[i+3][j+1] = a31;
                                }
                        }
                }
        }

        // Stop clock
        end = clock();
        time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
        printf("Time spent computing matrix product: %lf\n",time_spent);

        return 0;
}
```