CS-301

High Performance Computing

Solving the 2D Heat Equation to get the steady state temperature (Gauss-Seidel Algorithm)

Amarnath Karthi 201501005
Chahak Mehta 201501422

November 6, 2017

Contents

1 Hardware and Compiler specifications

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	1211.531
BogoMIPS	4804.10
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

Language	C
Compiler	gcc
Flag(s)	-fopenmp

2 Introduction

The Laplace Partial Differential Equation(PDE), also known as Laplace's Equation is a second order partial differential equation, represented as following:

$$\Delta\psi \equiv \nabla^2\psi = 0$$

Where Δ is the Laplace operator on function ψ.
In 2 Dimensions, the Laplace PDE can also be denoted as the following:

$$\frac{\delta^2\psi}{\delta x^2} + \frac{\delta^2\psi}{\delta y^2} = 0$$

The Dirichlet problem for Laplace PDE consists of finding the solution ψ to the Laplace PDE on a domain D such that it is equal to some given function at the boundary. This paper presents a parallel approach to numerically solve the 2D Dirichlet Problem using OpenMP. We have implemented this on a machine having the specifications as stated in Section 1.

2.1 Problem Statement

Given a set of boundary values, numerically find the solution to the 2 Dimensional Laplace PDE (Dirichlet problem) using the ***Gauss-Seidel Algorithm***. This is used to calculate steady state heat in 2 dimensions, and gives an output array which is visualised as shown below:

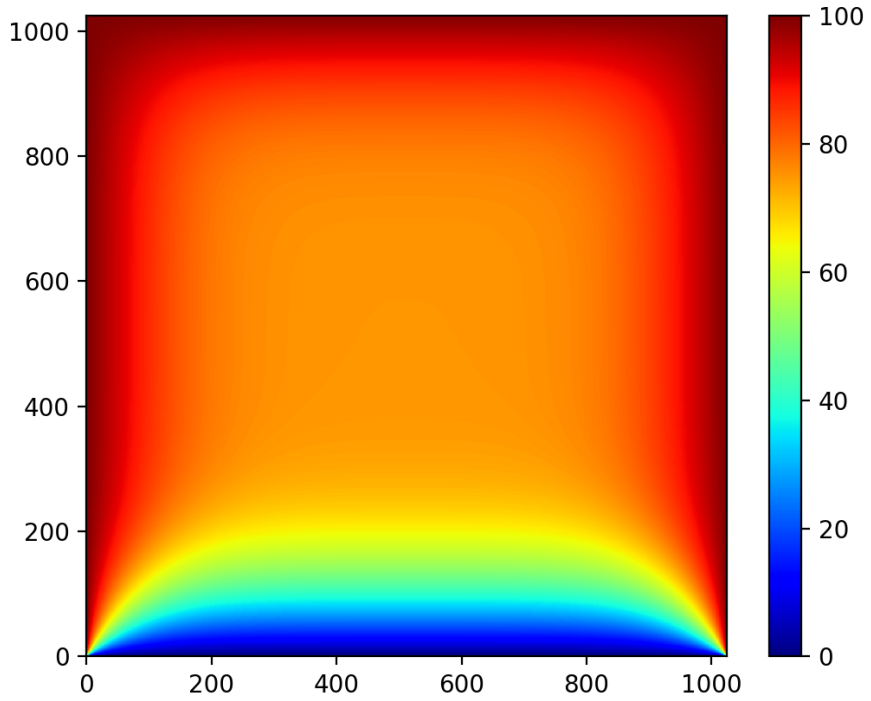


Figure 1: Steady state heat for 3 heat sources at 100°C and 1 heat sink at 0°C

2.2 Real world applications

The Laplace PDE is a rule seen in various fields of science and also in nature. In many domains like fluid dynamics, economics, electromagnetic theory this equation describes the steady state behaviour of a system. In the field of thermodynamics, this equation is also known as the steady state heat equation which gives the steady state temperature of any system given its boundary temperature.

3 Approaching to the problem solution

We can solve the 2D Dirichlet Problem numerically using the 2D finite difference method. We discretize our domain D as an mxn matrix ψ, where $\psi_{i,j}$ represents the value of ψ at (i, j). After discretization, the Laplace PDE is reduced into the following equation:

$$\psi_{i,j} = \frac{\psi_{i+1,j} + \psi_{i-1,j} + \psi_{i,j+1} + \psi_{i,j-1}}{4} \tag{1}$$

The above equation is quite intuitive in the steady state equation because one can think of the temperature ψ at a point is the average of its surrounding points in steady state.

Thus we can say that in the value of $\psi_{i,j}$ is linearly dependent on the value of ψ at other points.

3.1 Naive Gaussian Elimination

A naive way to get the solution would be to perform Gaussian Elimination on all mn discrete values of ψ. But this method is not optimal because it will have $O((mn)^2)$ space complexity and $O((mn)^3)$ time complexity. Imagine solving a 1024x1024 ψ matrix. Assuming the coefficients to be doubles of 4 Bytes each, the memory occupied by the coefficient matrix would be 4392 Gigabytes. Also it will take a lot of time to solve using Gaussian Elimination.

3.2 Gauss Seidel Method: A better approach

Notice that the coefficient matrix is sparse i.e. it has very few non zero elements. Therefore we use the Gauss Seidel method which is faster and more space efficient for sparse matrices. Basically, in this method we initialize each unknown with a random value and then in each iteration successively find one unknown in terms of the other dependent unknowns until they converge to the actual solution within some degree of accuracy.
In our case, we keep on updating the ψ matrix in-place until the maximum deviation from iteration to iteration does not come under a pre-specified error tolerance ϵ. Consider the following pseudo-code:

```
psi = discretized matrix
m   = num of rows
n   = num of columns
dev = 2*eps
def GaussSeidel(psi, m, n, epsilon):
  while dev > eps:
    dev=0
    for i=1 to m-2:
      for j=1 to n-2:
        newVal = 0.25*(psi[i+1][j]+psi[i-1][j]+psi[i][j+1]+psi[i][j-1])
        dev = max(dev, abs(newVal-psi[i][j]))
        psi[i][j] = newVal
```

Notice that the grid is filled from left to right and row-wise. Also the time complexity per iteration is $O(mn)$ and the number of iterations taken to converge depends on the value of ϵ. Also keep in mind that we do not update the boundary values.

3.3 Initializing the ψ matrix

The ψ matrix can be initialized with random values and will converge to the solution using the above pseudo-code. But as the values are random, it will almost always take a large amount of iterations to converge.

To converge faster, we can initialize the unknown values to be a mean of the boundary values. This is a good practice and saves a large amount of iterations. It works because the solution to the Laplace equation at any is confined within the minimum and maximum values at the boundary(Think of it in terms of temperature, at any point the temperature does not exceed the min and max boundary temperature, during steady state.).

4 Parallelizing Gauss-Seidel Method

The main problem faced in parallelizing this method is that of dependency. This is because the value of $\psi_{i,j}$ in an iteration is dependent on the values of $\psi_{i-1,j}$ and $\psi_{i,j-1}$, considering the serial filling order(We update the matrix row wise from left to right). This order of filling is known as the **Natural Order**. Under natural order of filling, this method cannot be parallelized.
Even though there is a dependency, notice that it is a weak one. To ensure that concurrency is maintained, two data decomposition schemes can be utilised, namely the **wavefront decomposition scheme** and the **red-black decomposition scheme**.

4.1 Wavefront decomposition scheme

To compute $\psi_{i,j}$ we need to compute the values of $\psi_{i-1,j}$ and $\psi_{i,j-1}$. Thus we say that $\psi_{i-1,j}$ and $\psi_{i,j-1}$ are the predecessors of $\psi_{i,j}$. The first wavefront is the set of all unknowns which have no predecessor. Each successive wavefront is the set of successors of the current wavefront. Elements in a given wavefront are independent of each other and thus for a given wavefront its elements can be calculated parallely. Also a wavefront can be calculated only after its previous wavefront is calculated.

For mxn matrix, we have $m + n - 5$ wavefronts. Also, for a given wavefront $i + j$ is always a constant and unique. Basically we are doing a diagonal traversal of the matrix.

Consider the following pseudo-code for wavefront decomposition:

```
currR, currC = 1, 1
nextR, nextC = 1, 1
dev = 2*eps
def GaussSeidel_Wavefront(psi, m, n, epsilon):
  while dev > eps:
    dev=0
    for wavefront_num=1 to m+n-5
      if currR+1<n-1:
        nextR=currR+1
      else
        nextC=currC+1
      i, j = currR, currC

      #begin parallel region
      parallel for j=1 to currC
        newVal = 0.25*(psi[i+1][j]+psi[i-1][j]+psi[i][j+1]+psi[i][j-1])
        dev = max(dev, abs(newVal-psi[i][j]))
        psi[i][j] = newVal
        i++
        j--
```

4.2 Red-Black decomposition scheme

In this scheme, we divide all the nodes into 2 sets, the red set and the black set. A red node has no immediate red neighbor and a black node has no immediate black neighbour.(Think of this division scheme as a red and black chessboard). Therefore all red nodes are independent of each other. The same holds true for black nodes. Therefore we can parallely update all red nodes (red sweep) and then parallely update all black nodes(black sweep).

Optimization: Modern processors have large pipelines to support instruction level parallelism, but this also increases **branch misprediction penalty**. Branch prediction/misprediction happens when the program uses lots of if-else statements. Therefore to decrease penalties, the color of the noe is not identified by if-else statements. Instead, "clever" for loop initializations combined with modulo 2 arithmetic allows us to avoid if-else statements entirely.

Consider the following pseudo-code:

```
psi = discretized matrix
m   = num of rows
n   = num of columns
dev = 2*eps

def GaussSeidel_RedBlack(psi, m, n, epsilon):
  while dev > eps:
    dev=0
    #begin parallel region for red sweep
    parallel for i=1 to m-2:
      for j=i%2 to n-2, j+=2:
          newVal = 0.25*(psi[i+1][j]+psi[i-1][j]+psi[i][j+1]+psi[i][j-1])
          dev = max(dev, abs(newVal-psi[i][j]))
          psi[i][j] = newVal
    #barrier

    #begin parallel region for red sweep
    for j=(i+1)%2 to n-2, j+=2:
          newVal = 0.25*(psi[i+1][j]+psi[i-1][j]+psi[i][j+1]+psi[i][j-1])
          dev = max(dev, abs(newVal-psi[i][j]))
          psi[i][j] = newVal
    #barrier
```

4.3 Work Span Analysis: Task dependency graph

If each node is seen as a task, then we can see a task dependency graph. Both the algorithms have different task dependency graphs.

For a given iteration, both decomposition methods do $O(mn)$ work, becaus each node is computed only once. But there is a huge amount of difference in span. In the wavefront decomposition, the span is equal to the number of wavefronts, and therefore the span is $O(m+n)$, whereas, in red-black decomposition the span is only 2, since the red nodes depend on black and vice versa. Therefore red-black decomposition has $O(1)$ span.

Therefore, asymptotically, the red-black method is better than the wavefront decomposition because given infinite processors, the time complexity per iteration will be lesser in the case of red-black method.

4.4 Cache Analysis

Another reason for the red-black decomposition to be better is that it uses cache because it preserves locality during memory access(row wise memory access). Whereas, in the case of wavefront decomposition, we are accessing elements diagonally, meaning that the cache locality is not preserved.

4.5 Theoretical Speedup Analysis

For simplicity, its assumed that the matrix is square($m = n$). Also it is assumed that the code has no parallel overheads and the machine has infinitely large memory bandwidth.

Considering the above scenario, red-black decomposition will give a theoretical speedup of p for small number (p) of processors and a speedup of $\frac{n^2}{2}$ for asymptotically large number of processors because out of these, only $\frac{n^2}{2}$ will be used at max because there are these many red nodes and black nodes.

Wavefront decomposition will give us a speedup of p for a small number of processors, because for a large matrix, almost each wavefront will be distributed amongst all processors. For a large number of processors, each wavefront will have speedup equal to the number of elements in

it. So adding all these and dividing by n will give us the average speedup of the decomposition which will be about n.
Due to the above reasons, Red Black Decomposition is a better method than wavefront decomposition.

5 Serial Results

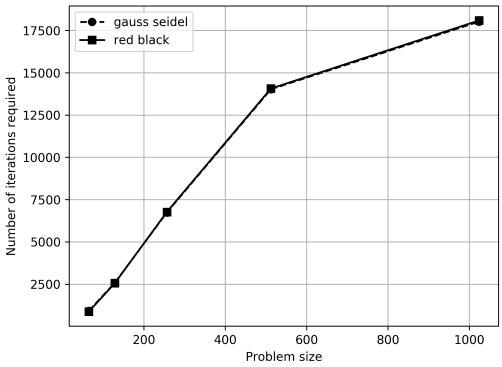
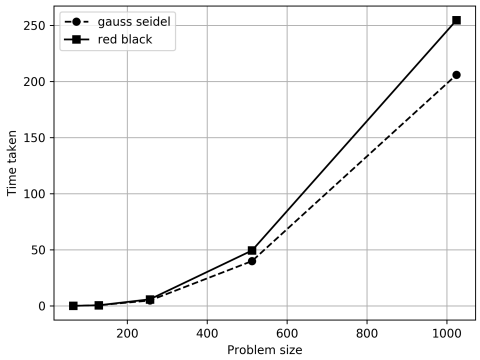


Figure 2: Number of iterations vs Problem size

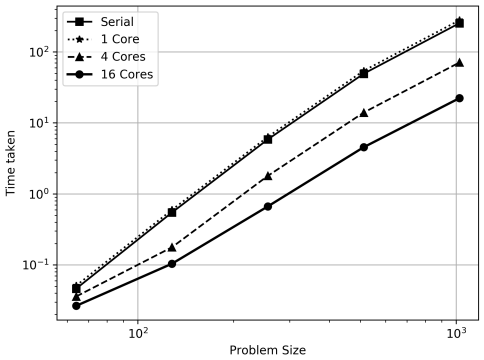
The above figure shows number of iterations required by naive Gauss Seidel and the red black Gauss Seidel method. Irrespective of the problem size, iterations are roughly the same. Therefore, asymptotically both the serial algorithms perform in relatively similar time bounds. However in practice, the naive Gauss Seidel method is slightly faster because it uses the cache line more effectively than its red black counterpart. It uses cache line in one sweep whereas the red black does the same in 2 sweeps.

6 Parallel Results

6.1 Time Curve analysis:



(a) Time taken vs Problem size (Serial)



(b) Time taken vs Problem size (Parallel)

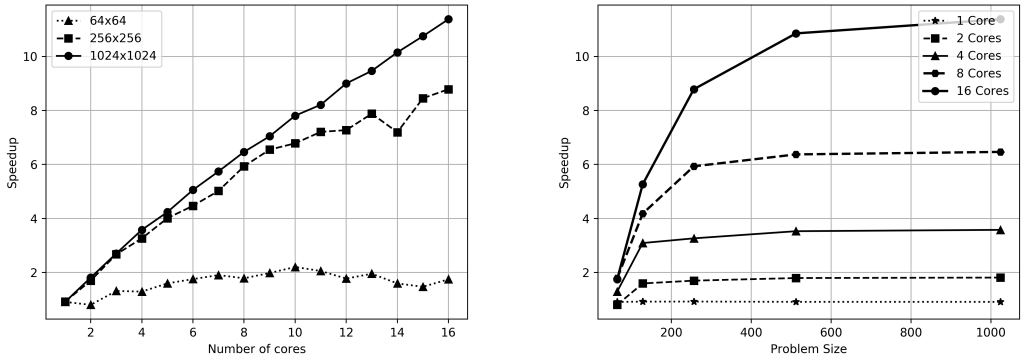
Figure 3: Time Curves for Serial and Parallel algorithms

This section shows the results of parallelization of the Gauss Seidel method using the red-black decomposition. Also, a comparison between the 2 serial variants is shown in the first figure(efficient cache line usage makes one faster).

As we increase the problem size by a factor of 2, we expect the execution time to also increase more or less proportionately. Off course this is a gross estimation which does not take into account the caching in the architecture. Also as we increase the number of cores, for a given input size, the execution time should decrease (for an optimal parallel algorithm). Shown above is the actual curve we get from our experiments. Note that it is plotted in log-log scale considering the huge ranges of data.We see that this more or less adheres to our observations.

The parallel implementation on 1 thread is slower always. This is because parallelizing the code with one thread adds only parallel overheads with no benefits. Time increases as we increase the number of cores. For small input sizes, the actual serial computation time is very low. Parallelizing the code only adds some more overheads, which dominate for small input sizes. The time taken for synchronization, and load distribution is greater than the time saved by parallelizing the code. Whereas, this is not the case for large input sizes, because here although the compute time increases significantly, the parallel overhead remains more or less the same. This can be seen in the graph, as we increase the number of processors, for large inputs, the curve is a line, and shifts downwards, indicating a reduction in execution time by a constant factor throughout.

6.2 Speedup Curve Analysis:



(a) Speedup vs Number of cores

(b) Speedup vs Problem size

Figure 4: Speedup Curves for *Number of cores* and *Problem size*

Shown above are the speedup curves in relation to the number of cores and the problem size. Notice that as problem size increases, the speedup first increases sharply, and then remains more or less constant.

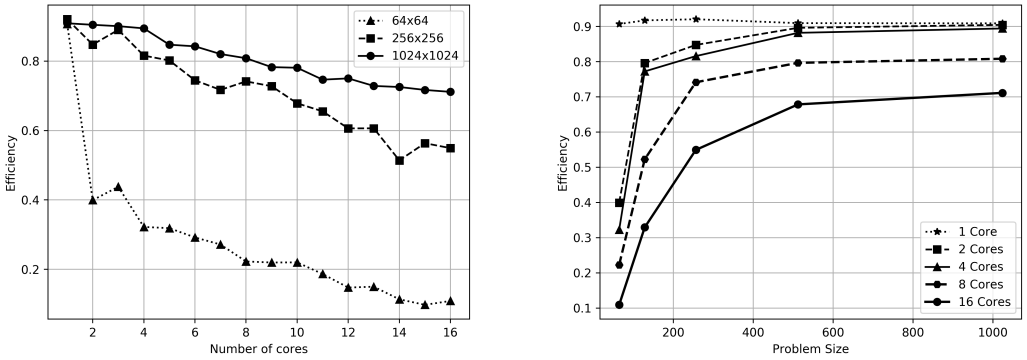
Initially as the input size is small, the parallel overhead dominates over the time saved by parallelizing the code. As the problem size increases, the compute time becomes larger , whereas the parallel overhead remains more or less the same. Hence more compute time is saved by parallelizing the code thereby giving us greater and greater speedup.

When the problem size is small increasing the number of cores just adds more parallel overhead because of increased inter-processor synchronization and communication. This is not compensated by the time saved in parallelization since the compute time is already very low in case of the serial code.
For large problem sizes, the case is opposite. Adding a new processor increases the speedup. This is because the compute time is inversely proportional to the number of processors. For large input sizes, the compute time is very high. So even though the parallel overhead increases slightly, a huge amount of time is saved by adding one extra processor.

For small number of threads (e.g. 2), the performance scales nearly linearly (A slight variation can be attributed to load on the host machine), however, for large number of threads (e.g.

8), the performance does not scale linearly. This is due to the fact that the red black decomposition algorithm involves multiple iterations and phases; and the synchronization required after each phase creates serialization bottleneck due to which multiple threads do not progress independently. If the number of iterations required for convergence is L, then synchronization is required for $2L$ times. Moreover, with the increasing number of cores, although the processing power increases, the other resources such as cache, memory bandwidth etc. do not increase linearly and hence, the program performance does not scale linearly.

6.3 Efficiency Curve Analysis



(a) Efficiency vs Number of cores

(b) Efficiency vs Problem size

Figure 5: Efficiency Curves for *Number of cores* and *Problem size*

Efficiency is defined as the speedup contributed per unit core. Essentially it is the ratio of speedup and the number of cores. It is a measure of the scalability of a parallel algorithm.

$$e = \frac{s_p}{p}$$

Ideally, given infinite resources, the efficiency of a parallel code would be 1 since the code speeds up by a factor of p (number of cores) theoretically but practically this is not the case since we don't have infinite resources and hence the efficiency decreases with increase in number of cores.

Furthermore, for smaller problem sizes and increasing number of cores, the parallel algorithm is not efficient because the number of computation operations is very less and it doesn't compensate for the parallel overhead that occurs because of the communication and synchronization between the threads. But on increasing the problem size, the efficiency for doesn't decrease at the same rate with increase in number of cores because the parallel overhead starts getting compensated with the increasing number of computations.

In Figure 5b, the speedup starts saturating with increasing problem size because after a point, if the number of processors become more than the problem size, then the parallel overhead becomes more than the computation time and hence speedup decreases.

6.4 Serial fraction and the Karp-Flatt metric:

Serial fraction is the ratio of the code which is inherently serial and can never be parallelized. The Karp-Flatt Metric is a method used to measure the serial fraction. This metric gives us the following relation for the serial fraction e:

$$e = \frac{\frac{1}{s(p)} - \frac{1}{p}}{1 - \frac{1}{p}} \tag{2}$$

Here $s(p)$ denotes the speedup s given by p processors.

If on increasing the number of processors, e converges, then there is no more scope of parallelism and adding more processors will not affect the performance.
Whereas, if on increasing the number of processors e keeps on decreasing, it indicates that there is scope for more parallelism and to decrease the apparent serial fraction, thereby indicating that further more processors can be added to boost the performance.

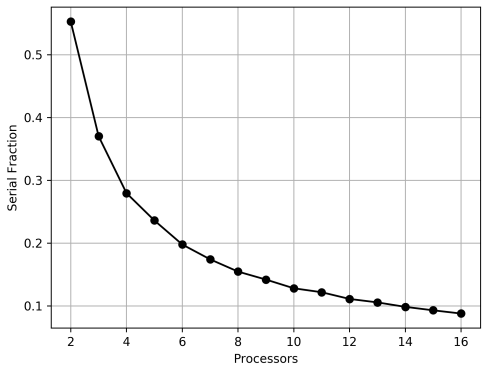


Figure 6: The serial fraction for problem size 1024x1024 vs the number of cores

7 Conclusion

7.1 Disadvantages of Red Black Decomposition

Though this method enables parallelism, it is cache inefficient as it travels through the memory in 2 phases (red sweep and black sweep). Each element is brought into the cache twice.

7.2 Future scope

This method can be extended to solve multidimensional Laplace PDE to get steady state solutions. Also, this method with slight modifications can probably be used to solve various other families of PDE like the Poisson PDE. For code and further development refer to the repository at https://github.com/97amarnathk/steadyStateHeat

8 References

1. Parallel SOR Iterative Algorithms and Performance Evaluation on a Linux Cluster: http://www.dtic.mil/dtic/tr/fulltext/u2/a449212.pdf

2. Numerical Solution of Laplace Equation, Gilberto E. Urroz, October 2004 http://ocw.usu.edu/Civil_and_Environmental_Engineering/ Numerical_Methods_in_Civil_Engineering/LaplaceDirichletTemperature.pdf