Title: OpenMP Coursework

Author: Usman Basharat

Date: 13th December 2019

Course: COMP-1680 Clouds, Grids and Virtualisation

Word Count: 4874

School of Computing and Mathematical Sciences

UNIVERSITY of GREENWICH

Table of Contents

Table of Contents 2

Introduction 4

Task 1 4

- Different levels of compiler optimisation 4
- Adjustments Boundaries Jacobi 2D 5
- Adjustments Boundaries Gauss 2D 6
- Screenshots Jacobi 2D 7
- Screenshots Gauss 2D 7
- Jacobi 2D Test Cases 8
- Gauss 2D Test Cases 9

Task 2 10

- Parallelisation 10
- Adjustments Parallel Jacobi 2D 11
- Adjustments Parallel Gauss 2D 12
- Screenshots Jacobi 2D (Smaller) 13
- Screenshots Jacobi 2D (Bigger) 13
- Screenshots Gauss 2D (Bigger) 14
- Jacobi 2D Test Cases 15
- Gauss 2D Test Cases 17
  - Gauss 2D 1 Iteration Test Cases 18

Task 3 19

- Screenshots Jacobi 2D 19
- Screenshots Gauss 2D 19
- Jacobi 2D Test Cases 20
- Gauss 2D Test Cases 21
- Comparison Jacobi and Gauss Average 22
- Comparison of Task 1 and Task 3 Speedup 23

Task 4 24

- Adjustments Improvements Jacobi 2D 24
- Adjustment Improvements Gauss 2D 24
- Screenshots Jacobi 2D Improvements 25
- Screenshots Gauss 2D Improvements 25
- Jacobi 2D Test Cases 26

Gauss 2D Test Cases 27
Comparison Jacobi and Gauss Average 28
Comparison of Task 3 and Task 4 Speedup 29
References 30

Introduction

In this report, I have been given temperature distribution 2D problems with boundary conditions that need to be specified. I have been given Jacobi 2D and Gauss-Seidel 2D. The aim of both these codes to solve dimensional heat conductivity within the 2D area. However, both codes that are given are run differently. Each task consists of executing both tasks using different variations by plotting the results and a comparison of each results. Each task will be explained of what changes are made, the test cases, and screenshots using different problem sizes.

Task 1

In this task, the aim of this task is to set the boundary condition of "top 20°C, bottom 100°C, left 10°C and right 140°C" by using different problem sizes. Each problem sizes needed to be ran using different compiler optimisation. We were advised to stop the printing to get accurate results. Please note that these changes have been reflected in both codes. Referring to Figure 1 and Figure 2, changes were made to these to make sure that this task is complete. The changes that were made for both were to fix the boundary conditions that were set, commented the printing for accurate results and including a runtime. To put a runtime within both of this, the start of the day needed to be recorded and the end of the day. This was recorded and printing using external libraries too. Again, please note that all these changes are reflected and highlighted within Figure 1 and Figure 2.

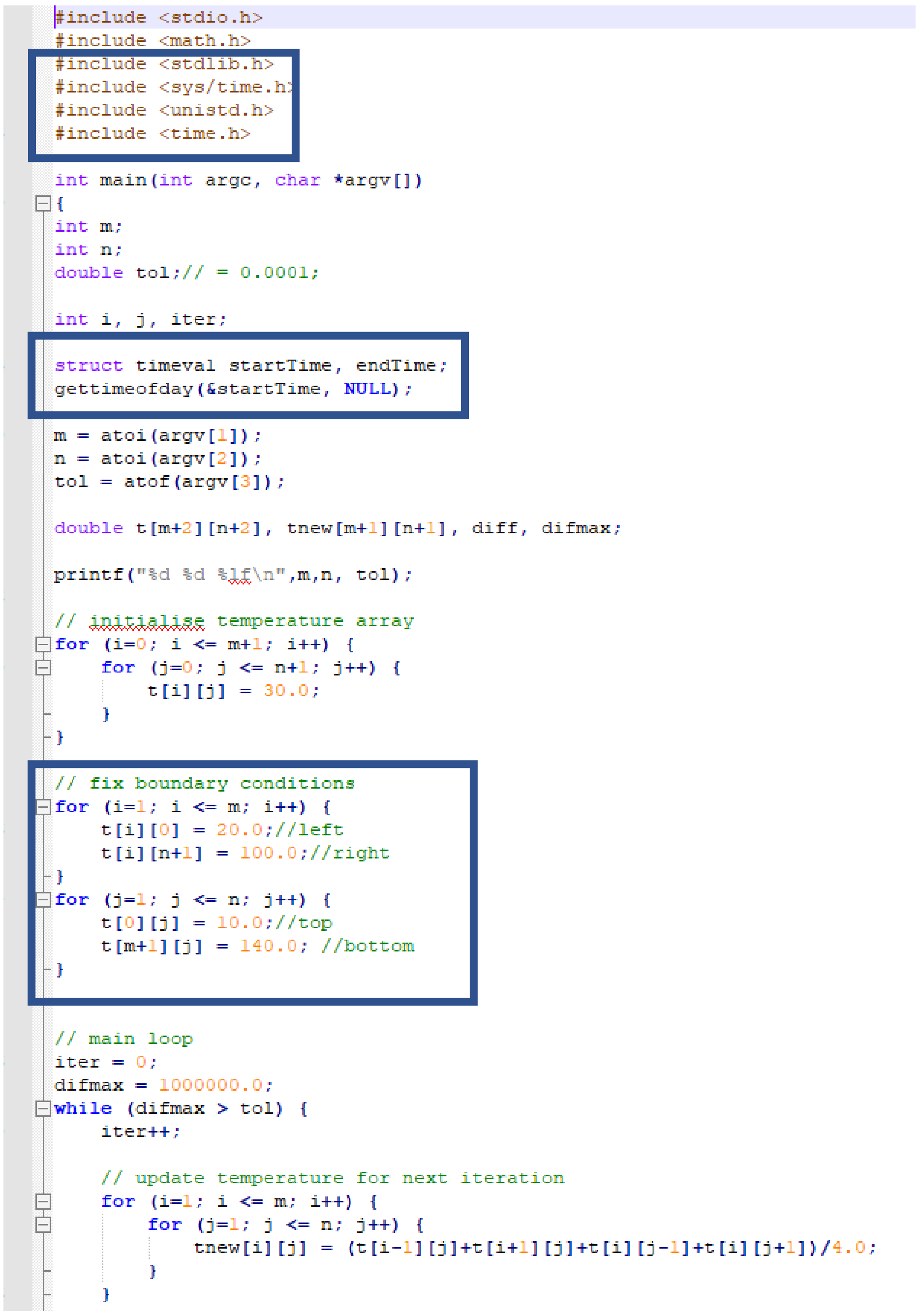
Different levels of compiler optimisation

An optimizing compiler tries to "minimise or maximise some attributes of executable computer programs" (Wikipedia, 2019). The aim of trying to compile algorithms is to produce the same result using different, or less, resources and expecting the same result. The aim of using different levels of compiler optimisation within Jacobi and Gauss is to expect different results. These different optimisations turn on different flags enabling them to run the code faster. Please refer yourself to Table 1 to get a good idea of the difference between each compiler that is used within Task 1. Table 1 consists of what each compiler does and the effect it has on the code. Some of the new optimisations have been introduced that has not been used in Jacobi nor Gauss too. This is only for descriptive in detail for the definition of compiler optimisation.

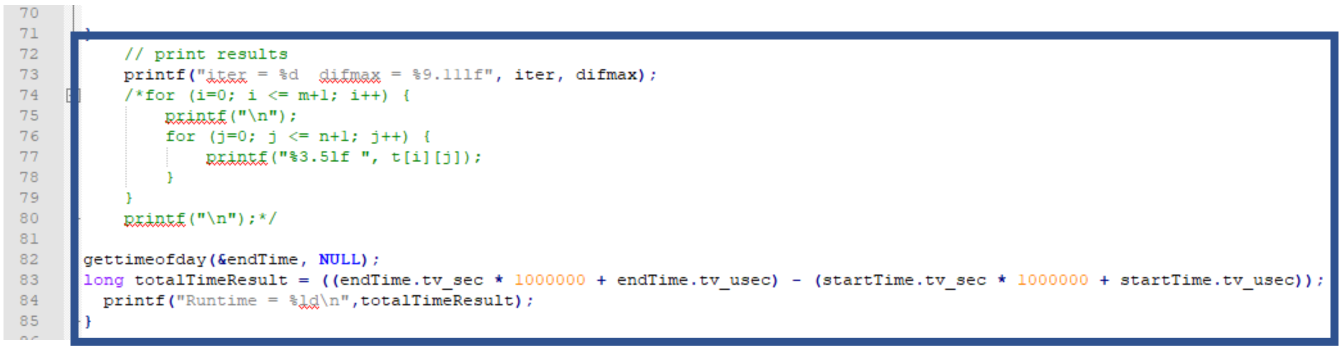
COMPILER OPTIMISATION	**DESCRIPTION**
NO OPTIMISATION	Results without using any optimisation.
-O0	"Reduces compiler time, however, only recommended for debugging purposes"
-O1	"Basic level of optimisation". Aims at smaller code for faster results only
-O2	Recommended level. A step-up from previous optimisation.
-O3	"The highest level of optimisation". Enables optimisation of expensive compiler time. However, it is known to break code too.
-OS	"Optimises code due to size". However, not a guarantee for results.
-OG	"Addresses fast optimization" A new level that has been introduced
-OFAST	A new level that has a plus of O3. Recommendation is not to use.

Table 1 shows the different level of compiler optimisation (Wiki Gentoo, 2019)

Adjustments Boundaries Jacobi 2D



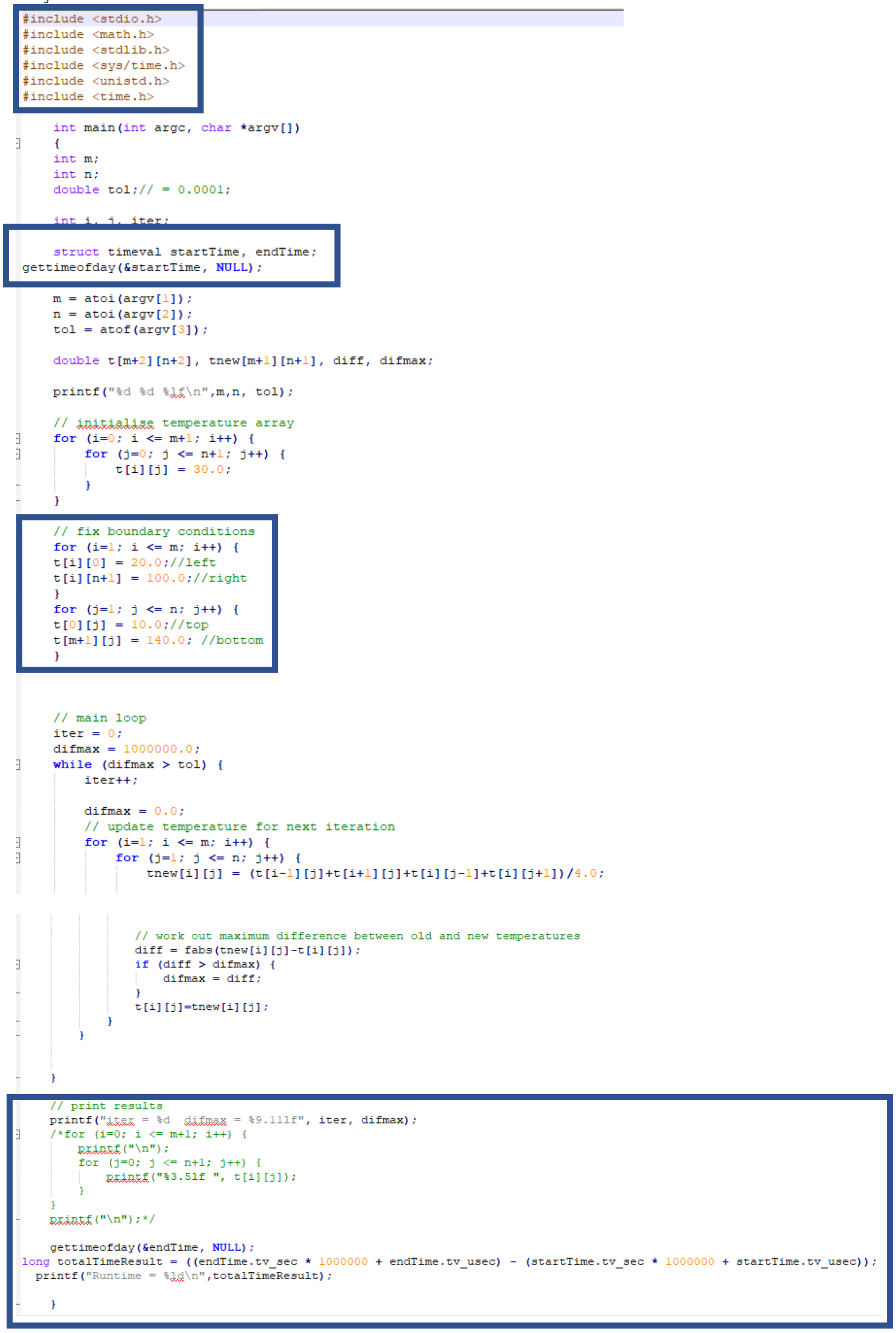
```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <sys/time.h>
#include <unistd.h>
#include <time.h>

int main(int argc, char *argv[])
{
int m;
int n;
double tol;// = 0.0001;

int i, j, iter;

struct timeval startTime, endTime;
gettimeofday(&startTime, NULL);

m = atoi(argv[1]);
n = atoi(argv[2]);
tol = atof(argv[3]);

double t[m+2][n+2], tnew[m+1][n+1], diff, difmax;

printf("%d %d %lf\n",m,n, tol);

// initialise temperature array
for (i=0; i <= m+1; i++) {
    for (j=0; j <= n+1; j++) {
        t[i][j] = 30.0;
    }
}

// fix boundary conditions
for (i=1; i <= m; i++) {
    t[i][0] = 20.0;//left
    t[i][n+1] = 100.0;//right
}
for (j=1; j <= n; j++) {
    t[0][j] = 10.0;//top
    t[m+1][j] = 140.0; //bottom
}

// main loop
iter = 0;
difmax = 1000000.0;
while (difmax > tol) {
    iter++;

    // update temperature for next iteration
    for (i=1; i <= m; i++) {
        for (j=1; j <= n; j++) {
            tnew[i][j] = (t[i-1][j]+t[i+1][j]+t[i][j-1]+t[i][j+1])/4.0;
        }
    }
```



```
}
    // print results
    printf("iter = %d  difmax = %9.11lf", iter, difmax);
    /*for (i=0; i <= m+1; i++) {
        printf("\n");
        for (j=0; j <= n+1; j++) {
            printf("%3.5lf ", t[i][j]);
        }
    }
    printf("\n");*/

gettimeofday(&endTime, NULL);
long totalTimeResult = ((endTime.tv_sec * 1000000 + endTime.tv_usec) - (startTime.tv_sec * 1000000 + startTime.tv_usec));
  printf("Runtime = %ld\n",totalTimeResult);
}
```

Figure 1 shows the changes that were made for Jacobi

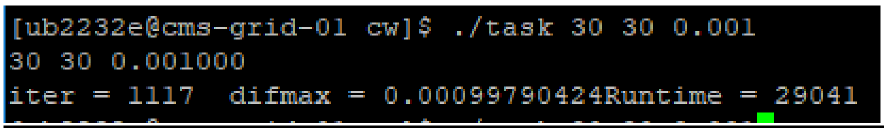
Adjustments Boundaries Gauss 2D



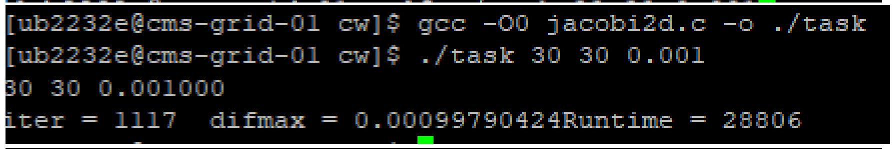
```c
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <sys/time.h>
#include <unistd.h>
#include <time.h>

    int main(int argc, char *argv[])
    {
    int m;
    int n;
    double tol;// = 0.0001;

    int i, j, iter;

    struct timeval startTime, endTime;
gettimeofday(&startTime, NULL);

    m = atoi(argv[1]);
    n = atoi(argv[2]);
    tol = atof(argv[3]);

    double t[m+2][n+2], tnew[m+1][n+1], diff, difmax;

    printf("%d %d %lf\n",m,n, tol);

    // initialise temperature array
    for (i=0; i <= m+1; i++) {
        for (j=0; j <= n+1; j++) {
            t[i][j] = 30.0;
        }
    }

    // fix boundary conditions
    for (i=1; i <= m; i++) {
    t[i][0] = 20.0;//left
    t[i][n+1] = 100.0;//right
    }
    for (j=1; j <= n; j++) {
    t[0][j] = 10.0;//top
    t[m+1][j] = 140.0; //bottom
    }


    // main loop
    iter = 0;
    difmax = 1000000.0;
    while (difmax > tol) {
        iter++;

        difmax = 0.0;
        // update temperature for next iteration
        for (i=1; i <= m; i++) {
            for (j=1; j <= n; j++) {
                tnew[i][j] = (t[i-1][j]+t[i+1][j]+t[i][j-1]+t[i][j+1])/4.0;


                // work out maximum difference between old and new temperatures
                diff = fabs(tnew[i][j]-t[i][j]);
                if (diff > difmax) {
                    difmax = diff;
                }
                t[i][j]=tnew[i][j];
            }
        }


    }
    // print results
    printf("iter = %d  difmax = %9.11lf", iter, difmax);
    /*for (i=0; i <= m+1; i++) {
        printf("\n");
        for (j=0; j <= n+1; j++) {
            printf("%3.5lf ", t[i][j]);
        }
    }
    printf("\n");*/

    gettimeofday(&endTime, NULL);
long totalTimeResult = ((endTime.tv_sec * 1000000 + endTime.tv_usec) - (startTime.tv_sec * 1000000 + startTime.tv_usec));
  printf("Runtime = %ld\n",totalTimeResult);

    }
```

Figure 2 shows the changes that were made for Gauss.

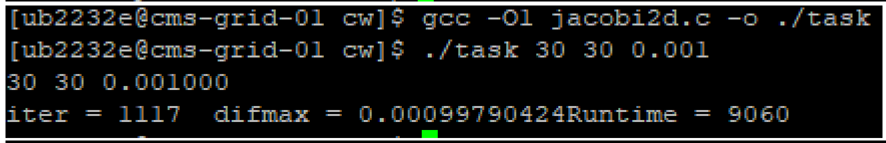
Screenshots Jacobi 2D



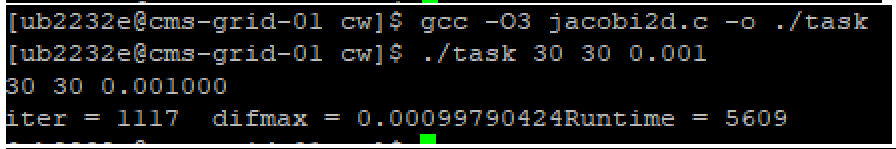
```
[ub2232e@cms-grid-01 cw]$ ./task 30 30 0.001
30 30 0.001000
iter = 1117   difmax = 0.00099790424Runtime = 29041
```



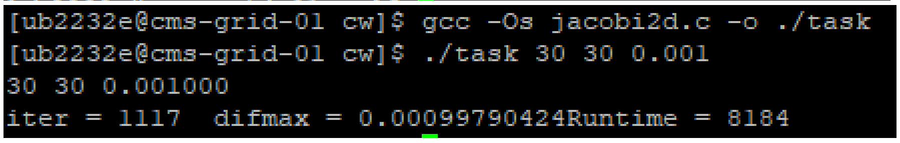
```
[ub2232e@cms-grid-01 cw]$ gcc -O0 jacobi2d.c -o ./task
[ub2232e@cms-grid-01 cw]$ ./task 30 30 0.001
30 30 0.001000
iter = 1117   difmax = 0.00099790424Runtime = 28806
```



```
[ub2232e@cms-grid-01 cw]$ gcc -O1 jacobi2d.c -o ./task
[ub2232e@cms-grid-01 cw]$ ./task 30 30 0.001
30 30 0.001000
iter = 1117   difmax = 0.00099790424Runtime = 9060
```

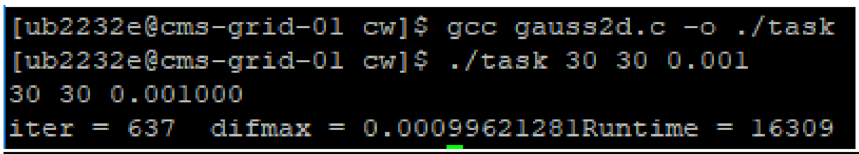


```
[ub2232e@cms-grid-01 cw]$ gcc -O3 jacobi2d.c -o ./task
[ub2232e@cms-grid-01 cw]$ ./task 30 30 0.001
30 30 0.001000
iter = 1117   difmax = 0.00099790424Runtime = 5609
```

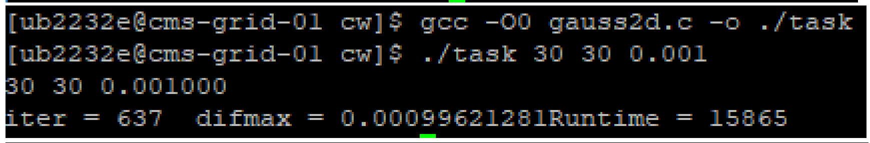


```
[ub2232e@cms-grid-01 cw]$ gcc -Os jacobi2d.c -o ./task
[ub2232e@cms-grid-01 cw]$ ./task 30 30 0.001
30 30 0.001000
iter = 1117   difmax = 0.00099790424Runtime = 8184
```

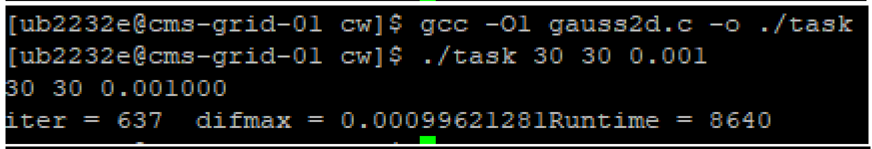
Screenshots Gauss 2D



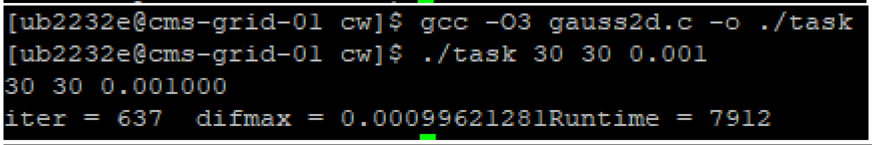
```
[ub2232e@cms-grid-01 cw]$ gcc gauss2d.c -o ./task
[ub2232e@cms-grid-01 cw]$ ./task 30 30 0.001
30 30 0.001000
iter = 637   difmax = 0.00099621281Runtime = 16309
```



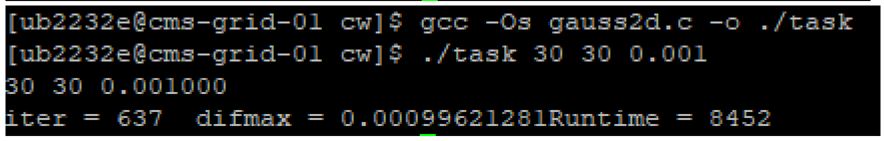
```
[ub2232e@cms-grid-01 cw]$ gcc -O0 gauss2d.c -o ./task
[ub2232e@cms-grid-01 cw]$ ./task 30 30 0.001
30 30 0.001000
iter = 637   difmax = 0.00099621281Runtime = 15865
```



```
[ub2232e@cms-grid-01 cw]$ gcc -O1 gauss2d.c -o ./task
[ub2232e@cms-grid-01 cw]$ ./task 30 30 0.001
30 30 0.001000
iter = 637   difmax = 0.00099621281Runtime = 8640
```



```
[ub2232e@cms-grid-01 cw]$ gcc -O3 gauss2d.c -o ./task
[ub2232e@cms-grid-01 cw]$ ./task 30 30 0.001
30 30 0.001000
iter = 637   difmax = 0.00099621281Runtime = 7912
```



```
[ub2232e@cms-grid-01 cw]$ gcc -Os gauss2d.c -o ./task
[ub2232e@cms-grid-01 cw]$ ./task 30 30 0.001
30 30 0.001000
iter = 637   difmax = 0.00099621281Runtime = 8452
```

These are the screenshots that were used of a different compiler optimisation were used on the same problem size. As you can clearly see the difference between each of the screenshots is the fact that O3, as explained previously, is the most dominant and fastest optimisation level for both Jacobi and Gauss. The difference between both codes is that Gauss is the faster rate of executing the code. As you may see, without optimisation for both codes consist of a difference between 29,041 for Jacobi; and significantly lower is 16,309 for Gauss. Please note that these changes are relatively the same for test cases below. However, I do notice a change is that for Gauss, as the problem sizes increase for different optimisation; it stays the same. A consistent pattern throughout Gauss. This is not the case for Jacobi as O3 has a clear advantage. However, Gauss is not the same. Please refer yourself over to Table 2 and Table 3 where you can see these test cases.

Jacobi 2D Test Cases

Jacobi 2D	20*20(s)	100*100(s)	250*250(s)	500*500(s)
No Optimisation	RESULTS			
TEST NO. 1	0.007007	1.259234	16.152857	98.223882
TEST NO. 2	0.009067	1.149241	17.118581	96.618963
TEST NO. 3	0.007411	1.121526	16.868968	96.646195
TEST NO. 4	0.007168	1.356270	16.083406	96.524509
TEST NO. 5	0.006946	1.088764	15.961578	96.540806
AVERAGE:	**0.0075198**	**1.195007**	**16.437078**	**96.910871**

Jacobi 2D	20*20(s)	100*100(s)	250*250(s)	500*500(s)
-O0	RESULTS			
TEST NO. 1	0.006977	1.340728	16.347451	96.012711
TEST NO. 2	0.006982	1.244599	16.293195	96.143704
TEST NO. 3	0.007268	1.333791	16.244055	96.535460
TEST NO. 4	0.007418	1.115653	16.180131	98.309539
TEST NO. 5	0.007609	1.176860	15.847265	96.475462
AVERAGE:	**0.0072508**	**1.2423262**	**16.1824194**	**96.6953752**

Jacobi 2D	20*20(s)	100*100(s)	250*250(s)	500*500(s)
-O1	RESULTS			
TEST NO. 1	0.002079	0.476736	5.232900	28.569098
TEST NO. 2	0.002082	0.343736	4.603637	27.434463
TEST NO. 3	0.002106	0.482531	4.580474	27.364959
TEST NO. 4	0.002027	0.424280	4.422464	27.664923
TEST NO. 5	0.002029	0.383116	4.375130	27.701728
AVERAGE:	**0.0020646**	**0.4220798**	**4.642921**	**27.7470342**

Jacobi 2D	20*20(s)	100*100(s)	250*250(s)	500*500(s)
-O3	RESULTS			
TEST NO. 1	0.001531	0.307831	3.477413	18.369137
TEST NO. 2	0.001579	0.395744	3.450681	18.401564
TEST NO. 3	0.001446	0.296989	3.736732	18.566448
TEST NO. 4	0.001551	0.285333	3.050917	18.558907
TEST NO. 5	0.001606	0.274192	3.304717	18.306188
AVERAGE:	**0.0015426**	**0.3120178**	**3.404092**	**18.4404488**

Jacobi 2D	20*20(s)	100*100(s)	250*250(s)	500*500(s)
-Os	RESULTS			
TEST NO. 1	0.002140	0.306868	4.580550	27.528298
TEST NO. 2	0.002094	0.429638	4.541586	29.372532
TEST NO. 3	0.001981	0.412990	4.920604	29.053330
TEST NO. 4	0.001969	0.400632	4.479400	28.739711
TEST NO. 5	0.002156	0.356029	4.786950	28.919932
AVERAGE:	**0.002068**	**0.3812314**	**4.661818**	**28.7227606**

Table 2 shows the test cases for Jacobi

Gauss 2D Test Cases

Gauss 2D	20*20(s)	100*100(s)	250*250(s)	500*500(s)
No Optimisation		RESULTS		
TEST NO. 1	0.004307	0.751120	12.371875	85.523379
TEST NO. 2	0.004081	0.859215	11.594849	85.372035
TEST NO. 3	0.003826	0.862438	11.423719	84.873826
TEST NO. 4	0.003782	0.823397	11.486222	84.373195
TEST NO. 5	0.003873	0.761650	11.350337	86.579961
AVERAGE:	**0.003974**	**0.811564**	**11.6454**	**85.344479**

Gauss2D	20*20(s)	100*100(s)	250*250(s)	500*500(s)
-O0		RESULTS		
TEST NO. 1	0.004004	0.757562	11.741479	84.546083
TEST NO. 2	0.004065	0.727403	11.492874	88.000630
TEST NO. 3	0.004003	0.844078	11.439810	86.134485
TEST NO. 4	0.003812	0.871283	11.804330	86.455544
TEST NO. 5	0.003789	0.879249	11.976221	85.236231
AVERAGE:	**0.003935**	**0.815915**	**11.690943**	**86.074595**

Gauss2D	20*20(s)	100*100(s)	250*250(s)	500*500(s)
-O1		RESULTS		
TEST NO. 1	0.001740	0.433881	6.917785	45.409179
TEST NO. 2	0.001794	0.421155	6.316302	45.812901
TEST NO. 3	0.001994	0.390309	6.169000	46.201633
TEST NO. 4	0.001734	0.466192	6.699500	45.466647
TEST NO. 5	0.001845	0.408194	6.144695	45.298540
AVERAGE:	**0.001821**	**0.423946**	**6.4494564**	**45.63778**

Gauss 2D	20*20(s)	100*100(s)	250*250(s)	500*500(s)
-O3		RESULTS		
TEST NO. 1	0.001781	0.393659	6.396697	45.433953
TEST NO. 2	0.001709	0.383639	6.421257	45.106833
TEST NO. 3	0.001743	0.426180	6.373042	46.373132
TEST NO. 4	0.001700	0.428743	6.599847	46.148424
TEST NO. 5	0.001717	0.420797	6.416398	45.402019
AVERAGE:	**0.00173**	**0.410604**	**6.4414482**	**45.692872**

Gauss 2D	20*20(s)	100*100(s)	250*250(s)	500*500(s)
-Os		RESULTS		
TEST NO. 1	0.001738	0.537205	6.034644	45.268853
TEST NO. 2	0.001774	0.390633	6.364533	45.413866
TEST NO. 3	0.001718	0.405520	6.084089	46.646487
TEST NO. 4	0.001707	0.440929	7.141049	45.379514
TEST NO. 5	0.00943	0.490913	6.086839	44.906773
AVERAGE:	**0.003273**	**0.45304**	**6.3422308**	**45.523099**

Table 3 shows the test cases for Gauss

Task 2

In this task, modifications are made from Task 1 to produce a parallel version of Jacobi and Gauss using OpenMP. The command that were used to produce the outcome was "***gcc -fopenmp jacobi2d.c -* task***". This was also the same execution used for Gauss by including the relevant file. This tasks also asked us to include timers to report the run-time of each code. This was completed as an example of one of the labs that were done. The highlighted code shows that the start time needs to be at the front of the code and the end time needs to be when the code has been executed. Once both have each time, a simple print statement by subtracting the end time from the start time. This is how this was executing using the time. This version needed to be included using different threads. These threads, as highlighted, were given as an argument for the user to type in. Lastly, Gauss-Seidel code must be only 1 iteration using 1 and 2 threads. A difference in each of the solutions will be explained in this report. Please refer yourself to Figure 3 and Figure 4 to see these reflected changes that were made to parallelise Jacobi and Gauss 2D.

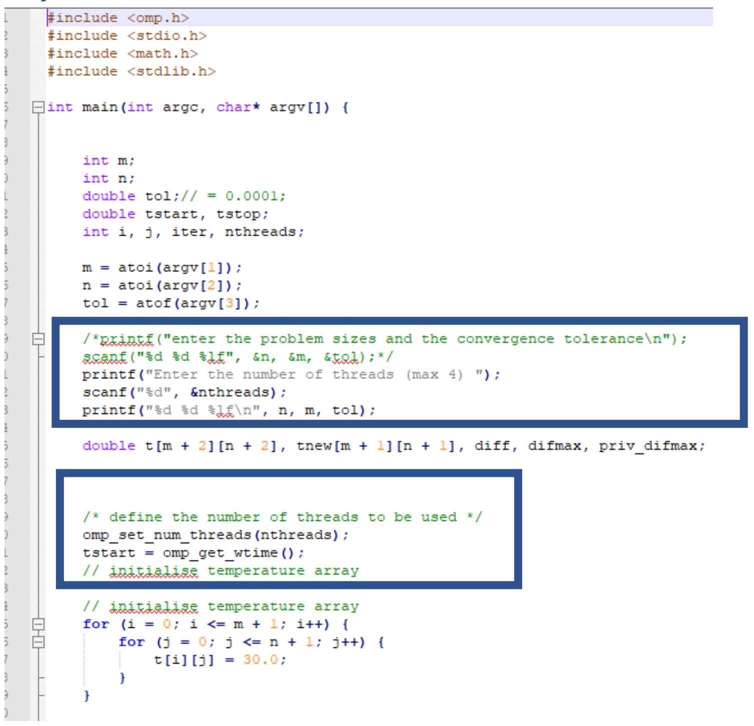
Parallelisation

In general, parallel programming is a type of computation that is broken and dissected into parts that can be solved separately (Barney, 2008). C programs often express their parallel by using ***for*** loops. This gets directly called to the compiler by communicating with what to do with it. As you can see for Figure 3 and Figure 4, many ***#pragmas*** are used as to parallelise these codes. Pragmas are a way of communicating directly to the compiler stating what this will do for you. Therefore, by including any information alongside this, it will tell the compiler of what to do. Please refer yourself to Table 4 where it can show the definitions used to parallelise Jacobi and Gauss.

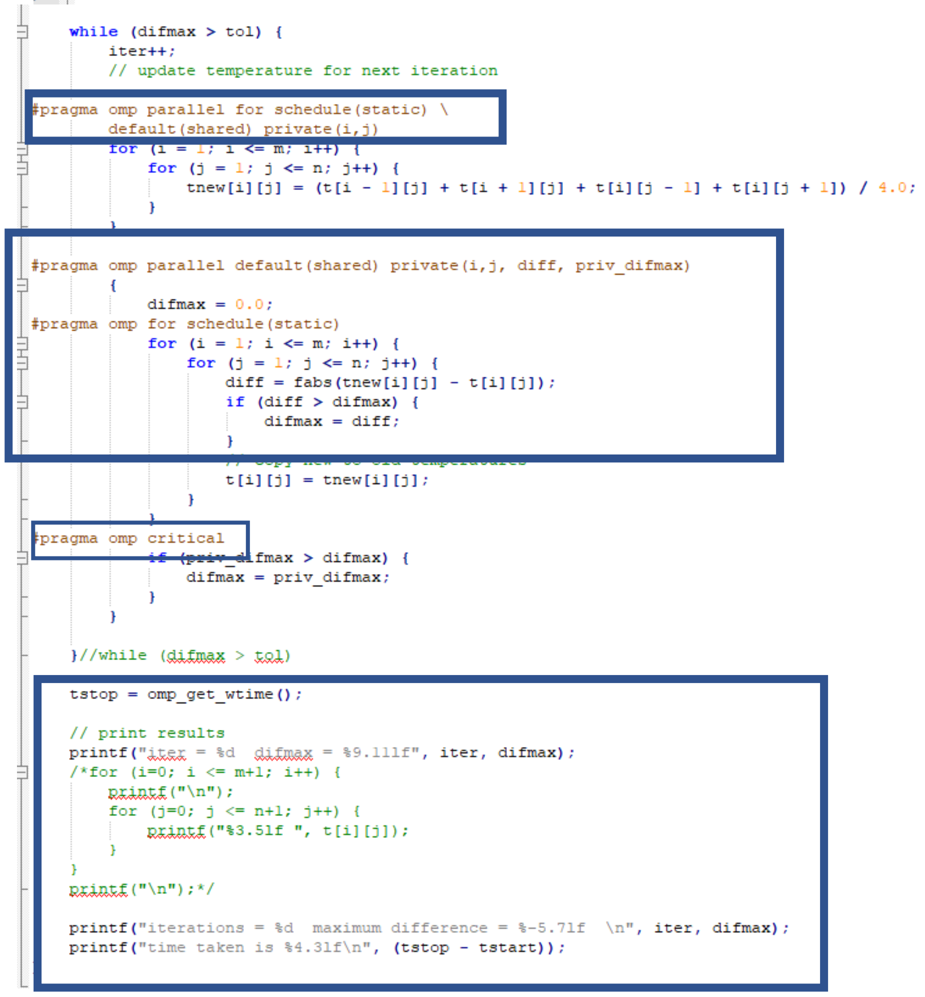
PARALLEL CODE	DESCRIPTION
PRAGMA	Verify information as it will have information it needs to schedule the loops iteration.
CRITICAL	A portion of the code so that a thread each time can execute. Therefore, it has been noted as critical
SCHEDULE (STATIC)	Static means that the iteration blocks are mapped statistically
SCHEDULE(DYNAMIC)	Dynamic works as a first-come first served basis
PRIVATE	Private is declared so that it can only be used within the relevant section
SHARED	Shared is so that the variables can be shared with the threads in a team
OMP_SET_NUM_THREADS()	Returns the number of threads used within the parallel region
OMP_GET_WTIME()	Returns a double value relevant to the clock time

Table 4 shows the definitions used in for parallelisation

Adjustments Parallel Jacobi 2D



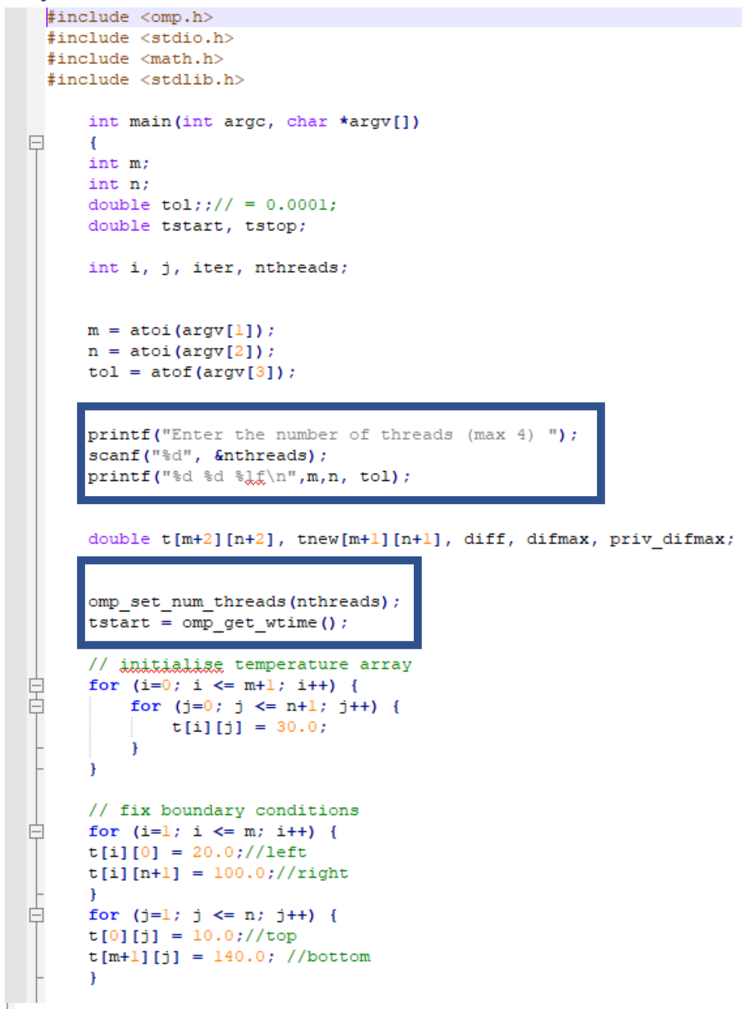
```
#include <omp.h>
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

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


    int m;
    int n;
    double tol;// = 0.0001;
    double tstart, tstop;
    int i, j, iter, nthreads;

    m = atoi(argv[1]);
    n = atoi(argv[2]);
    tol = atof(argv[3]);

    /*printf("enter the problem sizes and the convergence tolerance\n");
    scanf("%d %d %lf", &n, &m, &tol);*/
    printf("Enter the number of threads (max 4) ");
    scanf("%d", &nthreads);
    printf("%d %d %lf\n", n, m, tol);

    double t[m + 2][n + 2], tnew[m + 1][n + 1], diff, difmax, priv_difmax;


    /* define the number of threads to be used */
    omp_set_num_threads(nthreads);
    tstart = omp_get_wtime();
    // initialise temperature array

    // initialise temperature array
    for (i = 0; i <= m + 1; i++) {
        for (j = 0; j <= n + 1; j++) {
            t[i][j] = 30.0;
        }
    }
```



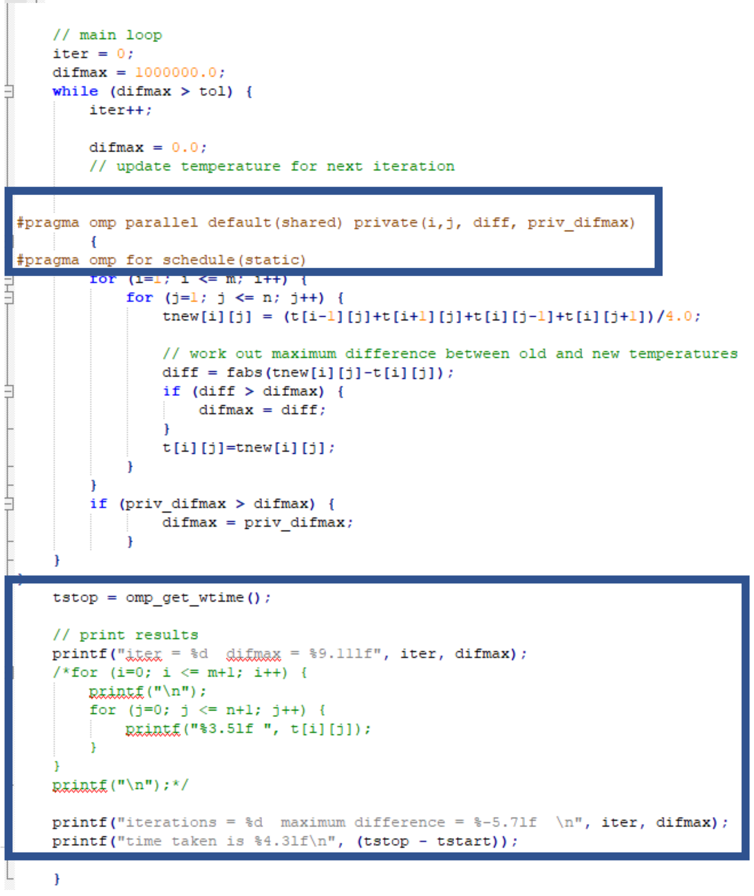
```
    while (difmax > tol) {
        iter++;
        // update temperature for next iteration

#pragma omp parallel for schedule(static) \
        default(shared) private(i,j)
        for (i = 1; i <= m; i++) {
            for (j = 1; j <= n; j++) {
                tnew[i][j] = (t[i - 1][j] + t[i + 1][j] + t[i][j - 1] + t[i][j + 1]) / 4.0;
            }
        }

#pragma omp parallel default(shared) private(i,j, diff, priv_difmax)
        {
            difmax = 0.0;
#pragma omp for schedule(static)
            for (i = 1; i <= m; i++) {
                for (j = 1; j <= n; j++) {
                    diff = fabs(tnew[i][j] - t[i][j]);
                    if (diff > difmax) {
                        difmax = diff;
                    }
                    // copy new to old temperatures
                    t[i][j] = tnew[i][j];
                }
            }
#pragma omp critical
            if (priv_difmax > difmax) {
                difmax = priv_difmax;
            }
        }

    }//while (difmax > tol)

    tstop = omp_get_wtime();

    // print results
    printf("iter = %d  difmax = %9.11lf", iter, difmax);
    /*for (i=0; i <= m+1; i++) {
        printf("\n");
        for (j=0; j <= n+1; j++) {
            printf("%3.5lf ", t[i][j]);
        }
    }
    printf("\n");*/

    printf("iterations = %d  maximum difference = %-5.7lf  \n", iter, difmax);
    printf("time taken is %4.3lf\n", (tstop - tstart));
```

Figure 3 shows the changes that were made to Jacobi

Adjustments Parallel Gauss 2D



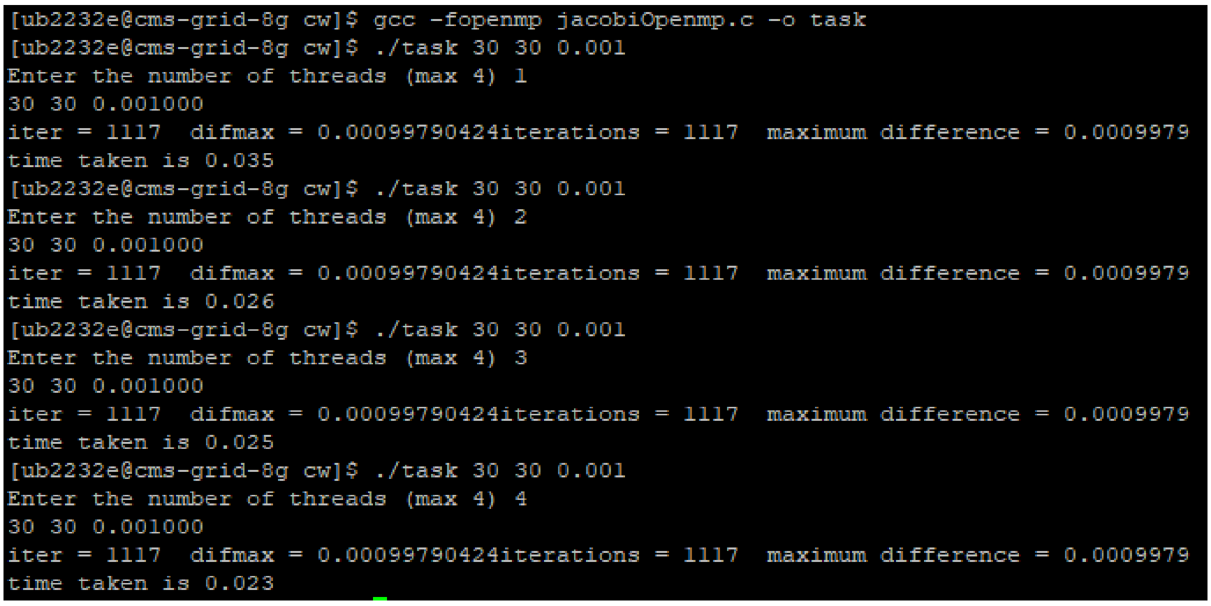
```c
#include <omp.h>
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

    int main(int argc, char *argv[])
    {
    int m;
    int n;
    double tol;;// = 0.0001;
    double tstart, tstop;

    int i, j, iter, nthreads;


    m = atoi(argv[1]);
    n = atoi(argv[2]);
    tol = atof(argv[3]);

    printf("Enter the number of threads (max 4) ");
    scanf("%d", &nthreads);
    printf("%d %d %lf\n",m,n, tol);

    double t[m+2][n+2], tnew[m+1][n+1], diff, difmax, priv_difmax;

    omp_set_num_threads(nthreads);
    tstart = omp_get_wtime();
    // initialise temperature array
    for (i=0; i <= m+1; i++) {
        for (j=0; j <= n+1; j++) {
            t[i][j] = 30.0;
        }
    }

    // fix boundary conditions
    for (i=1; i <= m; i++) {
    t[i][0] = 20.0;//left
    t[i][n+1] = 100.0;//right
    }
    for (j=1; j <= n; j++) {
    t[0][j] = 10.0;//top
    t[m+1][j] = 140.0; //bottom
    }
```



```c
    // main loop
    iter = 0;
    difmax = 1000000.0;
    while (difmax > tol) {
        iter++;

        difmax = 0.0;
        // update temperature for next iteration

#pragma omp parallel default(shared) private(i,j, diff, priv_difmax)
        {
#pragma omp for schedule(static)
        for (i=1; i <= m; i++) {
            for (j=1; j <= n; j++) {
                tnew[i][j] = (t[i-1][j]+t[i+1][j]+t[i][j-1]+t[i][j+1])/4.0;

                // work out maximum difference between old and new temperatures
                diff = fabs(tnew[i][j]-t[i][j]);
                if (diff > difmax) {
                    difmax = diff;
                }
                t[i][j]=tnew[i][j];
            }
        }
        if (priv_difmax > difmax) {
                difmax = priv_difmax;
            }
    }

    tstop = omp_get_wtime();

    // print results
    printf("iter = %d  difmax = %9.11lf", iter, difmax);
    /*for (i=0; i <= m+1; i++) {
        printf("\n");
        for (j=0; j <= n+1; j++) {
            printf("%3.5lf ", t[i][j]);
        }
    }
    printf("\n");*/

    printf("iterations = %d  maximum difference = %-5.7lf  \n", iter, difmax);
    printf("time taken is %4.3lf\n", (tstop - tstart));

    }
```

Figure 4 shows the changes that were made to Gauss

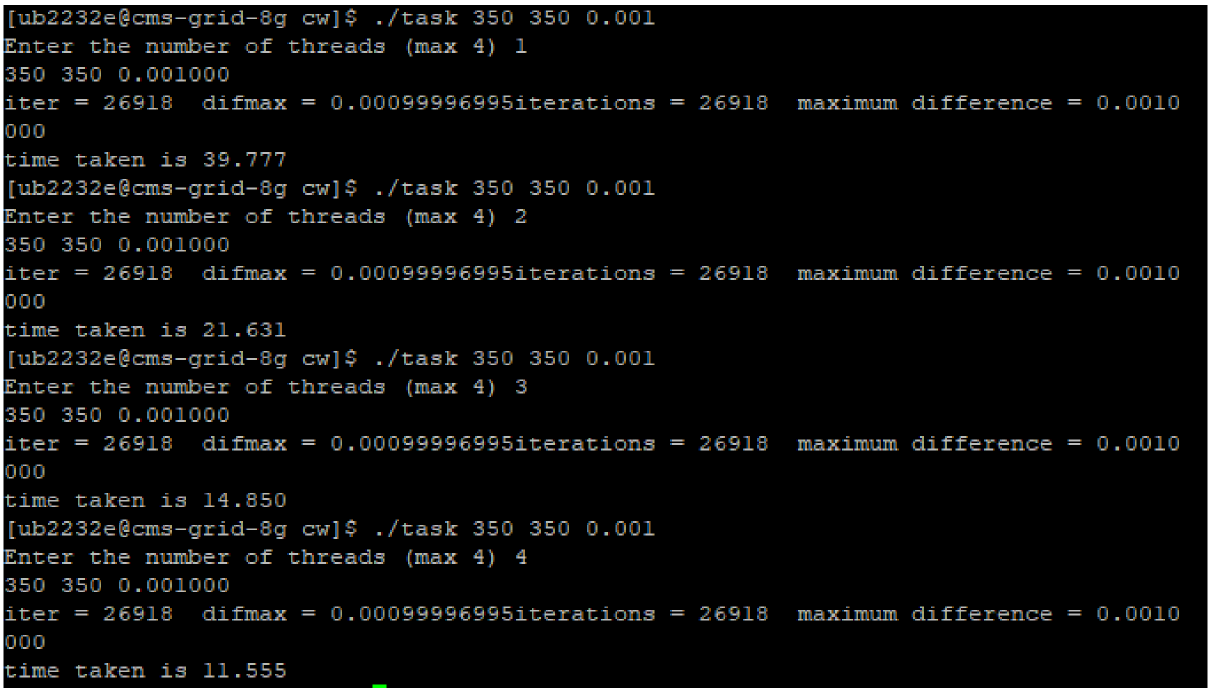
Screenshots Jacobi 2D (Smaller)



```
[ub2232e@cms-grid-8g cw]$ gcc -fopenmp jacobiOpenmp.c -o task
[ub2232e@cms-grid-8g cw]$ ./task 30 30 0.001
Enter the number of threads (max 4) 1
30 30 0.001000
iter = 1117  difmax = 0.00099790424iterations = 1117  maximum difference = 0.0009979
time taken is 0.035
[ub2232e@cms-grid-8g cw]$ ./task 30 30 0.001
Enter the number of threads (max 4) 2
30 30 0.001000
iter = 1117  difmax = 0.00099790424iterations = 1117  maximum difference = 0.0009979
time taken is 0.026
[ub2232e@cms-grid-8g cw]$ ./task 30 30 0.001
Enter the number of threads (max 4) 3
30 30 0.001000
iter = 1117  difmax = 0.00099790424iterations = 1117  maximum difference = 0.0009979
time taken is 0.025
[ub2232e@cms-grid-8g cw]$ ./task 30 30 0.001
Enter the number of threads (max 4) 4
30 30 0.001000
iter = 1117  difmax = 0.00099790424iterations = 1117  maximum difference = 0.0009979
time taken is 0.023
```

Figure 5 shows the smaller difference for Jacobi

Screenshots Jacobi 2D (Bigger)

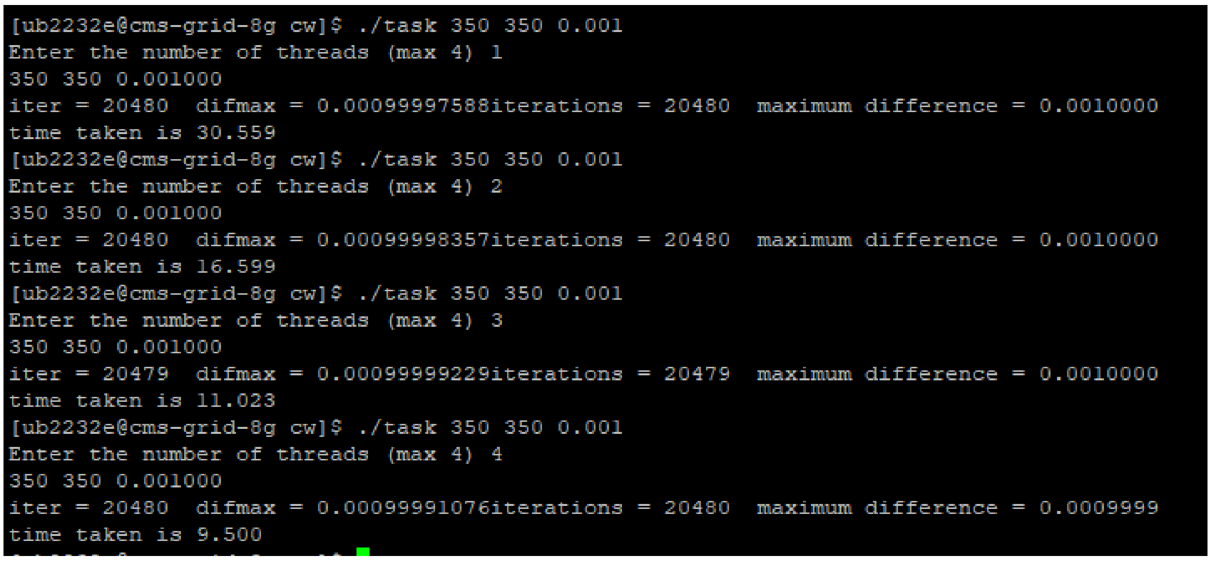


```
[ub2232e@cms-grid-8g cw]$ ./task 350 350 0.001
Enter the number of threads (max 4) 1
350 350 0.001000
iter = 26918  difmax = 0.00099996995iterations = 26918  maximum difference = 0.0010
000
time taken is 39.777
[ub2232e@cms-grid-8g cw]$ ./task 350 350 0.001
Enter the number of threads (max 4) 2
350 350 0.001000
iter = 26918  difmax = 0.00099996995iterations = 26918  maximum difference = 0.0010
000
time taken is 21.631
[ub2232e@cms-grid-8g cw]$ ./task 350 350 0.001
Enter the number of threads (max 4) 3
350 350 0.001000
iter = 26918  difmax = 0.00099996995iterations = 26918  maximum difference = 0.0010
000
time taken is 14.850
[ub2232e@cms-grid-8g cw]$ ./task 350 350 0.001
Enter the number of threads (max 4) 4
350 350 0.001000
iter = 26918  difmax = 0.00099996995iterations = 26918  maximum difference = 0.0010
000
time taken is 11.555
```

Figure 6 shows the bigger results for Jacobi

Figure 5 and Figure 6 both demonstrates the difference between the smaller and bigger results. As you can see, the difference between each thread is not much different for each thread. The difference between each thread is not much great considering the size of the problem. However, the bigger the result, the greater the difference. As you can see, the difference is much larger compared to the result of the smaller problem size. Please refer yourself to Table 5, Table 6 and Table 7 to show the test cases for this.

Screenshots Gauss 2D (Bigger)



```
[ub2232e@cms-grid-8g cw]$ ./task 350 350 0.001
Enter the number of threads (max 4) 1
350 350 0.001000
iter = 20480  difmax = 0.00099997588iterations = 20480  maximum difference = 0.0010000
time taken is 30.559
[ub2232e@cms-grid-8g cw]$ ./task 350 350 0.001
Enter the number of threads (max 4) 2
350 350 0.001000
iter = 20480  difmax = 0.00099998357iterations = 20480  maximum difference = 0.0010000
time taken is 16.599
[ub2232e@cms-grid-8g cw]$ ./task 350 350 0.001
Enter the number of threads (max 4) 3
350 350 0.001000
iter = 20479  difmax = 0.00099999229iterations = 20479  maximum difference = 0.0010000
time taken is 11.023
[ub2232e@cms-grid-8g cw]$ ./task 350 350 0.001
Enter the number of threads (max 4) 4
350 350 0.001000
iter = 20480  difmax = 0.00099991076iterations = 20480  maximum difference = 0.0009999
time taken is 9.500
```

Figure 7 shows the screenshots in Gauss.

Referring to Figure 7, this shows us that the same results show us the better difference. I did not include a smaller problem size for Gauss, as this would have been the same outcome. The difference of results shows us that, at the moment, Gauss is much faster when paralleling both together.

Table 5 shows us the consistency of each threat. However, the results do not show the greater difference for both. Hence, the reason to increase the problem size. When I ran both codes, I used Grid 01 for these results. These results came very in accurate as the greater the thread, the more fluctuated results that were presented. I was later advised to use 8G, as this shows much greater and accurate results. Once this was changed for all tables including Table 5, 6 and 7; these were the results that showed much more accurate results. To run this, I used the following command: **ssh cms-grid-8G.** However, I was advised that the greater the threads, performance is an issue. This is when the more people use this grid, it would affect the results of the run time.

Furthermore, Table 6 and Table 7 shows us the results of Jacobi and Gauss. As you can see for both tables are that Gauss is faster when executing the same results. However, the same issue, within Task 1 appears with Gauss, is that the greater the problem size; the more time it takes to execute the code. When specifically comparing the average of 600 of 3 threads of Jacobi and Gauss; there is any a slight difference between the two. This is the same difference for 700, as when the third and fourth thread is used; Gauss has a greater time than Jacobi. This could be a performance issue. However, I have ran the code doing more than 5 results; and I noticed that this is still the same. Possibly, this could be an area where this could be improved. To improve the parallelisation of Gauss and make sure that the greater threads of greater problem sizes are executed faster.

Jacobi 2D Test Cases

Jacobi 2D	20*20(s)	100*100(s)	250*250(s)
1	RESULTS		
TEST NO. 1	0.010	1.114	16.081
TEST NO. 2	0.010	1.198	16.316
TEST NO. 3	0.010	1.277	15.985
TEST NO. 4	0.010	1.073	16.171
TEST NO. 5	0.010	1.116	15.868
AVERAGE:	**0.010**	**1.1556**	**16.0842**

Jacobi 2D	20*20(s)	100*100(s)	250*250(s)
2	RESULTS		
TEST NO. 1	0.010	0.686	8.261
TEST NO. 2	0.009	0.863	8.887
TEST NO. 3	0.009	0.887	8.370
TEST NO. 4	0.012	0.692	8.346
TEST NO. 5	0.009	0.690	8.969
AVERAGE:	**0.0098**	**0.7636**	**8.5666**

Jacobi 2D	20*20(s)	100*100(s)	250*250(s)
3	RESULTS		
TEST NO. 1	0.008	0.551	6.307
TEST NO. 2	0.008	0.508	6.147
TEST NO. 3	0.009	0.483	6.462
TEST NO. 4	0.010	0.479	5.881
TEST NO. 5	0.009	0.584	5.740
AVERAGE:	**0.0088**	**0.521**	**6.1074**

Jacobi 2D	20*20(s)	100*100(s)	250*250(s)
4	RESULTS		
TEST NO. 1	0.008	0.491	4.998
TEST NO. 2	0.009	0.456	4.855
TEST NO. 3	0.009	0.514	4.461
TEST NO. 4	0.009	0.403	4.863
TEST NO. 5	0.010	0.517	4.954
AVERAGE:	**0.009**	**0.4762**	**4.8262**

Table 5 shows the results of smaller cases in Jacobi

Jacobi 2D	500*500(s)	600*600(s)	700*700(s)
1	RESULTS		
TEST NO. 1	96.677	141.888	197.405
TEST NO. 2	96.213	144.529	193.793
TEST NO. 3	97.788	146.178	194.471
TEST NO. 4	97.789	144.371	194.980
TEST NO. 5	98.164	144.354	198.203
AVERAGE:	**97.3262**	**144.264**	**195.7704**

Jacobi 2D	500*500(s)	600*600(s)	700*700(s)
2	RESULTS		
TEST NO. 1	50.487	73.968	102.128
TEST NO. 2	50.899	65.811	102.056
TEST NO. 3	51.321	74.461	102.176
TEST NO. 4	53.019	75.049	102.947
TEST NO. 5	50.427	72.361	100.792
AVERAGE:	**51.2306**	**72.33**	**102.0198**

Jacobi 2D	500*500(s)	600*600(s)	700*700(s)
3	RESULTS		
TEST NO. 1	34.650	49.899	69.409
TEST NO. 2	35.176	50.298	70.978
TEST NO. 3	34.044	45.535	69.057
TEST NO. 4	35.088	50.450	68.210
TEST NO. 5	34.338	49.940	68.287
AVERAGE:	**34.6592**	**49.2244**	**69.1882**

Jacobi 2D	500*500(s)	600*600(s)	700*700(s)
4	RESULTS		
TEST NO. 1	27.599	38.919	52.695
TEST NO. 2	28.115	39.235	53.375
TEST NO. 3	25.971	38.059	51.279
TEST NO. 4	27.091	39.587	52.836
TEST NO. 5	26.371	37.368	52.029
AVERAGE:	**27.0294**	**38.6336**	**52.4428**

Table 6 shows the test cases of bigger sizes in Jacobi

Gauss 2D Test Cases

Gauss 2D	500*500(s)	600*600(s)	700*700(s)
1	RESULTS		
TEST NO. 1	84.129	137.997	193.777
TEST NO. 2	85.345	138.249	192.524
TEST NO. 3	87.074	135.854	193.360
TEST NO. 4	85.531	135.050	193.002
TEST NO. 5	86.108	134.946	192.994
AVERAGE:	**85.6374**	**136.4192**	**193.1314**

Gauss 2D	500*500(s)	600*600(s)	700*700(s)
2	RESULTS		
TEST NO. 1	53.751	69.592	101.084
TEST NO. 2	45.557	68.988	103.088
TEST NO. 3	44.759	69.650	101.144
TEST NO. 4	43.995	70.876	100.608
TEST NO. 5	43.960	70.311	99.528
AVERAGE:	**46.4044**	**69.8834**	**101.0904**

Gauss 2D	500*500(s)	600*600(s)	700*700(s)
3	RESULTS		
TEST NO. 1	30.851	49.091	70.260
TEST NO. 2	29.657	48.641	70.279
TEST NO. 3	29.916	48.854	71.006
TEST NO. 4	30.960	47.723	70.939
TEST NO. 5	30.230	48.827	69.844
AVERAGE:	**30.3228**	**48.6272**	**70.4656**

Gauss 2D	500*500(s)	600*600(s)	700*700(s)
4	RESULTS		
TEST NO. 1	22.465	38.209	53.531
TEST NO. 2	22.933	37.010	53.608
TEST NO. 3	22.619	37.585	53.254
TEST NO. 4	22.447	36.883	53.765
TEST NO. 5	22.327	37.740	52.797
AVERAGE:	**22.5582**	**37.4854**	**53.391**

Table 7 shows the results in Jacobi

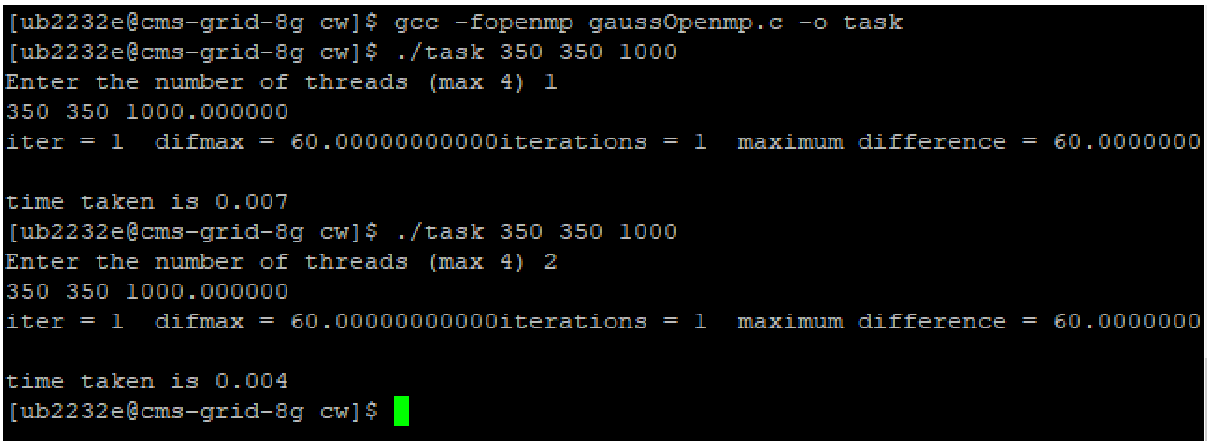
Gauss 2D 1 Iteration Test Cases

Referring to Table 8, these are the results of Gauss 2D being run through different problem sizes. This was enabled by increasing the tolerance enabling the problem to be ran in 1 iteration. The difference that has shown are that between each iteration there is not much difference when increasing the thread. However, when the problem sizes are increased for 1 thread, it shows a bigger difference than the second threat. The same problem sizes were used as previously to show the difference of each iteration. It would not make sense to make different problem sizes for this. Please refer yourself to Figure 8 to show that this has been done for this.

Gauss2d	500*500(s)	600*600(s)	700*700(s)
1 Thread		RESULTS	
TEST NO. 1	0.013	0.017	0.022
TEST NO. 2	0.012	0.016	0.022
TEST NO. 3	0.012	0.016	0.027
TEST NO. 4	0.012	0.016	0.020
TEST NO. 5	0.012	0.016	0.021
AVERAGE:	**0.0122**	**0.0162**	**0.0224**

Gauss2d	500*500(s)	600*600(s)	700*700(s)
2 Threads		RESULTS	
TEST NO. 1	0.008	0.010	0.014
TEST NO. 2	0.008	0.012	0.014
TEST NO. 3	0.008	0.011	0.014
TEST NO. 4	0.009	0.011	0.014
TEST NO. 5	0.009	0.011	0.014
AVERAGE:	**0.0084**	**0.011**	**0.014**

Table 8 shows the results of 1 iteration of Gauss



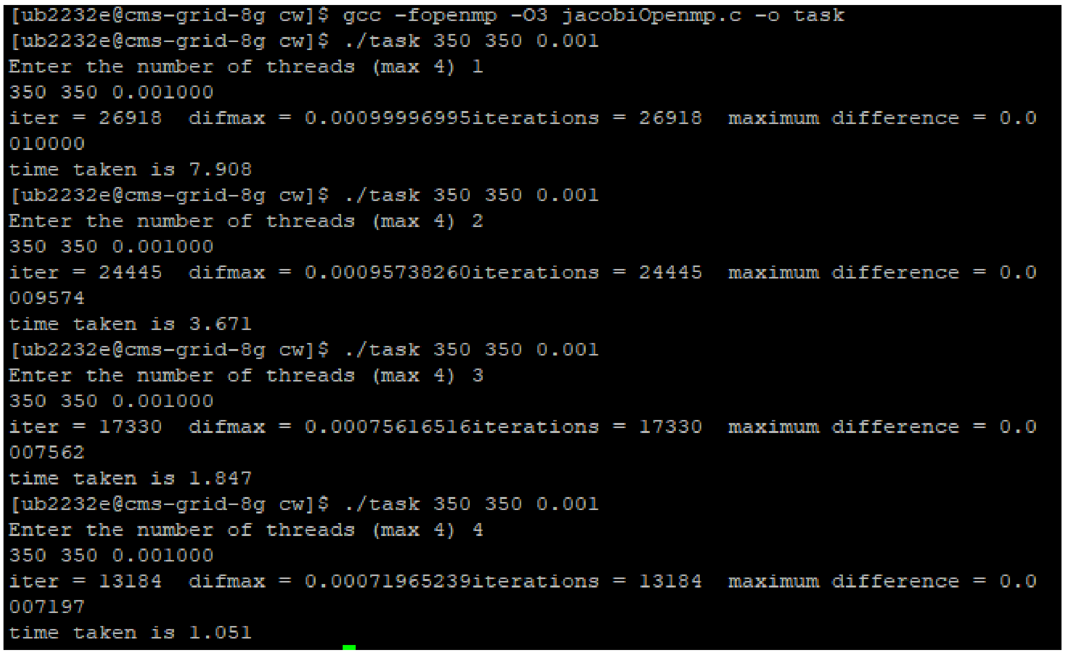
```
[ub2232e@cms-grid-8g cw]$ gcc -fopenmp gaussOpenmp.c -o task
[ub2232e@cms-grid-8g cw]$ ./task 350 350 1000
Enter the number of threads (max 4) 1
350 350 1000.000000
iter = 1  difmax = 60.00000000000iterations = 1  maximum difference = 60.0000000

time taken is 0.007
[ub2232e@cms-grid-8g cw]$ ./task 350 350 1000
Enter the number of threads (max 4) 2
350 350 1000.000000
iter = 1  difmax = 60.00000000000iterations = 1  maximum difference = 60.0000000

time taken is 0.004
[ub2232e@cms-grid-8g cw]$
```

Figure 8 shows the execution of 1 iteration in Gauss

Task 3

In this task, speed up results were asked by executing range of problem sizes alongside a single optimisation processor. A similar optimisation to the parallel code was asked. Furthermore, it only made sense to use the best result that was gained from Step 1 by using O3 optimisation. A different range of threads were used from 2 to 8. For this, I used 2, 4, 6, 8 as the threads. However, it was not necessary to improve on the parallelisation that was completed from Step 2. Therefore, the following tests were running on the Step 2. There, no separate code is made. This would not make any sense as changes were not made for this task. As you can see within Figure 9 and 10, I ran these tests by using the following command: ***gcc -fopenmp -O3 jacobiOpenmp.c -o task.***

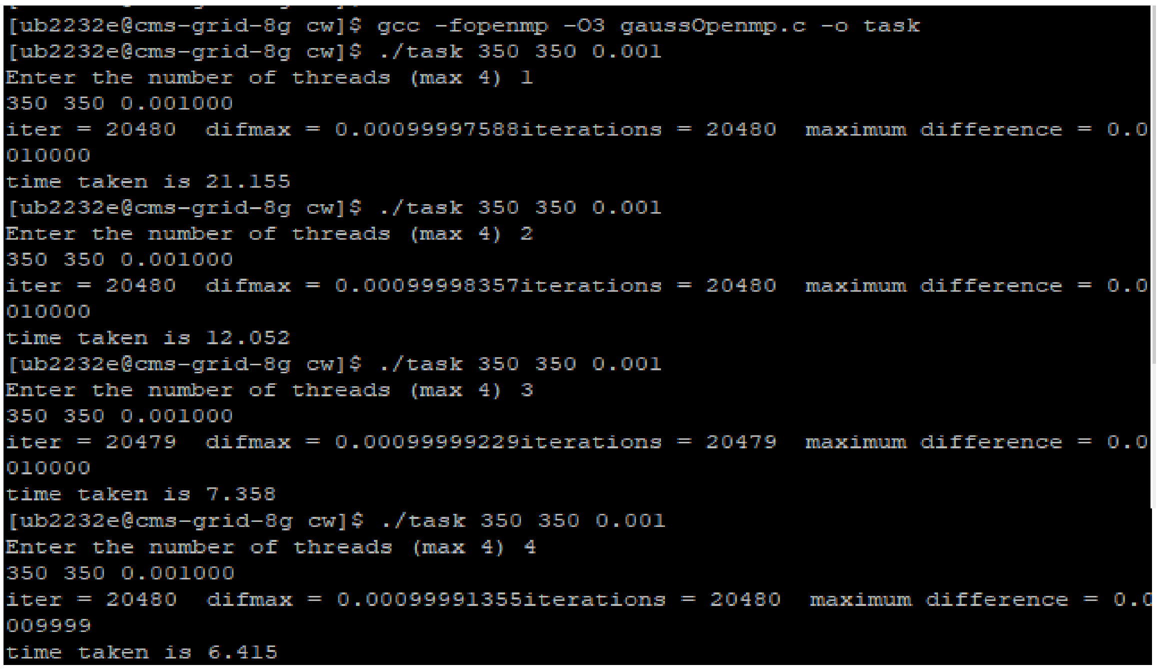
Screenshots Jacobi 2D



```
[ub2232e@cms-grid-8g cw]$ gcc -fopenmp -O3 jacobiOpenmp.c -o task
[ub2232e@cms-grid-8g cw]$ ./task 350 350 0.001
Enter the number of threads (max 4) 1
350 350 0.001000
iter = 26918  difmax = 0.00099996995iterations = 26918  maximum difference = 0.0
010000
time taken is 7.908
[ub2232e@cms-grid-8g cw]$ ./task 350 350 0.001
Enter the number of threads (max 4) 2
350 350 0.001000
iter = 24445  difmax = 0.00095738260iterations = 24445  maximum difference = 0.0
009574
time taken is 3.671
[ub2232e@cms-grid-8g cw]$ ./task 350 350 0.001
Enter the number of threads (max 4) 3
350 350 0.001000
iter = 17330  difmax = 0.00075616516iterations = 17330  maximum difference = 0.0
007562
time taken is 1.847
[ub2232e@cms-grid-8g cw]$ ./task 350 350 0.001
Enter the number of threads (max 4) 4
350 350 0.001000
iter = 13184  difmax = 0.00071965239iterations = 13184  maximum difference = 0.0
007197
time taken is 1.051
```

Figure 9 shows the results of Jacobi

Screenshots Gauss 2D



```
[ub2232e@cms-grid-8g cw]$ gcc -fopenmp -O3 gaussOpenmp.c -o task
[ub2232e@cms-grid-8g cw]$ ./task 350 350 0.001
Enter the number of threads (max 4) 1
350 350 0.001000
iter = 20480  difmax = 0.00099997588iterations = 20480  maximum difference = 0.0
010000
time taken is 21.155
[ub2232e@cms-grid-8g cw]$ ./task 350 350 0.001
Enter the number of threads (max 4) 2
350 350 0.001000
iter = 20480  difmax = 0.00099998357iterations = 20480  maximum difference = 0.0
010000
time taken is 12.052
[ub2232e@cms-grid-8g cw]$ ./task 350 350 0.001
Enter the number of threads (max 4) 3
350 350 0.001000
iter = 20479  difmax = 0.00099999229iterations = 20479  maximum difference = 0.0
010000
time taken is 7.358
[ub2232e@cms-grid-8g cw]$ ./task 350 350 0.001
Enter the number of threads (max 4) 4
350 350 0.001000
iter = 20480  difmax = 0.00099991355iterations = 20480  maximum difference = 0.0
009999
time taken is 6.415
```

Figure 10 shows the results of Gauss

Jacobi 2D Test Cases

Compiler Optimisation: O3

Jacobi 2D	500*500(s)	600*600(s)	700*700(s)
2		RESULTS	
TEST NO. 1	10.041	14.207	18.878
TEST NO. 2	10.651	14.443	19.997
TEST NO. 3	10.222	14.332	18.926
TEST NO. 4	9.717	14.227	19.548
TEST NO. 5	10.332	14.697	19.541
AVERAGE:	**10.1926**	**14.3812**	**19.378**

Jacobi 2D	500*500(s)	600*600(s)	700*700(s)
4		RESULTS	
TEST NO. 1	6.183	7.455	10.517
TEST NO. 2	5.930	8.054	10.460
TEST NO. 3	5.332	7.651	10.035
TEST NO. 4	5.285	7.411	11.016
TEST NO. 5	5.829	7.817	9.914
AVERAGE:	**5.7118**	**7.6776**	**10.3884**

Jacobi 2D	500*500(s)	600*600(s)	700*700(s)
6		RESULTS	
TEST NO. 1	4.474	5.340	7.777
TEST NO. 2	4.348	5.561	7.465
TEST NO. 3	4.327	5.659	7.297
TEST NO. 4	4.201	4.632	7.737
TEST NO. 5	3.247	3.507	6.852
AVERAGE:	**4.1194**	**4.9398**	**7.4256**

Jacobi 2D	500*500(s)	600*600(s)	700*700(s)
8		RESULTS	
TEST NO. 1	3.668	3.603	5.665
TEST NO. 2	2.529	4.502	5.852
TEST NO. 3	2.553	4.082	5.674
TEST NO. 4	3.206	4.994	5.521
TEST NO. 5	2.411	4.510	5.859
AVERAGE:	**2.8734**	**4.3382**	**5.7142**

Table 9 shows the results of Jacobi

Gauss 2D Test Cases

Compiler Optimisation: O3

Gauss 2D	500*500(s)	600*600(s)	700*700(s)
2		RESULTS	
TEST NO. 1	28.798	45.044	66.627
TEST NO. 2	28.636	45.144	65.028
TEST NO. 3	28.843	46.044	65.448
TEST NO. 4	28.887	45.856	65.633
TEST NO. 5	29.257	45.993	66.147
AVERAGE:	**28.8842**	**45.6162**	**65.7766**

Gauss 2D	500*500(s)	600*600(s)	700*700(s)
4		RESULTS	
TEST NO. 1	15.101	24.028	34.552
TEST NO. 2	15.485	23.311	34.038
TEST NO. 3	15.184	25.421	33.962
TEST NO. 4	15.137	23.575	34.323
TEST NO. 5	15.485	24.038	33.946
AVERAGE:	**15.2784**	**24.0746**	**34.1642**

Gauss 2D	500*500(s)	600*600(s)	700*700(s)
6		RESULTS	
TEST NO. 1	11.068	16.066	24.566
TEST NO. 2	12.337	16.854	23.563
TEST NO. 3	10.574	16.550	23.568
TEST NO. 4	11.218	16.299	23.012
TEST NO. 5	10.502	16.705	23.353
AVERAGE:	**11.1398**	**16.4948**	**23.6124**

Gauss 2D	500*500(s)	600*600(s)	700*700(s)
8		RESULTS	
TEST NO. 1	9.524	12.799	18.473
TEST NO. 2	8.156	12.506	17.207
TEST NO. 3	7.993	12.609	18.347
TEST NO. 4	8.625	13.647	17.415
TEST NO. 5	5.745	12.885	17.367
AVERAGE:	**8.0086**	**12.8892**	**17.7618**

Table 10 shows the results of Gauss.

Referring to Table 9 and 10, both show the results of using O3 optimisation. As you can see, there is a huge difference in both results that you can see. A problem size of 500 using 2 threads can be ran in 10 seconds. However, Gauss with the same problem size is 28 seconds. Gauss makes more sense as the difference speedup from Task 2 shows the difference in size. By using 8 threads, the execution time for both makes sense. Jacob, on average, by using 8 threads compiles it 4 times as faster than using 2 threads. This is roughly the same for Gauss as 8 threads has a 4 times compiler rate than using 2 threads too. Therefore, there is not much difference in terms of comparing both results.

Comparison Jacobi and Gauss Average

Please refer yourself to Figure 11 and Figure 12 where this is a graph representation of the results of the test cases that were completed using O3 optimisation. These are the average of each problem sizes that were used. This shows a great interpretation of each size is decreased as soon as more threads have been used. This shows that the more threads are used and worked together; it would enable the time to decrease. This is the same for both out comes.

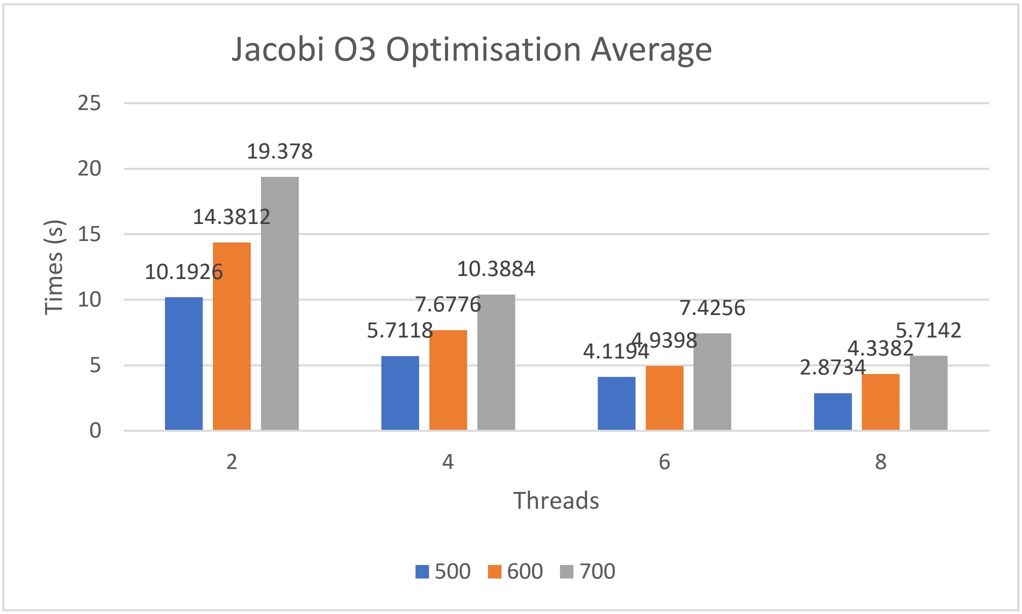


Figure 11 shows a graph of Jacobi using O3 optimisation

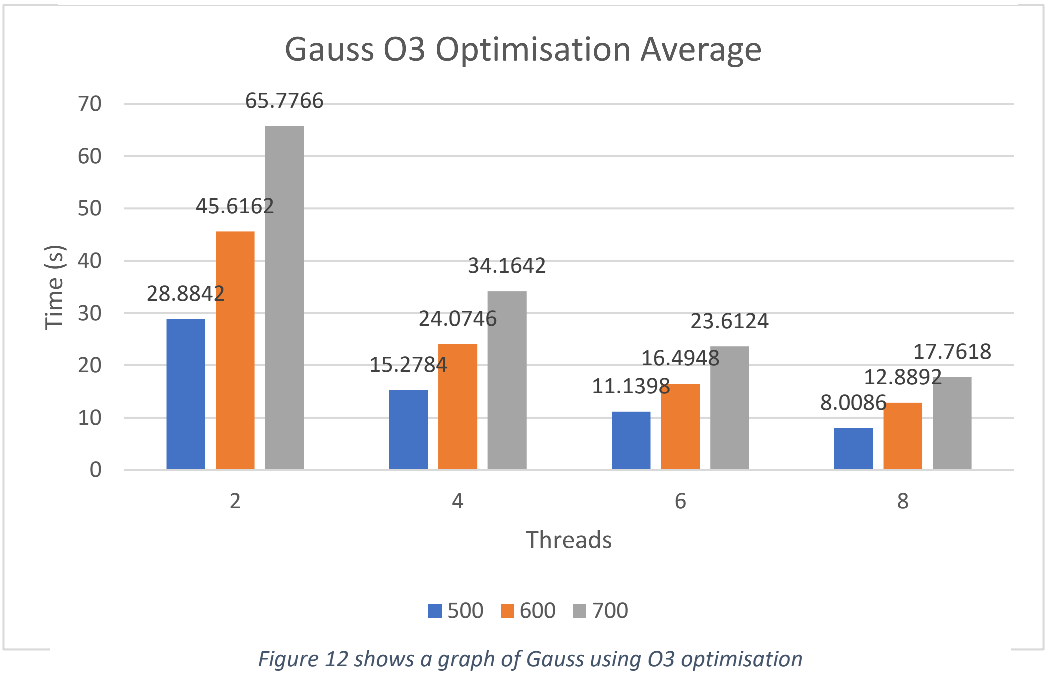


Figure 12 shows a graph of Gauss using O3 optimisation

Comparison of Task 1 and Task 3 Speedup

Referring to Figure 13 and Figure 14, this shows the results of the speed-up that of 500*500 problem size. This is due to the fact that 500 was used throughout each task. Therefore, it makes sense to provide a speedup result showing how much it has increased throughout each task. This was completed by getting Step 1 average of O3 which is **18.4404488.** This was divided by each average of 500 shown in Table 9. By doing this, you can see the results of the speed-up of this in Figure 13. **45.692872** is the result of Step 1 for Gauss too. By dividing the average of 500 show in Table 10, you can see the results of the speed-up of this Figure 14,

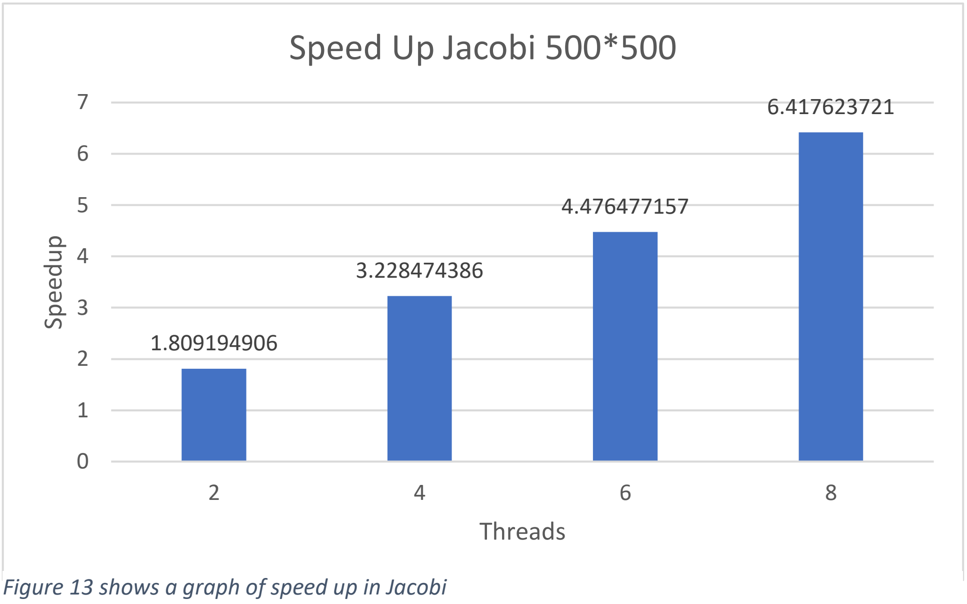


Figure 13 shows a graph of speed up in Jacobi

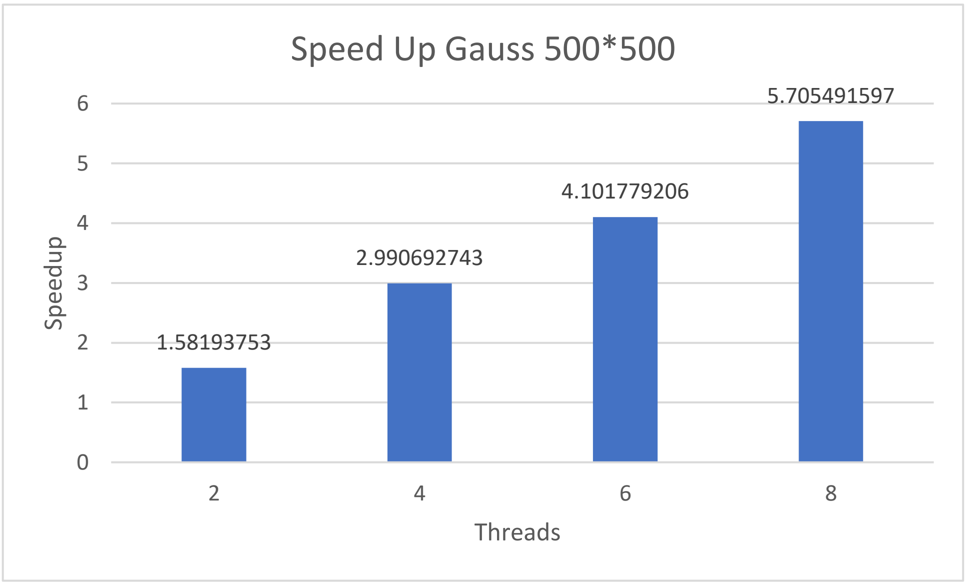
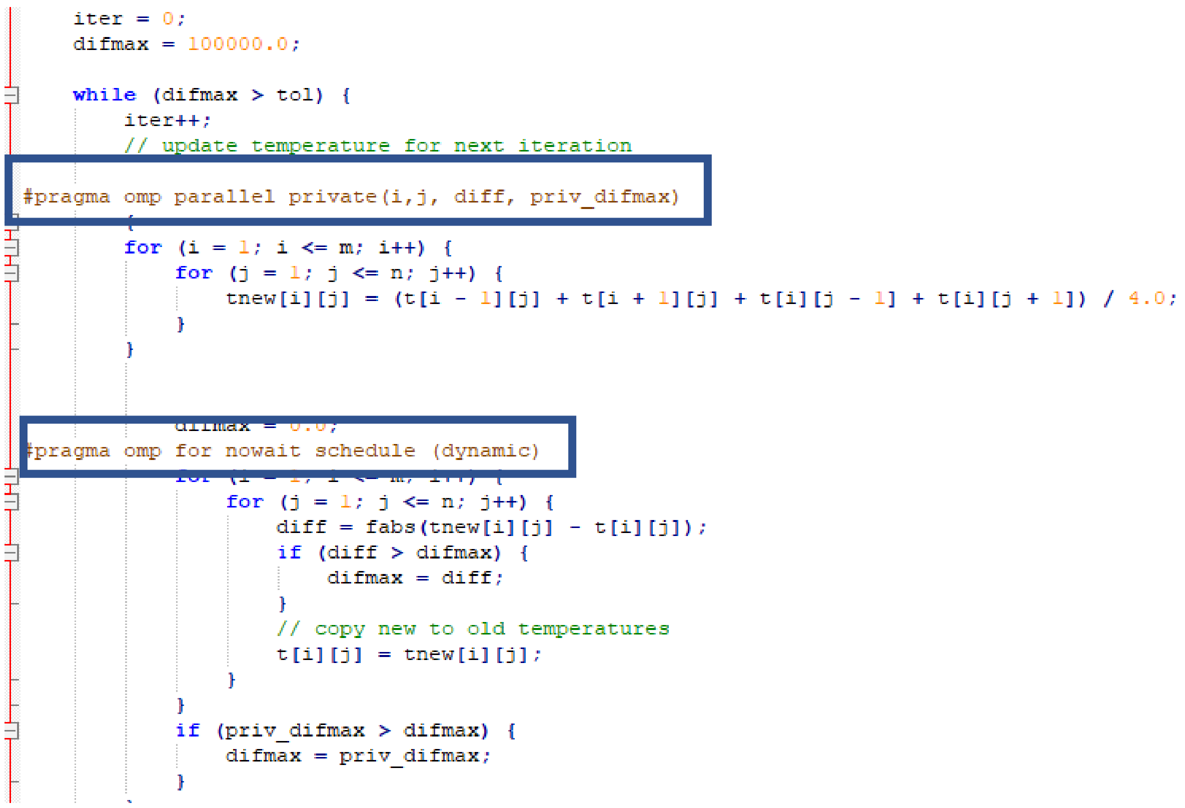


Figure 14 shows a graph of speed up in Gauss.

Task 4

In this task, further modifications were made to improve the parallel performance of this task. Please refer yourself to Figure 15 and Figure 16, these are the modifications of both codes. NOWAIT clause allows the thread to continue of execution by excluding the wait of other threads in the team to complete the region As this task was done on the last day, considering the performance issue, I could not alter these results for better and accurate results.

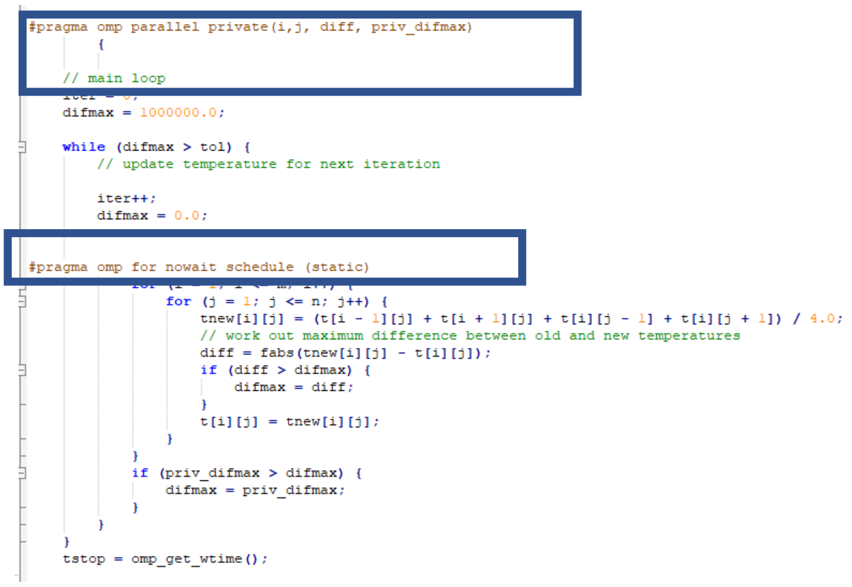
Adjustments Improvements Jacobi 2D



```
    iter = 0;
    difmax = 100000.0;

    while (difmax > tol) {
        iter++;
        // update temperature for next iteration
#pragma omp parallel private(i,j, diff, priv_difmax)
        {
        for (i = 1; i <= m; i++) {
            for (j = 1; j <= n; j++) {
                tnew[i][j] = (t[i - 1][j] + t[i + 1][j] + t[i][j - 1] + t[i][j + 1]) / 4.0;
            }
        }


            difmax = 0.0;
#pragma omp for nowait schedule (dynamic)
            for (i = 1; i <= m; i++) {
                for (j = 1; j <= n; j++) {
                    diff = fabs(tnew[i][j] - t[i][j]);
                    if (diff > difmax) {
                        difmax = diff;
                    }
                    // copy new to old temperatures
                    t[i][j] = tnew[i][j];
                }
            }
            if (priv_difmax > difmax) {
                difmax = priv_difmax;
            }
```

Figure 15 shows the modifications of what was made in Jacobi

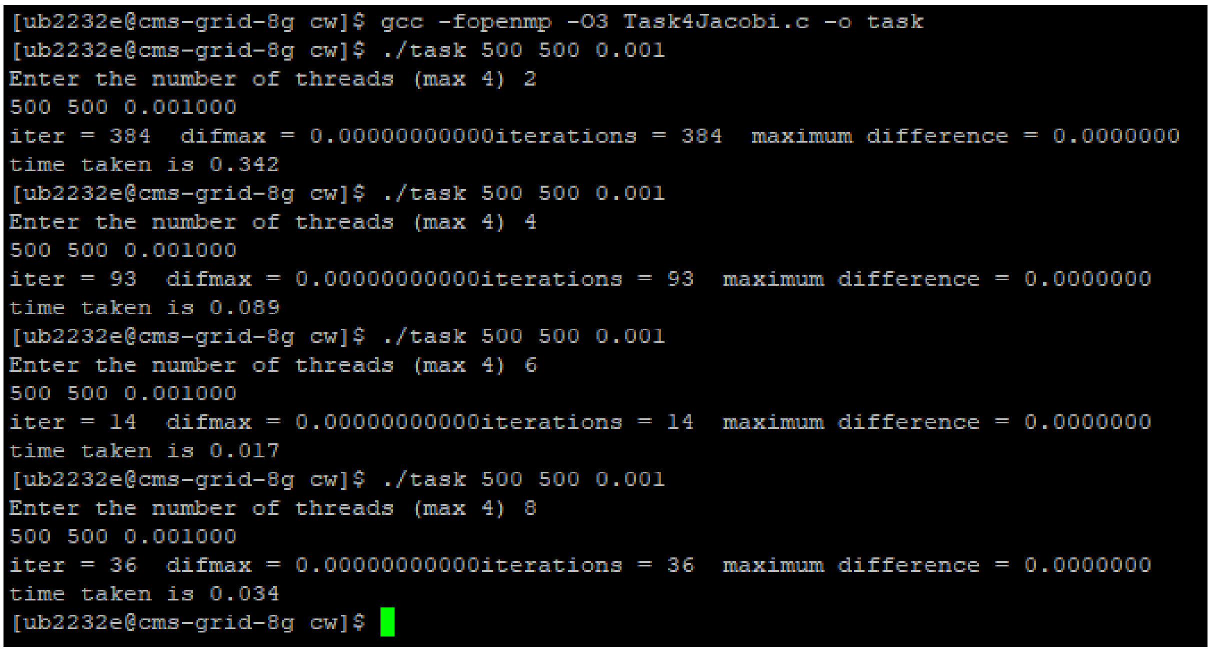
Adjustment Improvements Gauss 2D



```
#pragma omp parallel private(i,j, diff, priv_difmax)
        {

    // main loop
    iter = 0;
    difmax = 1000000.0;

    while (difmax > tol) {
        // update temperature for next iteration

        iter++;
        difmax = 0.0;


#pragma omp for nowait schedule (static)
            for (i = 1; i <= m; i++) {
                for (j = 1; j <= n; j++) {
                    tnew[i][j] = (t[i - 1][j] + t[i + 1][j] + t[i][j - 1] + t[i][j + 1]) / 4.0;
                    // work out maximum difference between old and new temperatures
                    diff = fabs(tnew[i][j] - t[i][j]);
                    if (diff > difmax) {
                        difmax = diff;
                    }
                    t[i][j] = tnew[i][j];
                }
            }
            if (priv_difmax > difmax) {
                difmax = priv_difmax;
            }
        }
    }
    tstop = omp_get_wtime();
```

Figure 16 shows the modifications of what was made in Gauss.

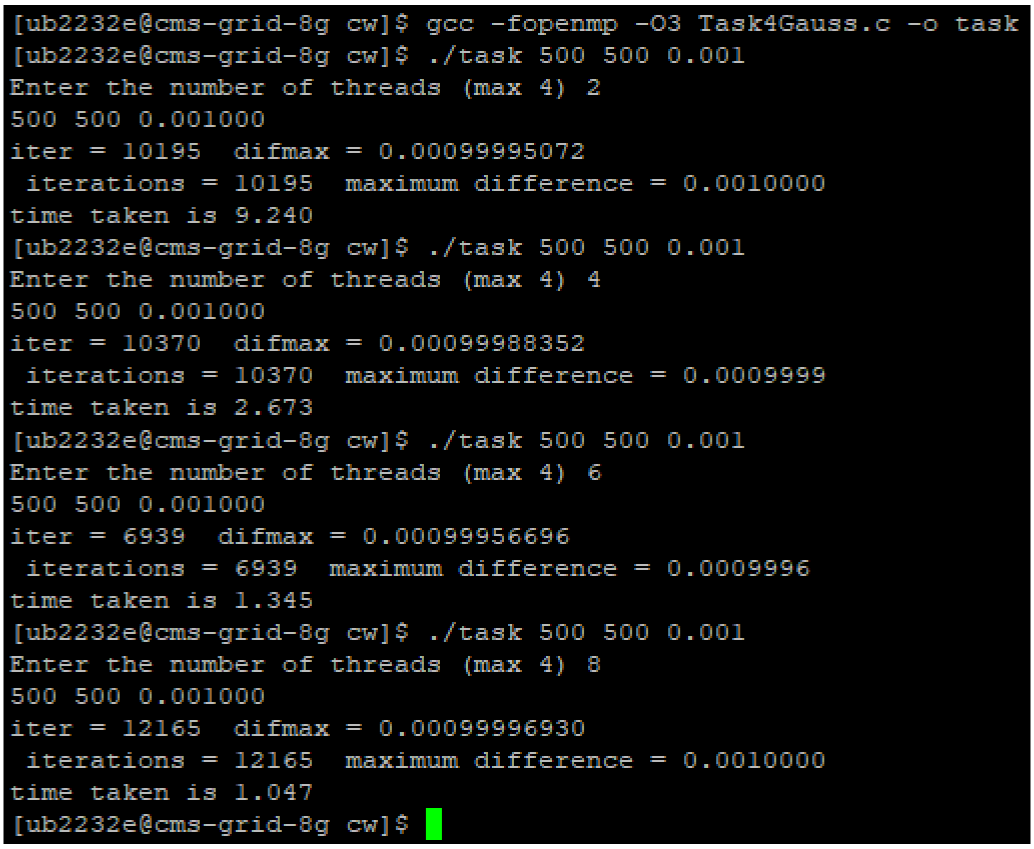
Screenshots Jacobi 2D Improvements



```
[ub2232e@cms-grid-8g cw]$ gcc -fopenmp -O3 Task4Jacobi.c -o task
[ub2232e@cms-grid-8g cw]$ ./task 500 500 0.001
Enter the number of threads (max 4) 2
500 500 0.001000
iter = 384  difmax = 0.00000000000iterations = 384  maximum difference = 0.0000000
time taken is 0.342
[ub2232e@cms-grid-8g cw]$ ./task 500 500 0.001
Enter the number of threads (max 4) 4
500 500 0.001000
iter = 93  difmax = 0.00000000000iterations = 93  maximum difference = 0.0000000
time taken is 0.089
[ub2232e@cms-grid-8g cw]$ ./task 500 500 0.001
Enter the number of threads (max 4) 6
500 500 0.001000
iter = 14  difmax = 0.00000000000iterations = 14  maximum difference = 0.0000000
time taken is 0.017
[ub2232e@cms-grid-8g cw]$ ./task 500 500 0.001
Enter the number of threads (max 4) 8
500 500 0.001000
iter = 36  difmax = 0.00000000000iterations = 36  maximum difference = 0.0000000
time taken is 0.034
[ub2232e@cms-grid-8g cw]$
```

Figure 17 shows an example of improvement within Jacobi

Screenshots Gauss 2D Improvements



```
[ub2232e@cms-grid-8g cw]$ gcc -fopenmp -O3 Task4Gauss.c -o task
[ub2232e@cms-grid-8g cw]$ ./task 500 500 0.001
Enter the number of threads (max 4) 2
500 500 0.001000
iter = 10195  difmax = 0.00099995072
 iterations = 10195  maximum difference = 0.0010000
time taken is 9.240
[ub2232e@cms-grid-8g cw]$ ./task 500 500 0.001
Enter the number of threads (max 4) 4
500 500 0.001000
iter = 10370  difmax = 0.00099988352
 iterations = 10370  maximum difference = 0.0009999
time taken is 2.673
[ub2232e@cms-grid-8g cw]$ ./task 500 500 0.001
Enter the number of threads (max 4) 6
500 500 0.001000
iter = 6939  difmax = 0.00099956696
 iterations = 6939  maximum difference = 0.0009996
time taken is 1.345
[ub2232e@cms-grid-8g cw]$ ./task 500 500 0.001
Enter the number of threads (max 4) 8
500 500 0.001000
iter = 12165  difmax = 0.00099996930
 iterations = 12165  maximum difference = 0.0010000
time taken is 1.047
[ub2232e@cms-grid-8g cw]$
```

Figure 18 shows an example of improvement within Gauss.

Jacobi 2D Test Cases

Jacobi 2D	500*500(s)	600*600(s)	700*700(s)
2		RESULTS	
TEST NO. 1	1.606	3.584	5.067
TEST NO. 2	1.285	3.489	8.308
TEST NO. 3	1.736	1.642	4.226
TEST NO. 4	2.628	3.851	5.510
TEST NO. 5	1.030	2.443	5.034
AVERAGE:	**1.657**	**3.0018**	**5.629**

Jacobi 2D	500*500(s)	600*600(s)	700*700(s)
4		RESULTS	
TEST NO. 1	0.767	1.078	1.552
TEST NO. 2	0.734	1.186	0.945
TEST NO. 3	0.404	0.678	0.886
TEST NO. 4	0.417	0.519	2.102
TEST NO. 5	0.168	1.003	0.765
AVERAGE:	**0.498**	**0.8928**	**1.25**

Jacobi 2D	500*500(s)	600*600(s)	700*700(s)
6		RESULTS	
TEST NO. 1	0.107	0.110	0.117
TEST NO. 2	0.033	0.145	0.125
TEST NO. 3	0.033	0.083	0.218
TEST NO. 4	0.028	0.077	0.190
TEST NO. 5	0.038	0.176	0.053
AVERAGE:	**0.0478**	**0.1182**	**0.1406**

Jacobi 2D	500*500(s)	600*600(s)	700*700(s)
8		RESULTS	
TEST NO. 1	0.033	0.016	0.035
TEST NO. 2	0.033	0.018	0.039
TEST NO. 3	0.024	0.013	0.027
TEST NO. 4	0.020	0.011	0.037
TEST NO. 5	0.005	0.023	0.045
AVERAGE:	**0.023**	**0.0162**	**0.0366**

Table 11 shows the results of Jacobi

Gauss 2D Test Cases

Gauss 2D	500*500(s)	600*600(s)	700*700(s)
2	RESULTS		
TEST NO. 1	9.521	14.772	25.102
TEST NO. 2	9.128	15.426	23.047
TEST NO. 3	9.531	15.305	23.577
TEST NO. 4	9.118	15.398	23.204
TEST NO. 5	9.229	15.158	23.230
AVERAGE:	**9.3054**	**15.2118**	**23.632**

Gauss 2D	500*500(s)	600*600(s)	700*700(s)
4	RESULTS		
TEST NO. 1	2.871	3.880	7.279
TEST NO. 2	2.820	4.992	6.572
TEST NO. 3	2.996	4.169	7.990
TEST NO. 4	3.135	3.855	7.297
TEST NO. 5	2.758	4.011	6.236
AVERAGE:	**2.916**	**4.1814**	**7.0748**

Gauss 2D	500*500(s)	600*600(s)	700*700(s)
6	RESULTS		
TEST NO. 1	1.540	2.372	3.267
TEST NO. 2	1.402	2.402	4.693
TEST NO. 3	1.391	2.114	4.048
TEST NO. 4	1.651	2.413	3.633
TEST NO. 5	1.335	2.335	4.049
AVERAGE:	**1.4638**	**2.3272**	**3.938**

Gauss 2D	500*500(s)	600*600(s)	700*700(s)
8	RESULTS		
TEST NO. 1	1.033	1.406	2.190
TEST NO. 2	0.853	1.656	2.174
TEST NO. 3	1.016	1.556	2.200
TEST NO. 4	0.921	1.408	2.401
TEST NO. 5	0.967	1.528	2.631
AVERAGE:	**0.958**	**1.5108**	**2.3192**

Table 12 shows the results of Gauss.

Referring to Table 11 and 12, these show the results of Jacobi and Gauss. As you may know, Task 3 improvements were to adjust from on average of 10s to run 500, to 1s. However, for Jacobi, these results fluctuating. This may be the case of testing this on the last day, due to a performance issue as more than one person would be using 8G. These results are too quick. However, the improvement for Gauss has improved drastically.

Comparison Jacobi and Gauss Average

Please refer yourself to Figure 19 and Figure 20 where this is a graph representation of the results of the test cases that were completed using improvements of Task 3. These are the average of each problem sizes that were used. This shows a great interpretation of each size is decreased as soon as more threads have been used. This shows that the more threads are used and worked together; it would enable the time to decrease. This is the same for both out comes.

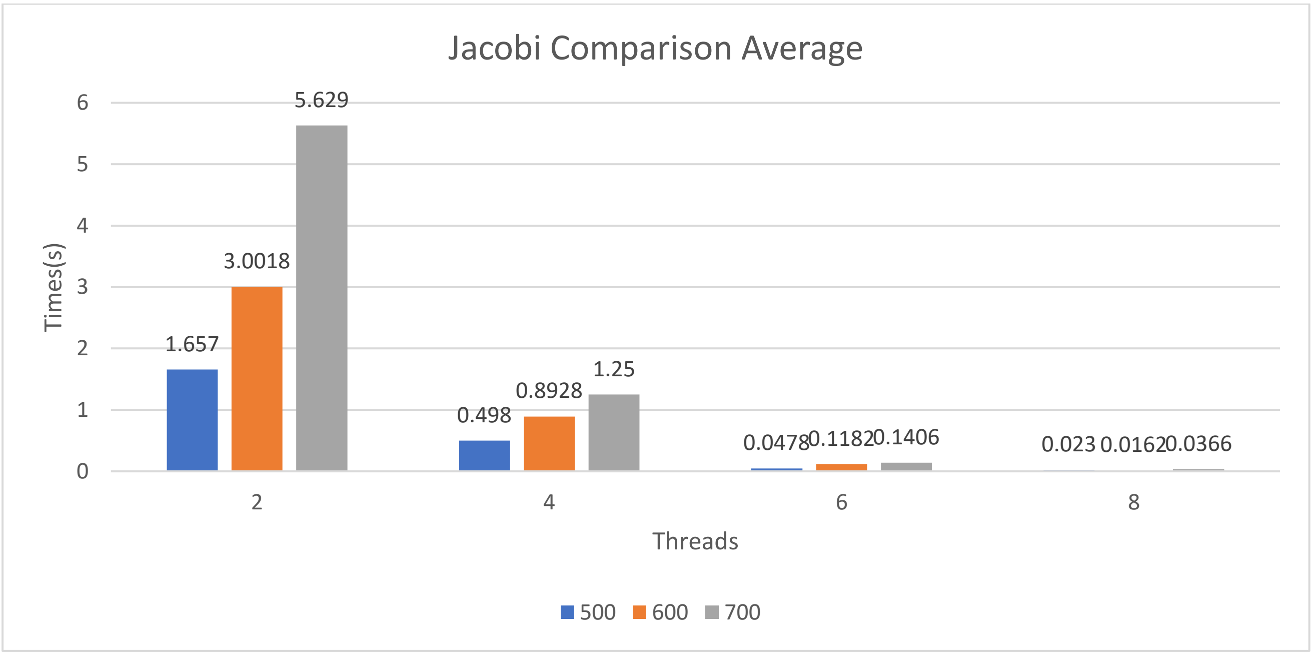


Figure 19 shows the Jacobi average of Task 4

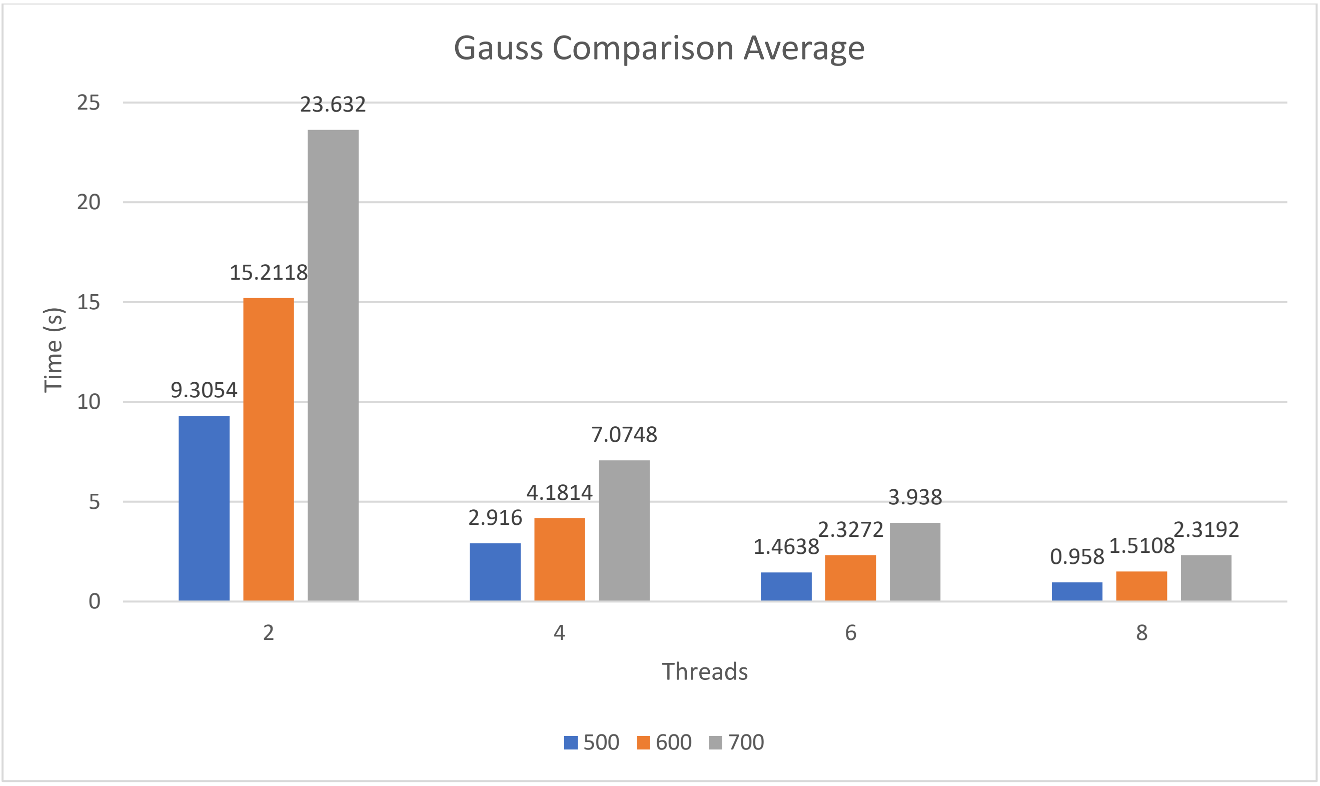


Figure 20 shows the Jacobi average of Task 4

Comparison of Task 3 and Task 4 Speedup

Figure 21 and Figure 22 show the speedup from Task 3. This has been done by simple calculation of subtracting each task from Task 3 average by Task 4. Here is the difference. As you can see, Figure 22 for Gauss has a bigger difference than Jacobi. This may be due to Jacobi was already having a quicker rate than Gauss. Therefore, when improvements were made; it had a smaller impact than Gauss.

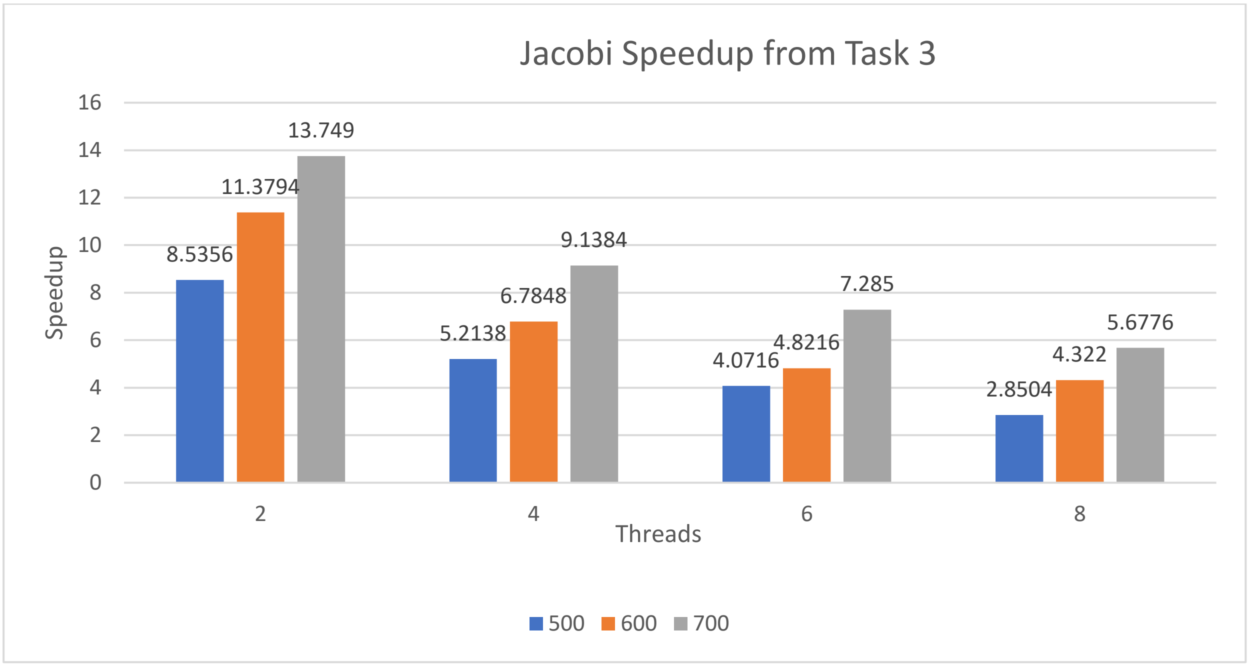


Figure 21 shows the speedup from Task 3 for Jacobi

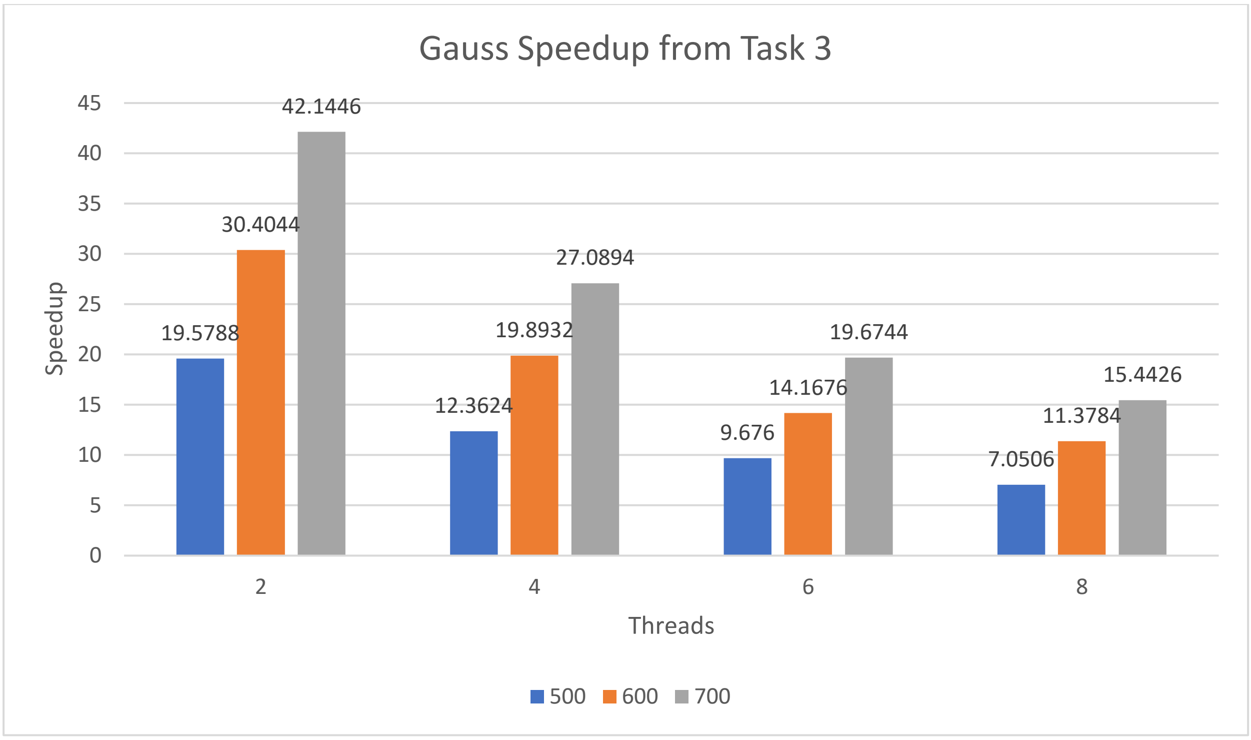


Figure 22 shows the speedup from Task 3 for Gauss

References

1. Barney, B. (2008). *Introduction to Parallel Computing*. [online] Computing.llnl.gov. Available at: https://computing.llnl.gov/tutorials/parallel_comp/ [Accessed 13 Dec. 2019].
2. Tonry, C. (2019). *Gauss 2D Code*. [Online] Moodle Current. Available at: https://moodlecurrent.gre.ac.uk/pluginfile.php/1585302/mod_resource/content/1/gauss2d.c [Accessed 12 Dec. 2019].
3. Tonry, C. (2019). *Jacobi2D Code* [Online] Moodle Current. Available at: https://moodlecurrent.gre.ac.uk/pluginfile.php/1585301/mod_resource/content/1/jacobi2d.c [Accessed 12 Dec. 2019].
4. Wiki Gentoo (2019). *GCC Optimization*. [online] Available at: https://wiki.gentoo.org/wiki/GCC_optimization [Accessed 13 Dec. 2019].
5. Wikipedia (2019). *Optimizing compiler*. [online] Available at: https://en.m.wikipedia.org/wiki/Optimizing_compiler [Accessed 13 Dec. 2019].