A PROJECT REPORT ON
"PARALLELIZATION OF THE STEADY STATE HEAT EQUATION"

COURSE:
CS 301 : High Performance Computing

UNDER THE GUIDANCE OF
PROF. BHASKAR CHAUDHURY

Authors:

Abhin Kakkad	201501419
Manthan Mehta	201501420

1 Context

1.1 Brief Description of the Problem

The problem involves the parallelization of the Heat Diffusion in 2D Square Plate using central average method with steady-state solution. Finding out the steady state of a system is useful in physical as well as chemical processes. E.g. Finding out the steady state in chemical reactions is a very important part of the process because we need to know at what point will the system come to a stable state. Here, we have tried to find out the stable state of a 2-dimensional plate using the steady state equation.

1.2 Complexity of the Algorithm

Suppose the plate has dimensions N x N (supposing it is a square plate) then the time required to traverse all the cells of a plate would be of the order O(N*N). Although the boundary points can be neglected since they have a fixed temperature, but if we take a very large plate then those points are very few in number as compared to the number of grid points so the time complexity for determining the temperature for all points would be O(N*N). For determining the temperature of all points we need to solve a second order differential equation which we would do so using the finite difference method.

1.3 Hardware Details

We ran this problem on a cluster of 12 nodes. The hardware details are:

- Architecture: x86_64
- CPU op-mode(s): 32-bit, 64-bit
- Byte Order: Little Endian
- CPU(s): 12
- On-line CPU(s) list: 0-11
- Thread(s) per core: 1
- Core(s) per socket: 6
- Socket(s): 2
- NUMA node(s): 2
- Vendor ID: GenuineIntel
- CPU family: 6
- Model: 63
- Model name: Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz
- Stepping: 2
- CPU MHz: 1225.406
- BogoMIPS: 4804.57
- Virtualization: VT-x
- L1d cache: 32K
- L1i cache: 32K
- L2 cache: 256K
- L3 cache: 15360K
- NUMA node0 CPU(s): 0-5
- NUMA node1 CPU(s): 6-11

1.4 Real World Applications

1) Thermal Diffusivity in Polymers

A direct practical application of the heat equation, in conjunction with the Fourier Theory is the prediction of thermal transfer profiles and the measurement of the thermal diffusivity in polymers. This method is applicable to rubber and various other polymeric materials of practical interest and micro fluids. In this equation we have a 3D model for the heat transfer. This equation has found its applications in energy transfer and thermal modeling.

*Reference: Unsworth, J.; Duarte, F. J. (1979), "Heat diffusion in a solid sphere and Fourier Theory", Am. J. Phys., **47** (11): 891–893*

2) Modeling of Options

The heat equation helps in the modeling of many phenomena and is used in modeling of options in stock markets. The famous Black-Scholes option pricing model can be transformed into the heat equation allowing easy solution from a familiar body of mathematics.

Reference: Wilmott, P.; Howison, S.; Dewynne, J. (1995), The Mathematics of Financial Derivatives:A Student Introduction, Cambridge University Press

3) Particle Diffusion

One can model particle diffusion by an equation involving:

- ➔ the volumetric concentration of particles, in the case of collective diffusion of a large number of particles
- ➔ The probability density function associated with the position of a single particle

For both the above cases the heat equation is used.

4) Other Applications:

The equation describing pressure diffusion in a porous medium is identical in form with the heat equation. The heat equation is also widely used in image analysis and in machine-learning as the driving theory behind Laplacian methods. An abstract form of heat equation provides a major approach to Atiyah-Singer index theorem and has led to much further work on heat equations in Riemannian geometry.

Reference: Carslaw, H. S.; Jaeger, J. C. (1959), Conduction of Heat in Solids (2nd ed.), Oxford University Press, ISBN 978-0-19-853368-9

2 Serial Algorithm

2.1 Serial Pseudo Code

1) Set the boundary conditions such that the interior points have a reasonable value to start with.
2) Calculate the average for the interior points.
3) Iterate the loop for all the M*N points such that the older value of that point doesn't differ from the new value by more than the error tolerance.
    a) This we do by selecting each and every square and calculating the mean temperature (similar to the method followed in image processing) amongst its neighbour points.
    b) We check the error tolerance in respect to the grid point which has the maximum difference in temperature as compared to the previous value.
    c) We then update the previous value of the temperature of each grid point with the new value for the next iteration.

2.2 Algorithm

Initially the plate has a certain temperature on three of its edges. Thus we have an initial condition set for the plate, also we would have the boundary conditions for the plate. This will provide us with a definite average temperature for the interior points of the plate.

For each of the interior points there would be nine neighbouring points except for the boundary points. For calculating the temperature of any point we calculate the average of the four points around it, i.e. the points to its north, south, east and west. The method used here is the central mean theorem.

$$W[Central] = (1/4) * (W[North] + W[South] + W[East] + W[West])$$

Where W[i,j] is the temperature at (i,j) th point in the M*N grid.

The point taken into consideration can be seen in the figure 2.2.1:

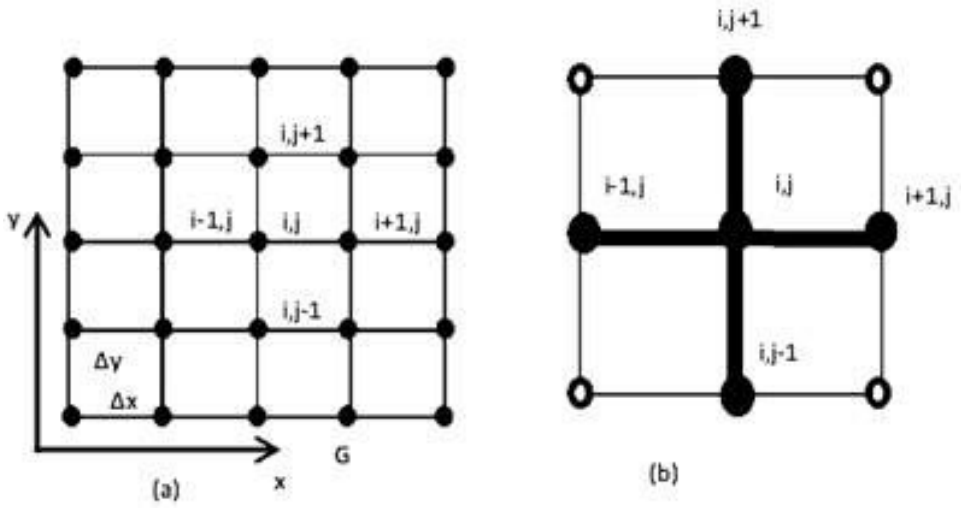

Fig: 2.2.1

Reference: http://article.sciencepublishinggroup.com/html/10.11648.j.dmath.20160101.12.html

For every iteration we then update the value of the temperature values at that point. We keep increasing the iterations until the temperature difference in the current iteration and the previous iteration isn't less than or equal to the error tolerance that was set by us.

When such a point is reached we can say that the system has reached a steady state and there would be no heat transfer between the individual points on the grids and more or less all the points on the grid would have the same temperature.

2.3 Optimising the Serial Code

One of the major optimisation would be in the **central** function. The central function calculates the average value of temperature for a grid point by taking the mean of the point surrounding it. Further optimisation of that function isn't possible because this is the version that we implemented after optimizing the stochastic algorithm for finding the temperature of the plate.

In that equation we had to solve a differential equation and to avoid that we have implemented the average function.

3 Parallelizing the Algorithm

3.1 Profiling Information and Possible SpeedUp

We have used gprof for profiling and these are the analysed results (for different input sizes):

1) Grid size: 500*500

    a) From the call graph (Fig: 3.2.2) function we saw that the central( ) part of the code took 49.7% of the time and from Amdahl's Law the maximum speedup that we get can be obtained for 12 threads comes out to be 1.9. We got a value greater than that in experimental results.

2) Grid size: 300*300

    a) From the call graph function we saw that the central( ) part of the code took 49% of the time and from Amdahl's Law the maximum speedup that we get can be obtained for 12 threads comes out to be 1.85. We got a value greater than that in experimental results.

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 49.80     33.72    33.72    17103     1.97     1.97  central
 26.60     51.73    18.01                             main
 23.80     67.85    16.11    17103     0.94     0.94  update
```

Fig: 3.1.1 Flat profile

```
granularity: each sample hit covers 2 byte(s) for 0.01% of 67.85 seconds

index % time    self  children    called     name
                                               <spontaneous>
[1]    100.0   18.01   49.83                 main [1]
                33.72    0.00   17103/17103       central [2]
                16.11    0.00   17103/17103       update [3]
-----------------------------------------------
                33.72    0.00   17103/17103       main [1]
[2]     49.7   33.72    0.00   17103         central [2]
-----------------------------------------------
                16.11    0.00   17103/17103       main [1]
[3]     23.7   16.11    0.00   17103         update [3]
-----------------------------------------------
```

Fig: 3.1.2 Call Graph

3.2 Parallel Pseudo Code

1) Set the boundary conditions such that the interior points have a reasonable value to start with.
2) Calculate the average for the interior points.
3) Distribute amongst **p** threads.
4) Iterate the loop for all the M*N points such that the older value of that point doesn't differ from the new value by more than the error tolerance.
    a) This we do by selecting each and every square and calculating the mean temperature (similar to the method followed in image processing) amongst its neighbour points.
    b) We check the error tolerance in respect to the grid point which has the maximum difference in temperature as compared to the previous value.
    c) We then update the previous value of the temperature of each grid point with the new value for the next iteration.

3.3 Optimization Strategy

There aren't many functions which require parallelization. The main time that was spent was in the count function which calculated the average. Applying the pragma for as shown below we were able to achieve the speedup as expected.

```
# pragma omp parallel shared ( u, w ) private ( i, j )
{
# pragma omp for
      for ( i = 1; i < M - 1; i++ )
      {
        for ( j = 1; j < N - 1; j++ )
        {
          w[i][j] = ( u[i-1][j] + u[i+1][j] + u[i][j-1] + u[i][j+1] ) / 4.0;
        }
      }
    }
}
```

The average calculation is an iterative method and with each iteration the values are updated for all the grid points. This results in a lot of time taken by each iteration. Hence all the loops have been parallelized.

4 Results and Observations

The input data was the initial conditions and the boundary conditions. The number of grid points i.e. the size of the square plate will affect the running time of the code. So we have considered the grid size parameter to get an idea about the variation of speedup with change in the problem size.

4.1 Speedup vs Cores

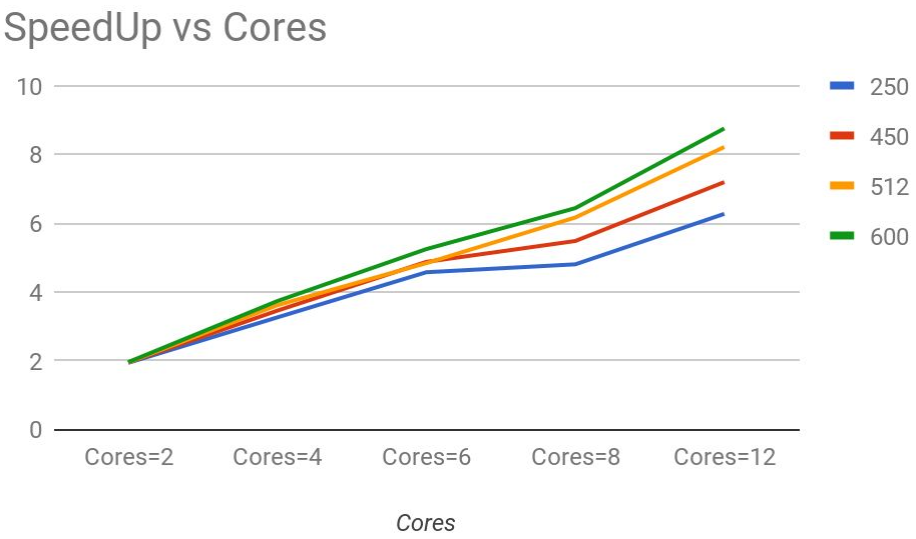

Fig.4.1.1 Speedup Vs. Cores

- We can see that as the number of threads increases, speedup increases. Increasing the number of grid size is nothing but increasing the problem size and the speedup increases as we increase the problem size.
- After 6 cores the speedup tends to flatten out, the reason for the same may be the hardware used by us. As it could be seen in the hardware specification there are only 2 sockets and 6 cores per socket, so for more than 6 cores the inter-socket communication may hinder the speedup.

4.2 Speedup vs ProblemSize

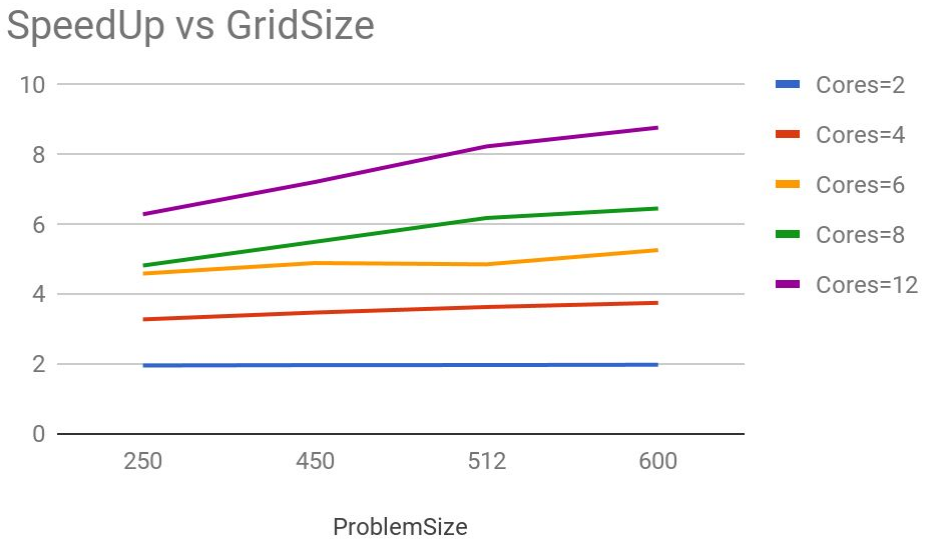

Fig.4.2.1 Speedup vs ProblemSize

- The speedup increases with the increase in number of cores. The speedup obtained is nearly equal to the number of cores (as expected) up till 6 cores. After that the speedup is high but not equal to the number of cores.
- The reason being as the cores increase the overheads for communication also increase. This may be one reason for decrease in the speedup.

4.3 Efficiency vs No. Of Cores

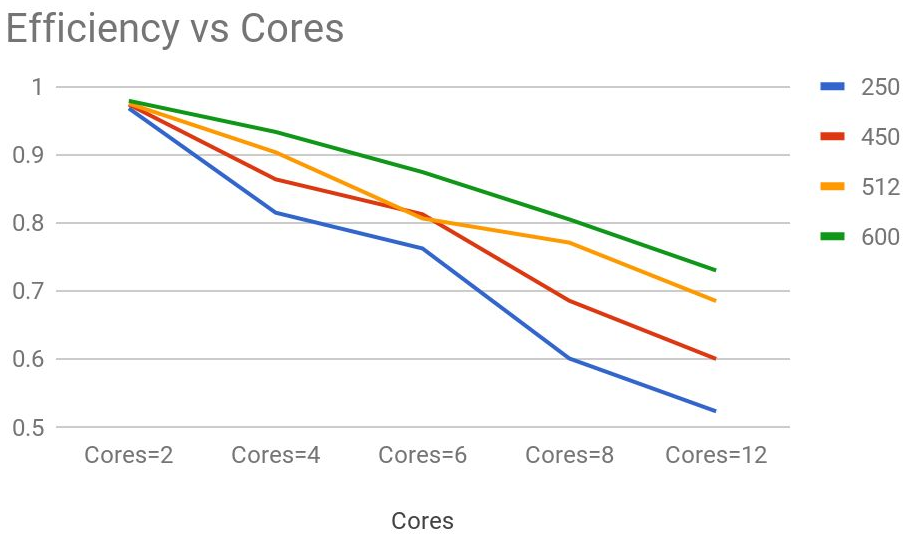

Fig.4.3.1 Efficiency vs Cores

- The efficiency is decreasing as the number of cores increase.
- As the grid size i.e. the problem size increases, the efficiency increases.
- We can conclude that the problem is weakly scalable. For weakly scalable problems, the efficiency remains nearly the same as number of processors increases by increasing the problem size.

4.4 Karp Flatt Analysis

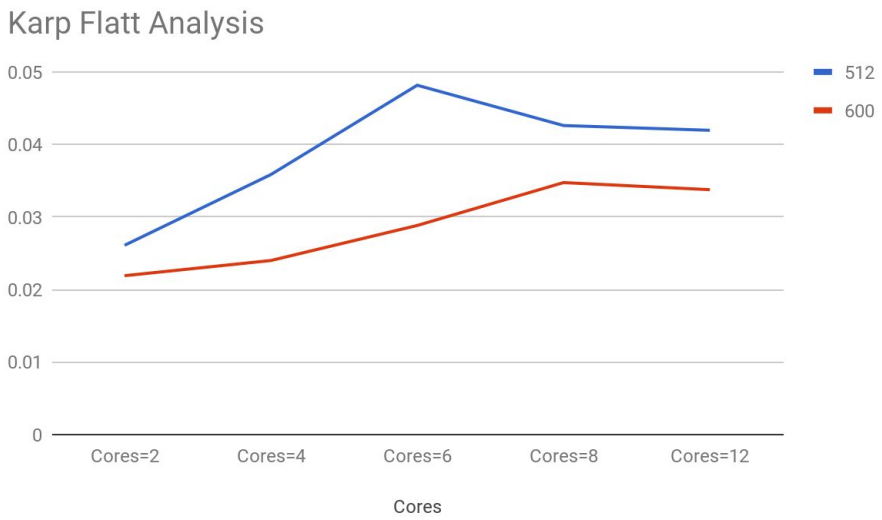

Fig.4.4.1 Karp-Flatt Analysis

- The Karp Flatt Metric for higher problem size is as shown above. Karp Flatt Metric shows that the parallelization is as per the requirement. Also we get the speedup that we expect according to the number of cores.

- Formula Used:

$$e = \frac{\frac{1}{\psi} - \frac{1}{p}}{1 - \frac{1}{p}}$$

Where,

$e = serial\ fraction$
$\psi = speedup$
$p = number\ of\ processors$

5 Conclusion and Future Scope

5.1 Conclusion

In this project we have parallelised the two dimensional heat equation for a steady state condition using Open MP API. The gprof profiler helped a lot in understanding the points (functions) where we needed to parallelize the code.

The speedup curve increases up till 6 threads and after that it flattens out. The reason for the same could be the hardware used. As seen in the hardware specifications the cluster has 2 sockets and each socket has 6 cores, as a result if we increase the number of cores beyond 6 there is some time lost in inter-socket communication(parallel-overhead time).

5.2 Future Scope

- The parallelized version has been run on a 12 core machine only. We can probably run it on larger machines having up to 100 processors for larger input sizes. This would help us know the actual scenario better because in scenarios of particle diffusion there are millions of particles and not just a few thousands.

- We can use the MPI libraries instead of the OpenMP ones for the increased number of processors.

- We have tried to implement a serial version considering the actual heat equation and all the related constants. That version could also be parallelized and better results could be found out using the stochastic equation.

5.3 References

- https://www.phas.ubc.ca/~berciu/TEACHING/PHYS312/LECTURES/FILES/2Dwave.pdf
- http://ramanujan.math.trinity.edu/rdaileda/teach/s12/m3357/lectures/lecture_3_6_short.pdf