Program 1 Report

CS6376

Parallel Processing

Submitted by –

Ayush Agrawal (ara150430)

1. Problem Statement

A laplace equation applies to many physical problems like Electrostatistics, Fluid flow, Temperature. For temperature, it is the steady state heat equation in which each point on grid is the average of its neighbors. We can converge to Final steady state from initial state by repeatedly computing new values at each point from the average of neighboring points. We keep doing this until the difference from one pass to the next is small enough to tolerate. The goal here is to modify an existing serial laplace solver such that its performance is improved.

2. Approach to Solution

Computing temperature for each point in grid at iteration in a laplace solver involves looping over entire grid points. Since we compute value for each point in grid independently of other points in that iteration, we can use techniques from parallel programming to improve the performance of laplace solver.

The given laplace solver code is in C programming language. To improve performance, I am using OpenMP with C to execute code in parallel. The main idea is to add OpenMP based pragma directives to code wherever possible such that the code runs parallel and performance is improved. For the purpose of this assignment I am using the bridges environment provided by Pittsburgh Supercomputing Center.

3. Solution Description

3.1 Adding OpenMP pragmas

Added 2 pragmas in the program first before-

```
for(i = 1; i <= ROWS; i++) {
      for(j = 1; j <= COLUMNS; j++) {
        Temperature[i][j] = 0.25 * (Temperature_last[i+1][j] + Temperature_last[i-1][j] +
                          Temperature_last[i][j+1] + Temperature_last[i][j-1]);
      }
   }
```

And second before this reduction block –

```
for(i = 1; i <= ROWS; i++){
      for(j = 1; j <= COLUMNS; j++){
          dt = fmax( fabs(Temperature[i][j]-Temperature_last[i][j]), dt);
          Temperature_last[i][j] = Temperature[i][j];
      }
```

}

The two pragmas added were –

1 #pragma omp parallel for private(i,j) - for first block shown above

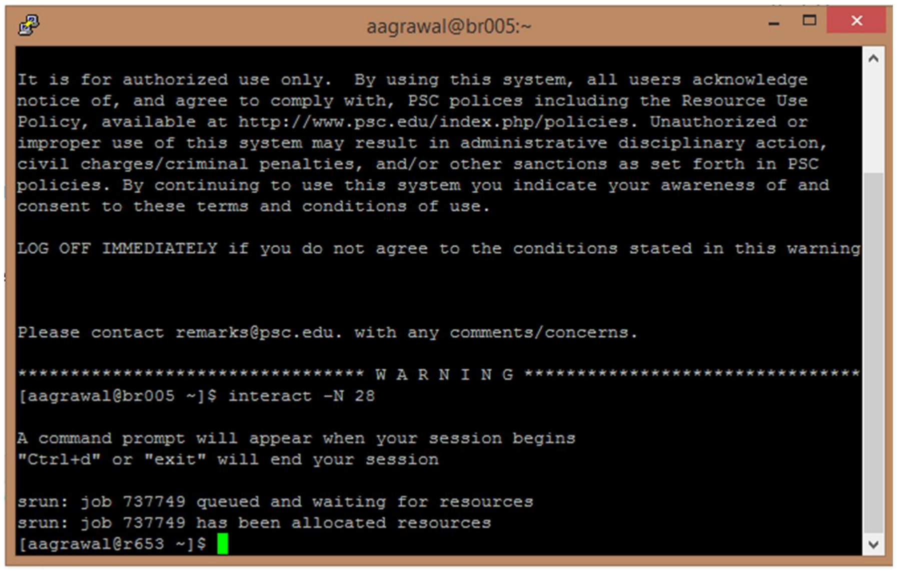
2 #pragma omp parallel for reduction(max:dt) private(i,j) – for second reduction based block shown above

3.2 Compiling and Building the solution

Step 1 – log into bridges using ssh username@bridges.psc.edu

Step 2 – enter password

Step 3 – use interact –N 28 command to access and allocate resources to use.



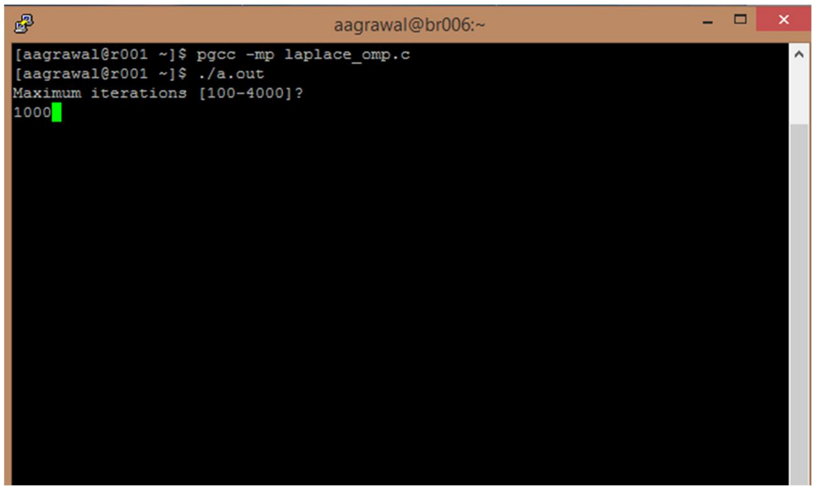
```
It is for authorized use only.  By using this system, all users acknowledge
notice of, and agree to comply with, PSC polices including the Resource Use
Policy, available at http://www.psc.edu/index.php/policies. Unauthorized or
improper use of this system may result in administrative disciplinary action,
civil charges/criminal penalties, and/or other sanctions as set forth in PSC
policies. By continuing to use this system you indicate your awareness of and
consent to these terms and conditions of use.

LOG OFF IMMEDIATELY if you do not agree to the conditions stated in this warning



Please contact remarks@psc.edu. with any comments/concerns.

******************************** W A R N I N G ********************************
[aagrawal@br005 ~]$ interact -N 28

A command prompt will appear when your session begins
"Ctrl+d" or "exit" will end your session

srun: job 737749 queued and waiting for resources
srun: job 737749 has been allocated resources
[aagrawal@r653 ~]$
```

Fig1 – usage of interact command, here –N 28 indicates 28 cores are requested for usage

Step 4 – Edit the source code using utilities like vi or emacs.

Step 5 – Compile the program using pgcc –mp filename.c

Step 6 – If you want to execute the code serially use ./a.out on command prompt.



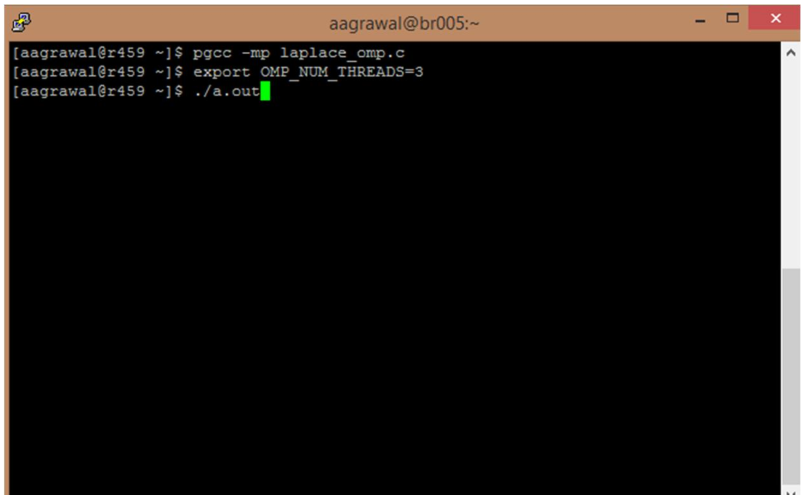
```
[aagrawal@r001 ~]$ pgcc -mp laplace_omp.c
[aagrawal@r001 ~]$ ./a.out
Maximum iterations [100-4000]?
1000
```

Fig2 – Compiling and running the program serially

Step 7 - if you want to run the program in parallel on multiple cores use command

export OMP_NUM_THREADS = x

Where x is the number of cores you want to use between 1 and 28. Followed by ./a.out to execute the program.

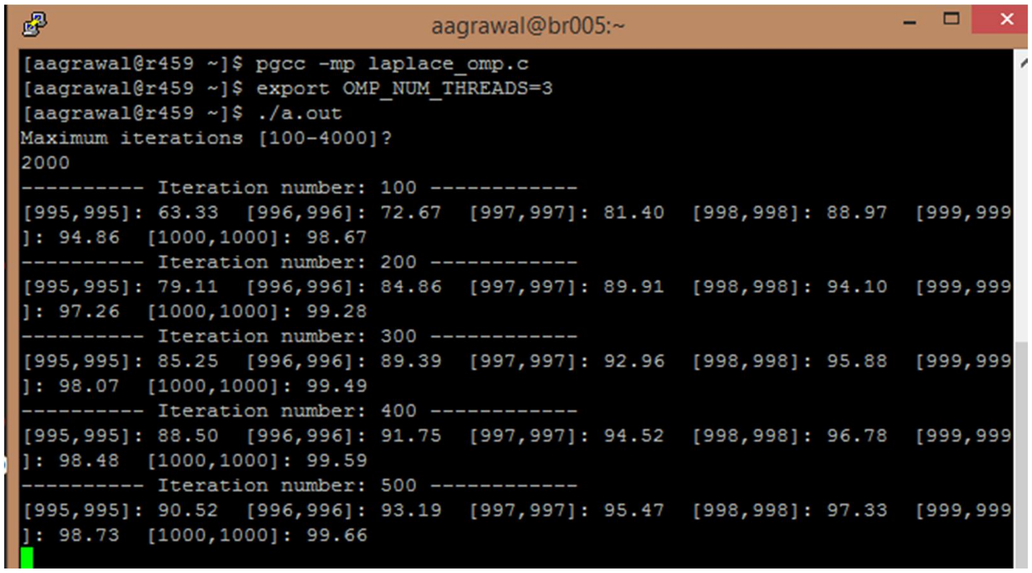


```
[aagrawal@r459 ~]$ pgcc -mp laplace_omp.c
[aagrawal@r459 ~]$ export OMP_NUM_THREADS=3
[aagrawal@r459 ~]$ ./a.out
```

Fig3 – compiling and running using multiple cores, in this case 3

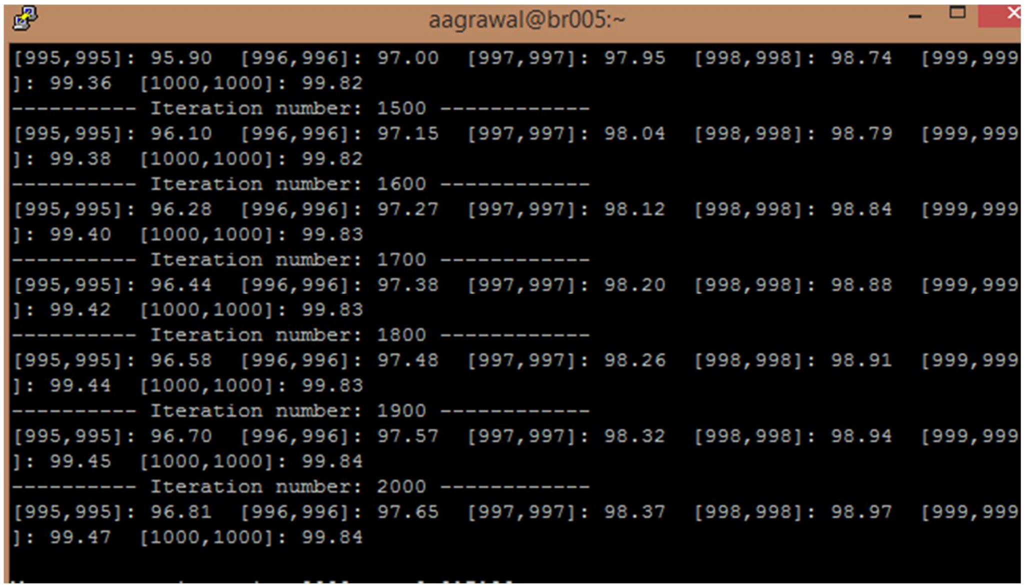
3.3 Some screen screenshots

Case 1 – Running on 3 cores for 2000 iteration



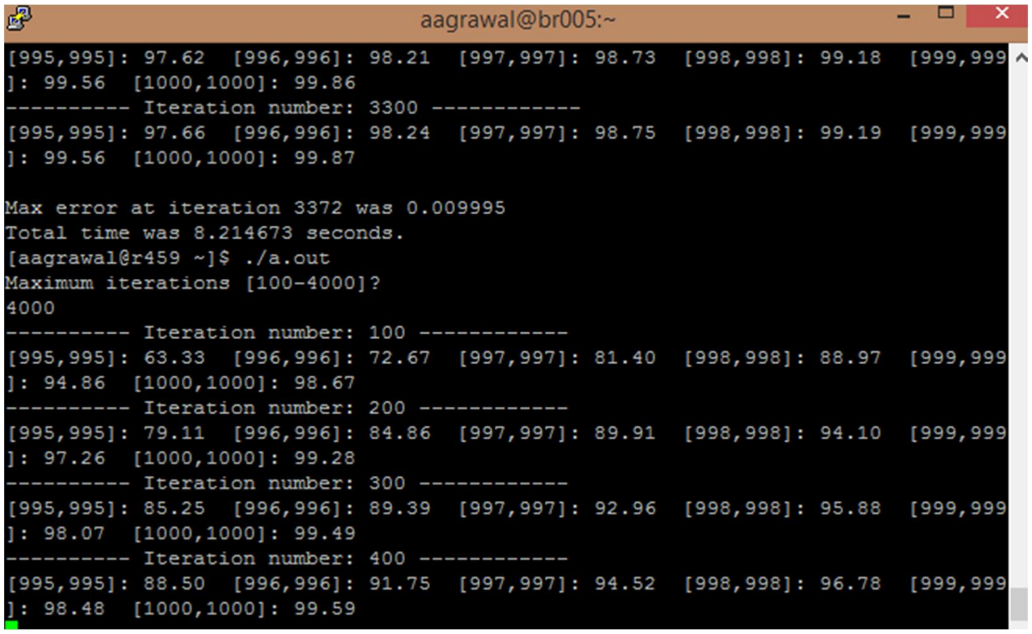
```
[aagrawal@r459 ~]$ pgcc -mp laplace_omp.c
[aagrawal@r459 ~]$ export OMP_NUM_THREADS=3
[aagrawal@r459 ~]$ ./a.out
Maximum iterations [100-4000]?
2000
---------- Iteration number: 100 ------------
[995,995]: 63.33  [996,996]: 72.67  [997,997]: 81.40  [998,998]: 88.97  [999,999
]: 94.86  [1000,1000]: 98.67
---------- Iteration number: 200 ------------
[995,995]: 79.11  [996,996]: 84.86  [997,997]: 89.91  [998,998]: 94.10  [999,999
]: 97.26  [1000,1000]: 99.28
---------- Iteration number: 300 ------------
[995,995]: 85.25  [996,996]: 89.39  [997,997]: 92.96  [998,998]: 95.88  [999,999
]: 98.07  [1000,1000]: 99.49
---------- Iteration number: 400 ------------
[995,995]: 88.50  [996,996]: 91.75  [997,997]: 94.52  [998,998]: 96.78  [999,999
]: 98.48  [1000,1000]: 99.59
---------- Iteration number: 500 ------------
[995,995]: 90.52  [996,996]: 93.19  [997,997]: 95.47  [998,998]: 97.33  [999,999
]: 98.73  [1000,1000]: 99.66
```

Case 2 – Result of above case1 after execution



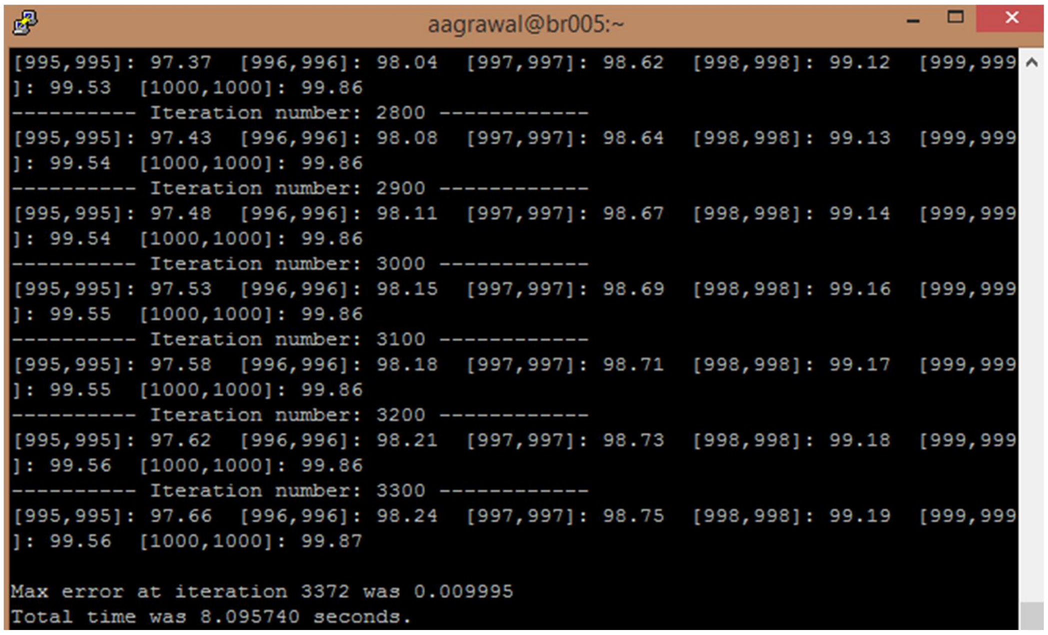
```
[995,995]: 95.90  [996,996]: 97.00  [997,997]: 97.95  [998,998]: 98.74  [999,999
]: 99.36  [1000,1000]: 99.82
---------- Iteration number: 1500 ------------
[995,995]: 96.10  [996,996]: 97.15  [997,997]: 98.04  [998,998]: 98.79  [999,999
]: 99.38  [1000,1000]: 99.82
---------- Iteration number: 1600 ------------
[995,995]: 96.28  [996,996]: 97.27  [997,997]: 98.12  [998,998]: 98.84  [999,999
]: 99.40  [1000,1000]: 99.83
---------- Iteration number: 1700 ------------
[995,995]: 96.44  [996,996]: 97.38  [997,997]: 98.20  [998,998]: 98.88  [999,999
]: 99.42  [1000,1000]: 99.83
---------- Iteration number: 1800 ------------
[995,995]: 96.58  [996,996]: 97.48  [997,997]: 98.26  [998,998]: 98.91  [999,999
]: 99.44  [1000,1000]: 99.83
---------- Iteration number: 1900 ------------
[995,995]: 96.70  [996,996]: 97.57  [997,997]: 98.32  [998,998]: 98.94  [999,999
]: 99.45  [1000,1000]: 99.84
---------- Iteration number: 2000 ------------
[995,995]: 96.81  [996,996]: 97.65  [997,997]: 98.37  [998,998]: 98.97  [999,999
]: 99.47  [1000,1000]: 99.84
```

Case 3 – Running on 3 cores with 4000 iterations



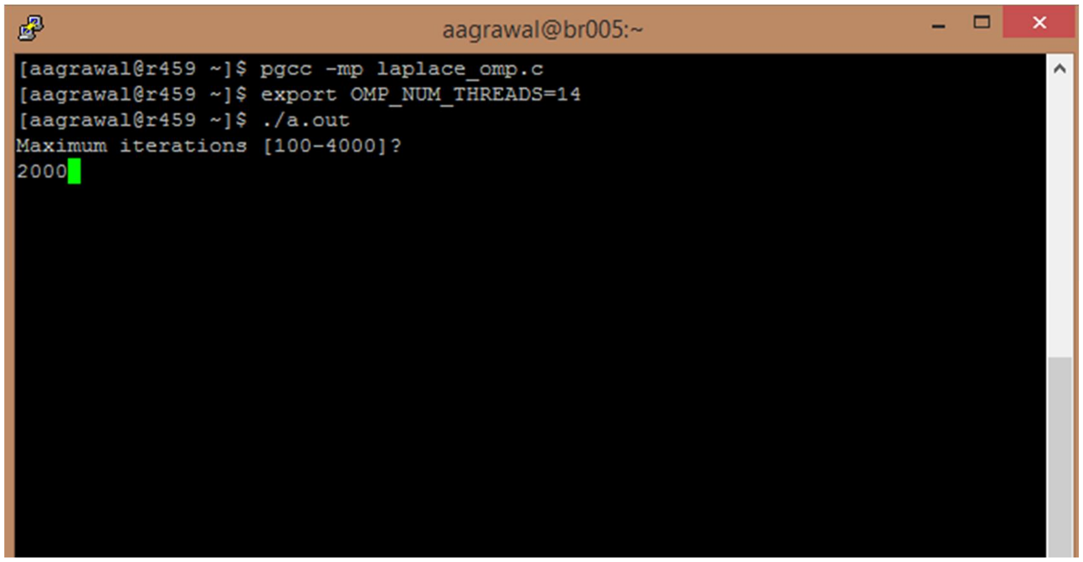
```
[995,995]: 97.62  [996,996]: 98.21  [997,997]: 98.73  [998,998]: 99.18  [999,999
]: 99.56  [1000,1000]: 99.86
---------- Iteration number: 3300 ------------
[995,995]: 97.66  [996,996]: 98.24  [997,997]: 98.75  [998,998]: 99.19  [999,999
]: 99.56  [1000,1000]: 99.87

Max error at iteration 3372 was 0.009995
Total time was 8.214673 seconds.
[aagrawal@r459 ~]$ ./a.out
Maximum iterations [100-4000]?
4000
---------- Iteration number: 100 ------------
[995,995]: 63.33  [996,996]: 72.67  [997,997]: 81.40  [998,998]: 88.97  [999,999
]: 94.86  [1000,1000]: 98.67
---------- Iteration number: 200 ------------
[995,995]: 79.11  [996,996]: 84.86  [997,997]: 89.91  [998,998]: 94.10  [999,999
]: 97.26  [1000,1000]: 99.28
---------- Iteration number: 300 ------------
[995,995]: 85.25  [996,996]: 89.39  [997,997]: 92.96  [998,998]: 95.88  [999,999
]: 98.07  [1000,1000]: 99.49
---------- Iteration number: 400 ------------
[995,995]: 88.50  [996,996]: 91.75  [997,997]: 94.52  [998,998]: 96.78  [999,999
]: 98.48  [1000,1000]: 99.59
```

Case 4 – Result of above case 3



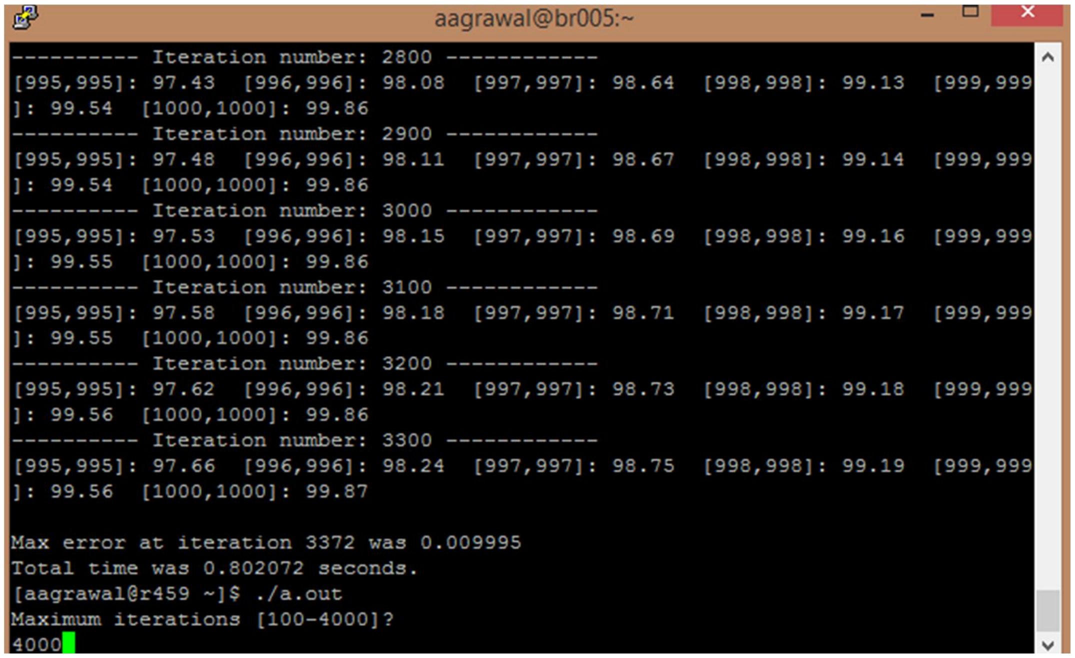
```
[995,995]: 97.37  [996,996]: 98.04  [997,997]: 98.62  [998,998]: 99.12  [999,999
]: 99.53  [1000,1000]: 99.86
---------- Iteration number: 2800 ------------
[995,995]: 97.43  [996,996]: 98.08  [997,997]: 98.64  [998,998]: 99.13  [999,999
]: 99.54  [1000,1000]: 99.86
---------- Iteration number: 2900 ------------
[995,995]: 97.48  [996,996]: 98.11  [997,997]: 98.67  [998,998]: 99.14  [999,999
]: 99.54  [1000,1000]: 99.86
---------- Iteration number: 3000 ------------
[995,995]: 97.53  [996,996]: 98.15  [997,997]: 98.69  [998,998]: 99.16  [999,999
]: 99.55  [1000,1000]: 99.86
---------- Iteration number: 3100 ------------
[995,995]: 97.58  [996,996]: 98.18  [997,997]: 98.71  [998,998]: 99.17  [999,999
]: 99.55  [1000,1000]: 99.86
---------- Iteration number: 3200 ------------
[995,995]: 97.62  [996,996]: 98.21  [997,997]: 98.73  [998,998]: 99.18  [999,999
]: 99.56  [1000,1000]: 99.86
---------- Iteration number: 3300 ------------
[995,995]: 97.66  [996,996]: 98.24  [997,997]: 98.75  [998,998]: 99.19  [999,999
]: 99.56  [1000,1000]: 99.87

Max error at iteration 3372 was 0.009995
Total time was 8.095740 seconds.
```

Case 5 – Running on 14 cores



```
[aagrawal@r459 ~]$ pgcc -mp laplace_omp.c
[aagrawal@r459 ~]$ export OMP_NUM_THREADS=14
[aagrawal@r459 ~]$ ./a.out
Maximum iterations [100-4000]?
2000
```

Case 6 – Result of running on 28 cores with 4000 iteration



```
---------- Iteration number: 2800 ------------
[995,995]: 97.43  [996,996]: 98.08  [997,997]: 98.64  [998,998]: 99.13  [999,999
]: 99.54  [1000,1000]: 99.86
---------- Iteration number: 2900 ------------
[995,995]: 97.48  [996,996]: 98.11  [997,997]: 98.67  [998,998]: 99.14  [999,999
]: 99.54  [1000,1000]: 99.86
---------- Iteration number: 3000 ------------
[995,995]: 97.53  [996,996]: 98.15  [997,997]: 98.69  [998,998]: 99.16  [999,999
]: 99.55  [1000,1000]: 99.86
---------- Iteration number: 3100 ------------
[995,995]: 97.58  [996,996]: 98.18  [997,997]: 98.71  [998,998]: 99.17  [999,999
]: 99.55  [1000,1000]: 99.86
---------- Iteration number: 3200 ------------
[995,995]: 97.62  [996,996]: 98.21  [997,997]: 98.73  [998,998]: 99.18  [999,999
]: 99.56  [1000,1000]: 99.86
---------- Iteration number: 3300 ------------
[995,995]: 97.66  [996,996]: 98.24  [997,997]: 98.75  [998,998]: 99.19  [999,999
]: 99.56  [1000,1000]: 99.87

Max error at iteration 3372 was 0.009995
Total time was 0.802072 seconds.
[aagrawal@r459 ~]$ ./a.out
Maximum iterations [100-4000]?
4000
```

3.4 Performance Measures

For computing all the measures or metrics ROW and COLUMN size of grid is 1000 and number of iterations used are 4000.

The below shown table shows performance measure in GFLOPs/sec for the code on various configurations. It is computed using formulae-

$$\text{Performance} = 5 * \text{ROW} * \text{COLUMN} * \text{number of iteration} / \text{execution time} * 10^9$$

Here, execution time is the amount of time it took to run under that configuration.

Number of cores used	Execution time is sec	Performance in GFLOPs/sec
1	14.6	1.36
2	12.0	1.65
3	8.0	2
4	6.0	3.39
5	5.0	4
6	4.0	5
7	3.5	5.71
8	3.07	6.5
10	2.4	8.33
12	2.06	9.7
14	1.74	11.49
20	1.21	16.52
28	0.8	25

Second measure is Speedup, which is computed using formulae -

Speedup = execution time on one core (serial execution time)/ execution time in parallel on multiple core

Below table shows the speedup value in case of various number of cores.

Number of cores used	Speedup
2	1.2
3	1.825
4	2.47
5	2.92
6	3.65
7	4.17
8	4.75
10	6.08
12	7.19
14	8.39
20	12.06
28	18.25

Third measure is Karp Flatt metric which computes experimentally determined serial function e.

The formulae to compute e is –

$$e = (1/\ speedup - 1/p) / (1 - 1/p)$$

Here p is the number of cores used and speedup is the speedup corresponding to that number of cores.

Below table shows the value for e using karp flatt metric

Number of cores used	Speedup	e
2	1.2	0.833
3	1.825	0.555
4	2.47	0.40
5	2.92	0.34
6	3.65	0.24
7	4.17	0.23
8	4.75	0.21
10	6.08	0.164
12	7.19	0.13
14	8.39	0.11
20	12.06	0.08
28	18.25	0.05

From above results we can conclude that the performance of laplace solver increases if we run it parallel on multiple cores. The performance increases with the increase in number of cores.