Uppsala University

HIGH PERFORMANCE AND PARALLEL COMPUTING

Final Project Report

Author:
Salman Khan

May 28, 2021

1 Abstract

This final project of high performance and parallel computing is a mandatory part of the course. The main goal of the project is to use all optimization and parallelization techniques that we learned during the course. We were allowed to choose a problem of our choice and then to write a code, optimize it and then parallelize the code to make the performance better.

In this report I explained the LU_factorization algorithm and its implementation along with optimization and parallelization methods. My main focus was to achieve great performance along with the correctness of the results because getting correct results are more important than any optimization and parallelization efforts.

Using parallelization methods along with optimization techniques, the performance and computation of the task was improved by 35.39%. Please note that this performance is measured for the whole program using `time` command and not for only the function where parallelization techniques are used.

2 Introduction

High performance and parallel computing opens the entryways for tackling various interesting scientific and engineering issues. In order for a High performance and parallel computing project to achieve the required results, the software engineer must be aware of the knowledge of coding optimization, parallel programming and load adjusting issues.

In this project I decided to work on LU-factorization. I worked on implementation of an algorithm, optimization of the code and then parallelization with OpenMP. For optimization both compiler and manually techniques were applied and we chooses the one which gives the best result.

2.1 What is LU-Factorization?

In linear variable based math and numerical analysis, lower–upper (LU) decay or factorization factors a matrix as the item of a lower triangular and an upper triangular matrix. Sometime it is also called a permutation matrix. LU decay can be seen as the matrix form of Gaussian end. Computers as a rule solve square frameworks of linear equations utilizing LU decomposition, and it is additionally a key step when rearranging a matrix or computing the determinant of a matrix. In the lower triangular matrix all elements above the diagonal are zero, in the upper triangular matrix, all the elements below the diagonal are zero. For example, for a 3 × 3 matrix A, its LU decomposition is shown in figure 1.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} \ell_{11} & 0 & 0 \\ \ell_{21} & \ell_{22} & 0 \\ \ell_{31} & \ell_{32} & \ell_{33} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}.$$

Figure 1: Showing 3 × 3 matrix, its LU decomposition looks like this.

The easy way is to start with making upper triangular matrix by doing elementary row operations on a matrix and always store each scalar number K, which we will be using in each elementary row operation, as shown below in equation 1. The storing of these K values is important, because we will be making lower triangular matrix using these stored values.

Replace Ri by Ri - K(Rj) — Eq (1)

Where Ri and Rj are different rows and K is scalar quantity.

Why we need LU-Factorization? A lot of matrix operations are much easier using triangular matrices. By much easier I mean it decreases the time complexity of the problem, which increase the performance of the system. In case, if we like to do a lot of calculations on a specific matrix, getting its LU decay will likely speed things up.

2.2 Optimization.

After composing a program which tackles the issue, our next task was to use the optimization procedures to form the execution superior. Optimization comes about in a more ideal arrangement and makes the execution much way better compared to the naive code. In this report we clarified the result of distinctive optimized functions and the strategies utilized for optimizations. Our primary center was to realize incredible execution, we chose the most excellent optimization strategy for each work. Along with manual optimization techniques we tried some of the compiler optimization flags as well like `-O1, -O2, -O2 -funroll-all-loops, -O3, -O3 -funroll-all-loops and -O3 -ffast-math.`

2.3 Parallelization.

After optimization our next task was to do parallelization of the code. By parallelization, I mean to do multi-threading tasks. We can do it either by using OpenMP or Pthreads. I choose OpenMP because it is more portable and doesn't constrain you to using C and it can be as straightforward as including a single pragma, and you will be 90% of the way to appropriately multithreaded code with straight speedup. To induce the same execution boost with pthreads needs more work.

2.3.1 OpenMP.

On a variety of platforms, instruction-set architectures, and operating systems, OpenMP supports multi-platform shared-memory multiprocessing programming in C, C++, and Fortran. It is made up of a collection of compiler

instructions, library routines, and environment variables that control the behavior of the program at runtime.

`#pragma omp parallel` command is used for threading creation using Openmp.

3 System Specifications.

It is very important to check system specifications as it matter the most when it comes to optimization and parallelization of coding. By using the command `lscpu` we got the following specifications of the system on which we ran our program.

- Architecture: x86_64
- CPU op-mode(s): 32-bit, 64-bit
- Byte Order: Little Endian
- CPU(s): 16
- On-line CPU(s) list: 0-15
- Thread(s) per core: 2
- Core(s) per socket: 4
- Socket(s): 2
- NUMA node(s): 2
- Vendor ID: GenuineIntel
- CPU family: 6
- Model: 26
- Model name: Intel(R) Xeon(R) CPU E5520 @ 2.27GHz
- Stepping: 5
- CPU MHz: 1600.126
- CPU max MHz: 2268,0000
- CPU min MHz: 1600,0000

- BogoMIPS: 4533.72
- Virtualization: VT-x
- L1d cache: 32K
- L1i cache: 32K
- L2 cache: 256K
- L3 cache: 8192K
- NUMA node0 CPU(s): 0-3,8-11
- NUMA node1 CPU(s): 4-7,12-15

4 The Solution.

4.1 Data Structure Used.

The data structure used in the code is simple 2D arrays.Arrays allow random access and require less memory per element while lacking efficiency for insertion/deletion operations as compared to other data structures but here in this project we are not updating or deleting elements within an array and we are after achieving great performance with using less memory. Also, arrays are easy to implement and the code seems more simple and understandable. Arrays make it easy to do mathematical operations.

4.2 Algorithms Used.

Different functions for different purposes were defined and called in the main function. Each function is explained below:

- `void create()`: This function purpose is to allocate a fixed sized matrix. Which I used for LU factorization.
- `void fill()`: This function is used to fill the elements in the original matrix.
- `void decompose()`: This is the most important function in my project. It is used to do the decomposition of a matrix into upper and lower triangular matrix.

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

        if(argc < 2) {
                printf("./sequential <size>!\n");
                return 1;
        }
        double **a, **l, **u, **ver, **atemp;
        struct timeval start_point, end_point;
    double elapsed_time;

        int size  = atoi(argv[1]);

        create(&a, &l, &u, &ver, &atemp, size);
        fill(a, u, atemp, size);
//wall clock time of the function
        gettimeofday(&start_point, NULL);
        decompose(a, l, u, size);
        gettimeofday(&end_point, NULL);
        ////////////
        elapsed_time = (end_point.tv_sec - start_point.tv_sec) * 1e6;
    elapsed_time = (elapsed_time + (end_point.tv_usec -  start_point.tv_usec)) * 1e-6;

        int correct = correctLU(l, u, atemp, ver, size);
        FILE * fp = fopen("result.txt", "a+");
        if(correct) {
                fprintf(fp, "%d ", size);
                fprintf(fp, "%.6f ", elapsed_time);
                fprintf(fp, "%s\n", "MATCH");
        }
        else {
                fprintf(fp, "%d ", size);
                fprintf(fp, "%.6f ", elapsed_time);
                fprintf(fp, "%s\n", "NOT MATCH");
        }
        fclose(fp);
//we need to destoru the matrix  for preventing memory leakeage
        destroy(a, size);
        destroy(l, size);
        destroy(u, size);
        destroy(ver, size);
        destroy(atemp, size);

        return 0;
}
```

Figure 2: Showing the main function of the program where the different functions was called.

- `int correctLU()`: This function is used to check the correctness of the LU-factorization. This is crosscheck between what we found and what we already have.
- `void destroy()`: The purpose of this function is delete all the data from the memory once my calculations are done. This is used to avoid memory leakage.

Memory leaks are a class of bugs where the application fails to release memory when no longer needed. Over time, memory leaks affect the performance of both the particular application as well as the operating system. A large leak might result in unacceptable response times due to excessive paging.

Some pre-defined functions are also used to write the results to a file and check wall clock time of the functions. The resultant file shows us if the results are correct or not by mentioned MATCHED or NOTMATCHED.

5 Performance and Discussion.

All times were measured using the `time` command. As an example the command for sequential code is shown below:

```
time ./compiled_file 2000
```

Where 2000 is the size of sq.matrix.

Below are some of the important definition of the out of time command:

- `Real time` is wall clock time - time from start to finish of the call.
- `User time` is the sum of time the CPU was active executing code in user space.
- `System time` is the sum of time the CPU was active executing code in kernel space.

The performance of the program is discussed in the similar manner as we did in the course. Like first I discussed the performance of the sequential code, then optimized and finally parallelized code.

5.1 Sequential.

Before trying optimization and parallelization on the code written for the LU-factorization, it is very important to check the performance of the naive code for different sizes of matrices. This helped us in comparing the time taken by different sizes matrices in optimization and parallelization processes.

As the size of the matrix increases the real and user time taken by the program is also increases as show in the below table and graphs. There is a

drastic increase in real and user time taken when the matrix size is increase by 2000 as compared to 1000. Please note that for matrix size 2000 the system is also greater as compared to other size matrices because the the kernel took long to perform the operation.

	Square Matrix Size															
Time Taken	32		64		100		150		250		500		1000		2000	
	real	0m0,036s	real	0m0,055s	real	0m0,054s	real	0m0,110s	real	0m0,284s	real	0m1,944s	real	0m21,064s	real	3m53,694s
	user	0m0,002s	user	0m0,007s	user	0m0,020s	user	0m0,049s	user	0m0,226s	user	0m1,890s	user	0m20,977s	user	3m53,403s
	sys	0m0,000s	sys	0m0,000s	sys	0m0,000s	sys	0m0,004s	sys	0m0,004s	sys	0m0,004s	sys	0m0,028s	sys	0m0,068s

Figure 3: Shows the table which contain real and system time taken by different sizes matrices.

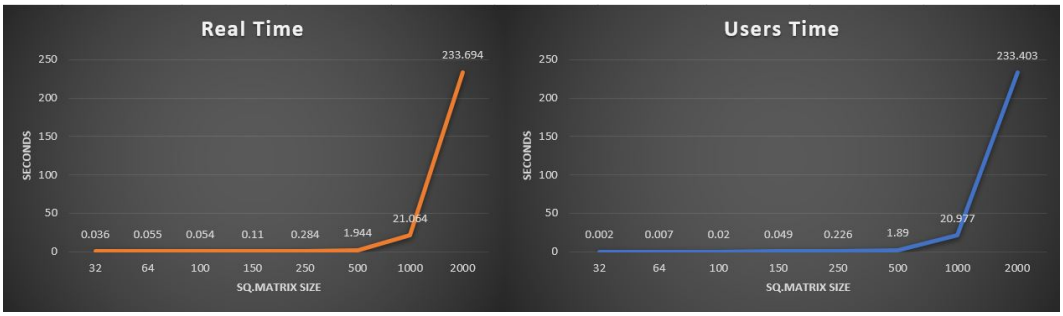


Figure 4: Shows the graphs for real and system time taken by different sizes matrices. Lift hand side graph is showing real time taken by the code and right hand graph shows user time taken by the code.

5.2 Optimization.

Here is this phase, I tried different optimization techniques along with manual optimization. These methods are defind below:

5.2.1 -O1 Flag.

By using this flag the compiler tries to decrease code size and execution time without performing any optimizations that take a extraordinary deal of compilation time. This flag increase the performance as compared to sequential coding.

5.2.2 -O2 Flag.

Compile tries to optimize as much as possible while keeping the executable the same size as without an optimization flag. Here the performance is a bit increase as compared to -O1 flag.

5.2.3 -O3 Flag.

This compiler flag will optimize without regard to the size of the executable or the time for the compilation. This compiler flag perform a bit better than the other two mentioned above. See the values in the below table.

	Square Matrix Size							
Flags	32	64	100	150	250	500	1000	2000
-O1	0.038	0.042	0.046	0.061	0.162	0.804	12.567	157.131
-O2	0.033	0.041	0.047	0.049	0.152	0.604	12.246	157.81
-O3	0.028	0.042	0.04	0.045	0.15	0.581	12.04	155.643

Figure 5: Shows the table for real time in seconds for compiler flags -O1, -O2 & -O3

5.2.4 -funroll-all-loops.

The flag -funroll-loops tells the compiler to unroll loops instead of doing it manually. Similarly to the manual unrolling -funroll-loops did not have a positive impact on the performance, which might be because of branch prediction. This flag did not perform as expected. This flag is not even not better than -O1 flag. Please see the below table and graph in figure 6 and 7 respectively.

5.2.5 -ffast-math.

This compiler flag is used for the optimizations of mathematical operations. This flag did not showed a good performance. you can see the results obtained for this flags by using different size of matrices in the below table and graph in figure 6 and 7.

5.2.6 -O3 -funroll-all.

We can also combine different flags with each other to use it for the optimization of our code. Here I combined `-O3` flag with `-funroll-all` flag and it showed a great performance achievement. As you can see in the below graph the `-O3 -funroll-all` flag increase the performance of the program as compared to other flags.

Compiler Flags	Square Matrix Size							
	32	64	100	150	250	500	1000	2000
-O1	real 0m0,038s user 0m0,002s	real 0m0,042s user 0m0,000s	real 0m0,046s user 0m0,009s	real 0m0,061s user 0m0,022s	real 0m0,162s user 0m0,116s	real 0m0,804s user 0m0,744s	real 0m12,567s user 0m12,479s	real 2m37,131s user 2m36,956s
-O2	real 0m0,033s user 0m0,002s	real 0m0,041s user 0m0,000s	real 0m0,047s user 0m0,006s	real 0m0,049s user 0m0,015s	real 0m0,152s user 0m0,115s	real 0m0,604s user 0m0,561s	real 0m12,264s user 0m12,205s	real 2m37,810s user 2m37,356s
-O3	real 0m0,028s user 0m0,002s	real 0m0,042s user 0m0,002s	real 0m0,040s user 0m0,005s	real 0m0,052s user 0m0,007s	real 0m0,155s user 0m0,116s	real 0m0,581s user 0m0,543s	real 0m12,049s user 0m11,981s	real 2m35,643s user 2m35,484s
-funroll-all-loops	real 0m0,031s user 0m0,002s	real 0m0,034s user 0m0,007s	real 0m0,050s user 0m0,019s	real 0m0,156s user 0m0,121s	real 0m0,270s user 0m0,237s	real 0m1,920s user 0m1,889s	real 0m21,148s user 0m21,080s	real 3m55,948s user 3m55,712s
-ffast-math	real 0m0,027s user 0m0,000s	real 0m0,037s user 0m0,005s	real 0m0,056s user 0m0,014s	real 0m0,082s user 0m0,051s	real 0m0,415s user 0m0,370s	real 0m1,919s user 0m1,894s	real 0m21,322s user 0m21,245s	real 3m54,175s user 3m53,707s
-O3 -funroll-all-loops	real 0m0,036s user 0m0,002s	real 0m0,039s user 0m0,000s	real 0m0,037s user 0m0,005s	real 0m0,042s user 0m0,013s	real 0m0,149s user 0m0,111s	real 0m0,575s user 0m0,534s	real 0m12,006s user 0m11,948s	real 2m32,473s user 2m32,272s
-O3 -funroll-loops	real 0m0,020s user 0m0,001s	real 0m0,021s user 0m0,002s	real 0m0,033s user 0m0,006s	real 0m0,032s user 0m0,013s	real 0m0,134s user 0m0,112s	real 0m0,569s user 0m0,531s	real 0m11,831s user 0m11,775s	real 2m31,952s user 2m31,797s

Figure 6: Shows the table for real time taken by all the above mentioned compiler flags.

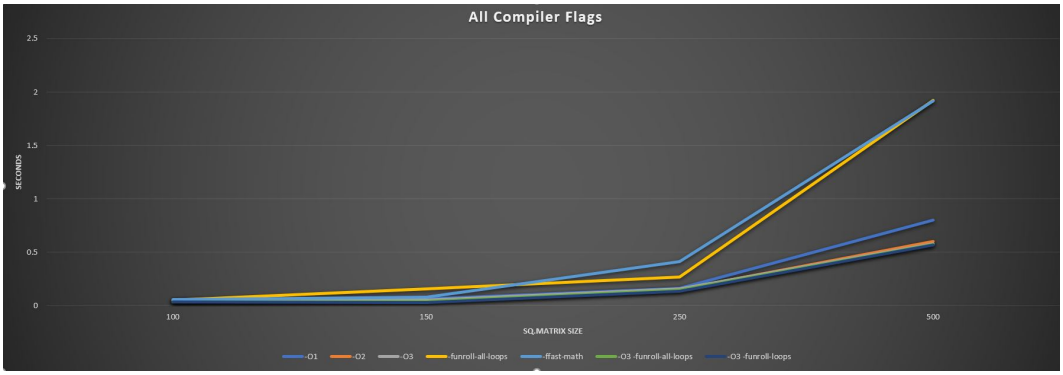


Figure 7: Graph shows the comparison between different compiler flags.

5.2.7 Manually Optimization Using Unroll Factor 2.

In this phase we are required to optimize the manually without any compiler help. As this project is about LU-factorization so, we used three nested loops

for our operation to do decomposition and then to check our results. It is hard to optimize such nested loops and I tried unrolling all the loops manually but but it was impossible to do it for loops inside functions like decompose() and correctLU(). Upon checking the results it was not matching. Then I tried to use unfactor 2 for simple loop inside function fill() and it worked, also the result of the decomposes matrices was matched with the copy of the original matrix.

However, the performance improved was not as good as `-O3 -funroll-all` flag. Please see the below graph and table. So, as a result of these comparison I will moving forward with -O3 -funroll-loops flag optimization and will be doing parallelization of the code.

Compiler Flags	Square Matrix Size															
	32		64		100		150		250		500		1000		2000	
-O3 -funroll-loops	real	0m0,020s	real	0m0,021s	real	0m0,033s	real	0m0,032s	real	0m0,134s	real	0m0,569s	real	0m11,831s	real	2m31,952s
	user	0m0,001s	user	0m0,002s	user	0m0,006s	user	0m0,013s	user	0m0,112s	user	0m0,531s	user	0m11,775s	user	2m31,797s
Manually optimization	real	0m0,034s	real	0m0,047s	real	0m0,043s	real	0m0,079s	real	0m0,264s	real	0m1,947s	real	0m20,858s	real	3m57,080s
	user	0m0,002s	user	0m0,007s	user	0m0,020s	user	0m0,052s	user	0m0,230s	user	0m1,890s	user	0m20,791s	user	3m56,807s

Figure 8: Shows the taken taken by Real and Users by manually and compiler optimization flag.

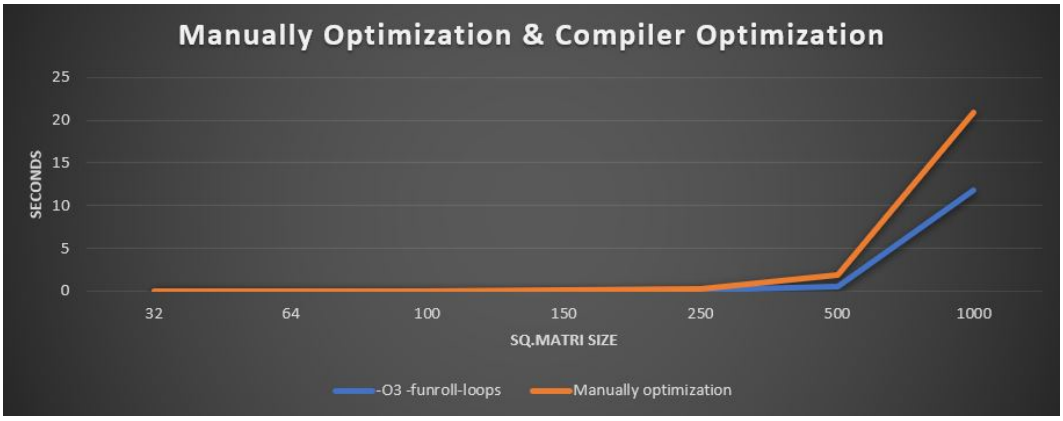


Figure 9: Shows the performance comparison between -O3 -funroll-loops flag and manually optimized code

5.3 Parallelization.

We used `#pragma omp` for directive for the creation of threads here. This directive instructs the compiler to divide loop iterations within the team of threads that comes across this work-sharing structure.

Here, I parallelized one of our most important and time consuming functions known as `decompose()` and another consuming and performance impacting function is `decompose(correctLU())` but I did not parallelize it due variable dependencies. I used OpenMP for adding threading to our code as this is what I choosed but one can also do it using pthread method. Along with `#pragma omp` for directive we use the schedule clause to specify scheduling and to avoid load balancing problem.

The two schedule clauses used are static and guided.

5.3.1 Using Static Schedule.

However, a static schedule might not be the best option. When different iterations take different amounts of time. Figure 10 shows the function where we used static schedule. If you want to change the schedule type just change the static to dynamic or guided. Figure 11 shows a table of different number of threads taking different amount of time for different matrices.

5.3.2 Using Guided Schedule

The aim of dynamic/guided scheduling is to maximize job distribution when each loop iteration does not involve the same amount of work.

We improved the performance of the function by changing the schedule to "guided". Please see the below data.

	Square Matrix Size							
No Of threads	32	64	100	150	250	500	1000	2000
2	real 0m0,027s	real 0m0,028s	real 0m0,038s	real 0m0,047s	real 0m0,146s	real 0m0,575s	real 0m11,736s	real 2m32,754s
	user 0m0,005s	user 0m0,006s	user 0m0,010s	user 0m0,015s	user 0m0,121s	user 0m0,580s	user 0m11,818s	user 2m34,509s
6	real 0m0,028s	real 0m0,030s	real 0m0,035s	real 0m0,042s	real 0m0,129s	real 0m0,520s	real 0m11,615s	real 2m30,989s
	user 0m0,007s	user 0m0,019s	user 0m0,021s	user 0m0,039s	user 0m0,119s	user 0m0,514s	user 0m11,584s	user 2m30,065s
8	real 0m0,029s	real 0m0,036s	real 0m0,034s	real 0m0,047s	real 0m0,147s	real 0m0,563s	real 0m11,759s	real 2m33,415s
	user 0m0,020s	user 0m0,019s	user 0m0,032s	user 0m0,047s	user 0m0,164s	user 0m0,593s	user 0m11,696s	user 2m36,094s
16	real 0m0,034s	real 0m0,038s	real 0m0,036s	real 0m0,047s	real 0m0,149s	real 0m1,322s	real 0m13,647s	real 2m34,436s
	user 0m0,088s	user 0m0,100s	user 0m0,107s	user 0m0,130s	user 0m0,264s	user 0m9,646s	user 0m36,231s	user 2m0,825s

Figure 12: Guided schedule clause: Table shows the time taken by different number of thread for different sizes of matrices

```
void decompose(double **a, double **l, double **u, int size) {

        int i, k, j;

    for(k=0; k<size; k++){

        for(i=k+1; i<size; i++)
            a[i][k] /= a[k][k];

        #pragma omp parallel for shared(a, size, k) private(i) schedule(static)
        for(i=k+1; i<size; i++){
            for(j=k+1; j<size; j++){
                a[i][j] -= a[i][k] * a[k][j];
            }
        }
    }


    for(i=0; i<size; i++){
        for(j=0; j<=i; j++){
            if(i==j) {
                l[i][j] = 1.0f;
            }
            else  {
                l[i][j] = a[i][j];
            }
        }
    }

    for(i=0; i < size; i++){
        for(j=i; j < size; j++){
            u[i][j] = a[i][j];
        }
    }


}
```

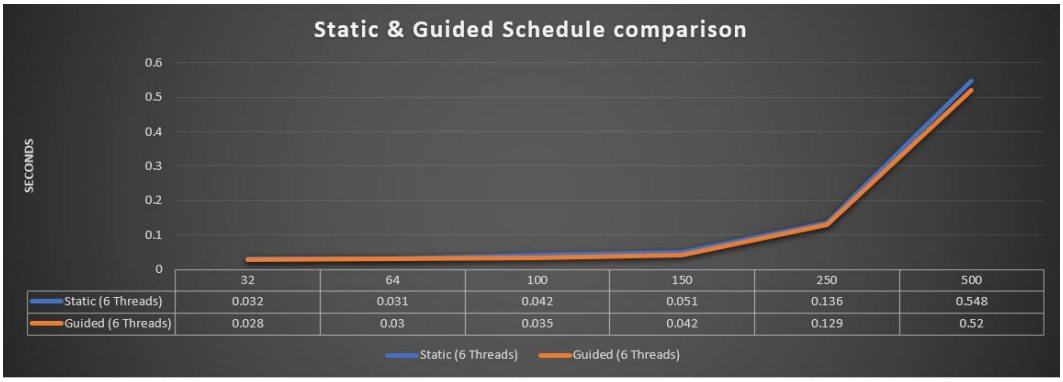
Figure 10: Shows the code of the decompose function where we declared static schedule clause with thread creation.

	Square Matrix Size							
No Of threads	32	64	100	150	250	500	1000	2000
2	real 0m0,032s	real 0m0,028s	real 0m0,026s	real 0m0,034s	real 0m0,143s	real 0m0,603s	real 0m12,509s	real 2m35,730s
	user 0m0,000s	user 0m0,005s	user 0m0,007s	user 0m0,013s	user 0m0,120s	user 0m0,579s	user 0m12,570s	user 2m37,997s
6	real 0m0,032s	real 0m0,031s	real 0m0,042s	real 0m0,051s	real 0m0,136s	real 0m0,548s	real 0m11,660s	real 2m31,568s
	user 0m0,016s	user 0m0,018s	user 0m0,022s	user 0m0,032s	user 0m0,141s	user 0m0,596s	user 0m11,984s	user 2m39,293s
8	real 0m0,024s	real 0m0,028s	real 0m0,031s	real 0m0,048s	real 0m0,146s	real 0m0,567s	real 0m11,734s	real 2m31,374s
	user 0m0,020s	user 0m0,024s	user 0m0,029s	user 0m0,029s	user 0m0,154s	user 0m0,619s	user 0m12,204s	user 2m41,275s
16	real 0m0,036s	real 0m0,035s	real 0m0,048s	real 0m0,289s	real 0m0,178s	real 0m0,623s	real 0m12,700s	real 2m34,582s
	user 0m0,088s	user 0m0,092s	user 0m0,072s	user 0m3,779s	user 0m0,629s	user 0m1,564s	user 0m28,044s	user 3m18,564s

Figure 11: Static schedule clause: Table shows the time taken by different number of thread for different sizes of matrices

In both static and dynamic scheduling, the performance is the best when the

number of threads are 6. So, we will compare both type of scheduling using 6 threads data from the tables provided in figure 11 and figure 12.



	32	64	100	150	250	500
Static (6 Threads)	0.032	0.031	0.042	0.051	0.136	0.548
Guided (6 Threads)	0.028	0.03	0.035	0.042	0.129	0.52

Figure 13: Graph shows the comparison between static and guided scheduler when the threads are 6 in number.

The results shows that the parallelization using OpenMp with guided scheduler increased the performance by a small margin but still it is an increase.

6 Results.

I tried to parallelize the whole program as well but it showed no improvement in performance in fact with some threads number it showed a lower performance as compared when we just parallelized one function i.e. decompose(). The reason could be that there are a lot of dependencies of variables while doing matrices operations.

The below table and graph shows the data for all the three phases i.e. naive code performance, optimized code performance and finally parallelized code performance. The table shows how the performance was increased when the matrix size increases.

		Squre Matrix Size							
		32	64	100	150	250	500	1000	2000
Time Taken in Seconds	Naïve Code	0.036	0.055	0.054	0.11	0.284	1.944	21.064	233.694
	Optimized code (-O3 -funroll-loops)	0.02	0.021	0.033	0.032	0.134	0.569	11.831	151.952
	Guided (6 Threads)	0.028	0.03	0.035	0.042	0.129	0.52	11.615	150.989

Figure 14: Graph shows the comparison between static and guided scheduler when the threads are 6 in number.

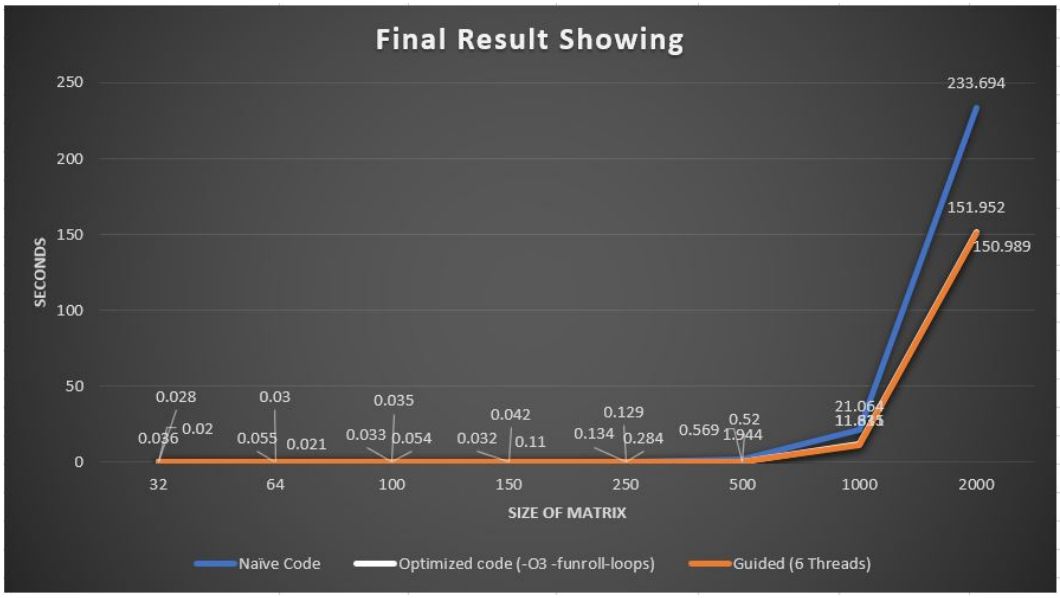


Figure 15: Graph shows the comparison between static and guided scheduler when the threads are 6 in number.

The results showed that the program overall did performed a bit better when the optimized program is parallelized. Here the difference between optimized and parallelized code is not too much but as we can see in the graph as the sze of the matrix increases the parallelized code performance is increases more and more.

The Real time which the most optimized plus paralleized program took for 2000 size of square matrix was 150.989 seconds compared to the non optimized version which took 233.694 seconds. This means that the program became 35.39% faster with optimization and parallelization ((233.694

- 150.989)/233.694 = 0.35390296712). This result was achieved while parallelizing plus optimizing the whole program using time command.

Please note that results mentioned in the tables and graphs are best timing results inside 5 measurements.

7 Conclusion.

One can see the comparison between the performance increased by using OpenMP with 6 threads in the above graphs. The highest performance is achieved when we parallelized our optimized code with guided schedule and 6 threads. So, I concluded this project with high performance using -O3 -funroll-loops flag for optimization and using OpenMP with 6 threads. Please note that other threads like 8 and 16 shows a decrease in the program performance.

8 References.

https://en.wikipedia.org/wiki/OpenMP
https://www.ibm.com/docs/en/xl-c-and-cpp-aix/16.1?topic=processing-pragma-omp
https://en.wikipedia.org/wiki/Call_graph
https://www.youtube.com/results?search_query=lu+factorization+method
https://en.wikipedia.org/wiki/LU_decomposition
https://www.geeksforgeeks.org/l-u-decomposition-system-linear-equations/: :text=L%20U%20deco
https://stackoverflow.com/questions/3949901/pthreads-vs-openmp: :text=Pthreads%20and%20Op