Minor Project (CSCE 735)
Sharatchandra Janapareddy
UIN: 329006499

1) Code submitted separately. To compile the code use the following command on ADA:

    icpc -qopenmp -std=c++11 GPR.cpp -o GPR.exe

   To run the code, use the following command with the size of the matrix and rstar coordinates as parameters:

    ./GPR.exe 32 0.5 0.5

   I have also submitted a job file which can be run on ADA with the following command:

    bsub<GPR.job

   The program outputs the time taken to compute fstar and the value of fstar.

2) To parallelize the algorithm, I parallelized the standard LU factorization operation as shown in the code below:

```
void LU(float **A){

    int n=GRID_SIZE*GRID_SIZE;
    int i,j,k;
    float m = 0;


    for( i=0;i<n-1;i++){
        #pragma omp barrier
        #pragma omp parallel for private(j,k,m) shared(A)
        for( j=i+1;j<n;j++){
            m= A[j][i]/A[i][i];
            for( k=i+1;k<n;k++){
                A[j][k]=A[j][k]-m*A[i][k];
            }
            A[j][i]=m;
        }
    }
```

Parallelized LU factorization

Parallelizing the LU factorization had the biggest improvement in time for the program. I also parallelized the solver operations (back substitution and forward substitution) with the reduction clause but this actually increased the computation time ever so slightly for the smaller values of m.

```
/////////forward substitution/////////////////
for(i=0;i<n;i++){
    temp=0;
    #pragma omp parallel for reduction(+:temp)
    for(j=0;j<i;j++){
        temp=temp+L[i][j]*y[j];
    }
    y[i] = (f[i] - temp)/L[i][i];
}

///////backward substitution//////////////
for(i=n-1;i>-1;i--){
    temp = 0.0;
    #pragma omp parallel for reduction(+:temp)
    for(j=n-1;j>i;j--){
     temp = temp + U[i][j]*z[j];
    }
    z[i]= (y[i]-temp)/U[i][i];
}
```

Parallelized Solver Equations

3) Results of running the program on ADA for varying inputs and threads

The following table shows the speedups achieved:

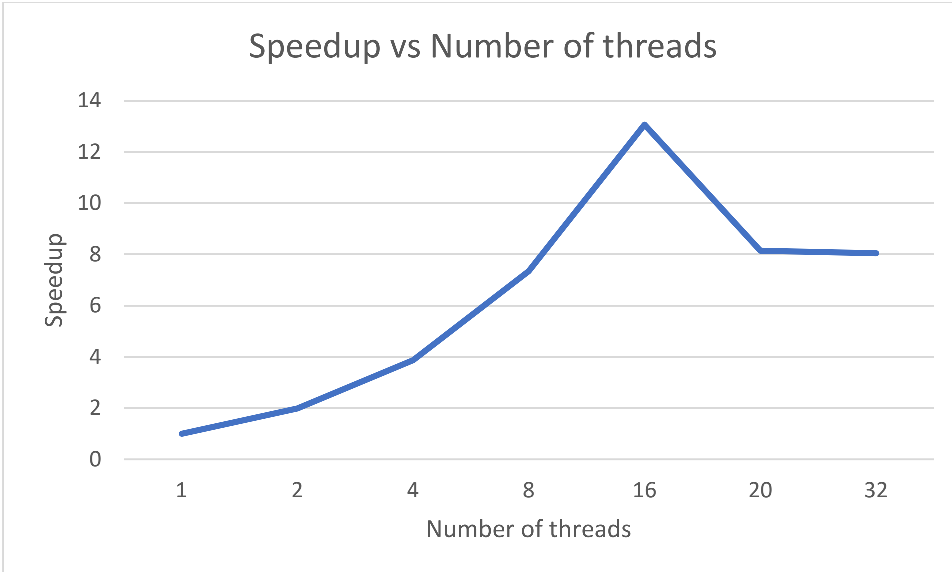
Size (m)	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads	20 Threads	32 Threads
32	1	1.962963	3.785714	7.162162	12.045455	5.8888889	7.3611111
50	1	1.983598	3.87	7.343454	13.074324	8.1473684	8.045738
100	1	1.937469	3.446987	6.273858	7.215903	6.674432	5.8099367

The following table shows the efficiencies:

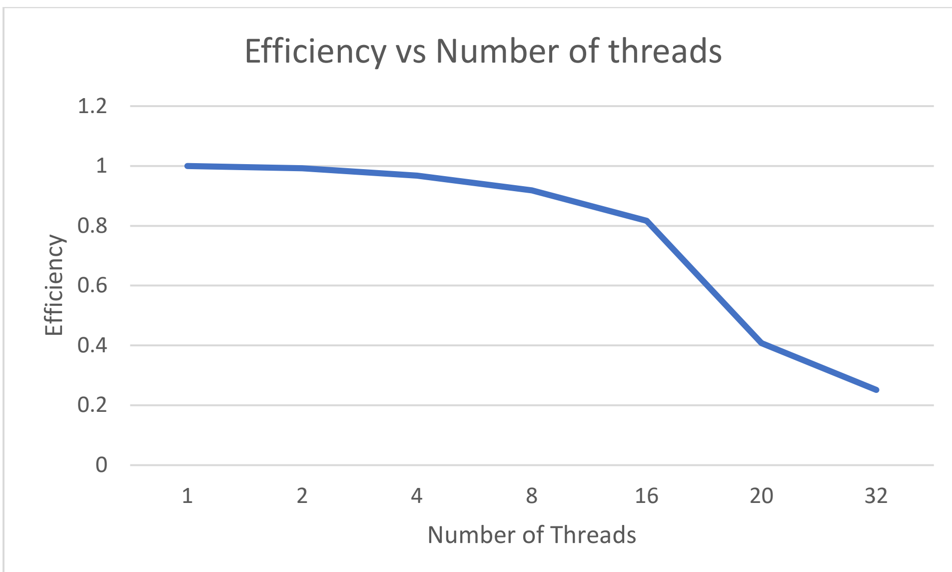
Size	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads	20 Threads	32 Threads
32	1	0.981481	0.946429	0.89527	0.7528409	0.2944444	0.2300347
50	1	0.991799	0.9675	0.917932	0.8171453	0.4073684	0.2514293
100	1	0.968734	0.861747	0.784232	0.4509939	0.3337216	0.1815605

It can be seen that the highest speedup was achieved when m = 50 as the following data graph shows. But as the speedup increases, the efficiency decreases. The drop in efficiency is steeper after the thread count is increased beyond 16. Another important observation is the drop in speedup when the thread count is increased beyond 16.

The program performs best when the thread count is 16.



Speedup for m=50



Efficiency for m=50

I have attached an excel sheet with all the results of my programs execution.

FLOPs:

To compute the flop rate I made use of linux's perf command along with the appropriate option to get the number of floating point operations in during the execution of my program.

```
Performance counter stats for './GPR.exe 32 0.5 0.5':

          237,268      r530110
       12,603,561      r531010
      729,911,779      r532010
          956,026      r534010
          842,854      r538010

      0.157562889 seconds time elapsed

[sharat.chandraj@ada3 Minor_Project]$ perf stat -e r530110 -e r531010 -e r532010 -e r534010 -e r538010 ./GPR.exe
5 0.5
Predicted value of function at rstar: 1.00109
Time taken to compute fstar: 257ms

Performance counter stats for './GPR.exe 50 0.5 0.5':

          584,332      r530110
       75,081,675      r531010
   10,717,465,522      r532010
        5,438,896      r534010
        1,933,935      r538010

      0.327221053 seconds time elapsed
```

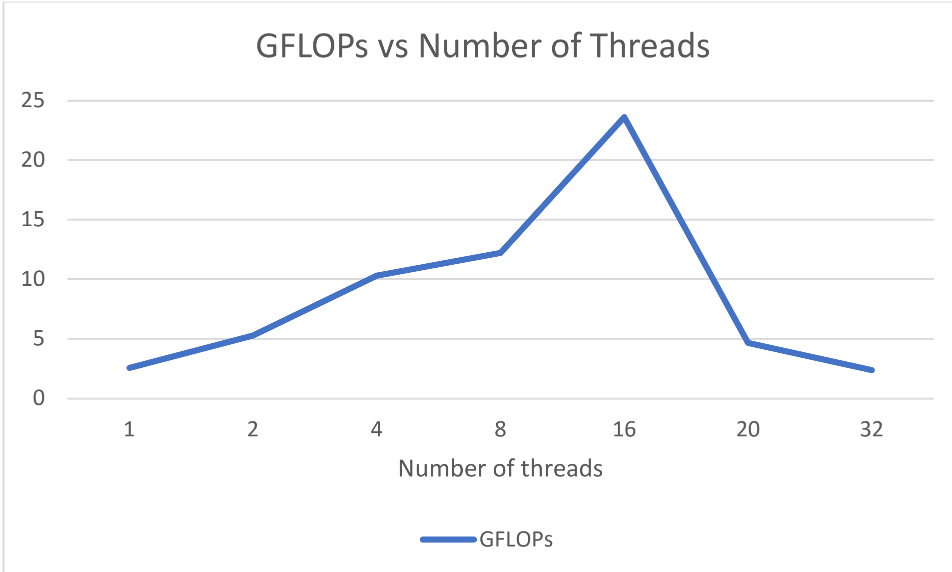
Flop count using perf command

The command uses CPU performance counters and each of the options in the image above unmasks different events.

- r530110: traditional 8087 style 80bit floating point operations
- r531010: traditional 8087 style 80bit floating point operations
- r532010: one single-precision operation
- r534010: four single-precision operation (32bit single precision packed into 128-bit register)
- r538010: one double-precision operation

I added the perf command to my job file to measure the single precision flops in my program when m=50 (best speedup case). The results shown in the table and graph below are performance measurements for the entire program and not just the LU factorization.

m	Thread count	Giga Flop count	Time (seconds)	GFLOPs
50	1	10.7249	4.1727	2.570254
50	2	10.7249	2.033	5.275406
50	4	10.7267	1.041	10.30423
50	8	10.7228	0.879	12.19886
50	16	10.7112	0.4534	23.62417
50	20	10.7259	2.298	4.667493
50	32	10.7229	4.481	2.39297



I also counted the floating-point operation manually in my code for m=50. The LU factorization has $10.416x10^9$ floating point operations and the solvers have $6.25x10^6$ floating-point operations each and I got the following results for the flop rate:

Thread count	Giga-Flop count	time(ms)	Flop Rate (GFLOPS)
1	10.429	3870	2.694832041
2	10.429	1951	5.345463865
4	10.429	1000	10.429
8	10.429	527	19.78937381
16	10.429	296	35.23310811
20	10.429	475	21.95578947
32	10.429	481	21.68191268

Once again, the max flop rate is achieved when the thread count is 16.

The theoretical peak performance for a single core can be computed with the following formula:

Node performance in GFlops = (CPU speed in GHz) x (number of CPU cores) x (CPU instruction per cycle) x (number of CPUs per node)

Setting the number of cores to 1, CPU speed to 2.5GHz and number of instructions to 8, we get the theoretical peak flop rate as **20 GFLOPs**.