# Finding 2D Steady State Heat Distribution with Parallel Programming

**San Kho Lin (829463), Guang Hu (777430)**
(sanl1, ghu1)@student.unimelb.edu.au

## 1 Introduction

In this project, we experiment the parallelization of the program that finds the Heat Distribution over an isotropic two-dimensional square plate using MPI and OpenMP parallel programming environment.

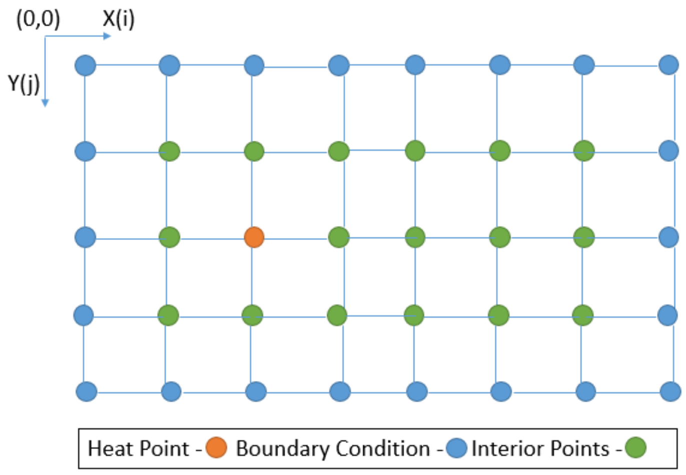
## 2 Heat Flow Problem

To begin with, we first study the general case of heat flow problem and its deriving equations. The **heat equation** is an important **partial differential equation (PDE)** that describes the distribution of heat (or variation in temperature) in a given region over time[1]. Though there are stages for conduction and convection of heat transfer[3], in this project we only focus on the two dimensional steady **Dirichlet Problem** (i.e the region is rectangular/square). This 2D heat transfer problem is governed by **Laplace's equation**[2] which is a second-order PDE:

$$\frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} = 0$$

In mathematics, **finite-difference methods (FDM)**[3] are dominant approach to numerical solutions for solving PDE by approximating them with *difference equations*. In this case, the continuous variables are discretized into a large number of points on the domain. A system of algebraic equations can be formulated by doing the approximate difference substitution for a large number of points in the domain. The domain is evenly divided into **a square mesh** where the points are evenly spaced $h$ units apart in both directions: $\Delta x = \Delta y = h$. The solution at each point is computed in an iterative manner until the convergence reach to zero or an acceptable tolerance. It is then said to approximate the temperature distribution.

Figure 1: Domain Decomposition of Heat Distribution

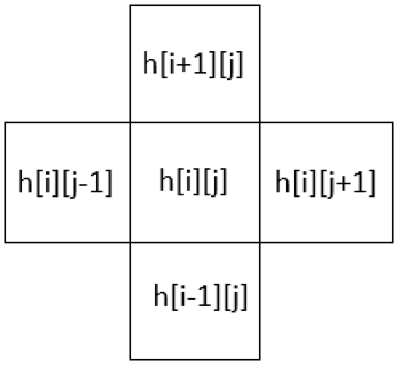


[1]https://en.wikipedia.org/wiki/Heat_equation
[2]https://en.wikipedia.org/wiki/Laplace's_equation
[3]https://en.wikipedia.org/wiki/Finite_difference_method

## 3 Deriving the Sequential Program using FDM

As discuss in previous section, by using FDM, the solution space is discretized into a large number of points. Then we need to consider for how to calculate the temperature among these points. In our experiment, we consider the approach discuss in the text book Wilkinson and Allen[4] where the author states that the temperature at an inside point can be taken to the average of the temperature of the four neighbouring points. We consider the initial boundary conditions are to all zero and

Figure 2: Heat Point Depends on Average of Neighbouring Points

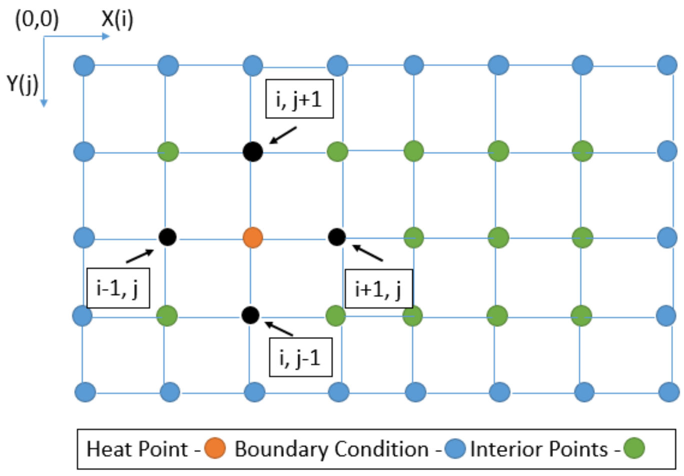


then the interior points of $h_{i,j}$ are where $0 < i < n, 0 < j < n$ for $(n-1) \times (n-1)$ interior points. The boundary points are when $i = 0, i = n, j = 0, or j = n$, and have fixed values corresponding to the fixed temperature of the edges. And the $n$ is problem domain size. We can then compute the temperature of each interior point by iterating the equation:

$$h_{i,j} = \frac{h_{(i-1,j)}+h_{(i+1,j)}+h_{(i,j-1)}+h_{(i,j+1)}}{4}$$

$(0 < i < n, 0 < j < n)$ for a fixed number of iterations or until the difference between iterations of a point is less than some very small precision value, e.g. Epsilon that require for the problem accuracy.

Figure 3: Computing an interior Point based on Average of Neighbour



### 3.1 Jacobi Iteration

Since the FDM solution rely on using the iteration to solve the Laplace's equation, we continue learn that the Jacobi method of iteration technique can be incorporated into FDM for solving the heat flow problem. The general idea of Jacobi iteration is such that given a general system of $n$ linear equations,

$$Ax = b$$

then the Jacobi iteration is described by:

$$x_i^k = \frac{1}{a_{i,i}}[b_i - \Sigma_{j \neq i} a_{i,j} x_j^{k-1}]$$

where the superscript $k$ indicates the $k$-th iteration[1]. By using a fixed number of iteration, the calculation as sequential code would be

```
for (n = 0; n < MAXITER; n++)
  for (i = 1; i < SIZE; i++)
    for (j = 1; j < SIZE; j++)
      matrix[i][j] = 0.25 * (previous[i-1][j] + previous[i+1][j] + previous[i][j-1]
              + previous[i][j+1]);
  for (i = 1; i < SIZE; i++)
    for (j = 1; j < SIZE; j++)
      previous[i][j] = matrix[i][j];
```

In our experiment, we only approach to the fixed number of iteration to simplify our assumption for later part of parallelizing the program (i.e. the consideration for the termination condition in parallel processes). We also learn that there are several ways to include the covergence algorithms to determine where $matrix[i][j]$ has converged to the required precision. For instance, one heuristic approach suggested in Quinn[2] book would be using some Epsilon value in such that

```
#include <math.h>
#define EPSILON 0.01
...
for (;;)
  diff = 0.0;
  for (i = 1; i < SIZE; i++)
    for (j = 1; j < SIZE; j++)
      matrix[i][j] = 0.25 * (previous[i-1][j] + previous[i+1][j] + previous[i][j-1]
              + previous[i][j+1]);
      if (fabs(matrix[i][j] - previous[i][j]) > diff)
        diff = fabs(matrix[i][j] - previous[i][j]);
  if (diff <= EPSILON) break;
  for (i = 1; i < SIZE; i++)
    for (j = 1; j < SIZE; j++)
      previous[i][j] = matrix[i][j];
```

In the nutshell, using the values of one iteration to compute the values for the next iteration in such way is the significant program style of the Jacobi iteration. It relies strictly on values from the previous iteration. This highlights on the potential parallelization in such that each point could theoretically be handled on its own process and computing the entire domain in one step. This also means that the maximum number of parallel computation for each point is equal to the number of available processors.

### 3.2 Gauss-Seidel Relaxation

In the above Jacobi iteration code, we observe that given the sequential order of the computation, two of the values ($previous[i-1][j]$ and $previous[i][j-1]$) used to compute $matrix[i][j]$ have already been computed in the same iteration. We continue learn that this can be used to avoid the step of copying values to $previous[i][j]$ matrix by using Gauss-Seidel relaxation.

In general, the Gauss-Seidel relaxation attempts to increase the convergence rate by using values computed for the $k$-th iteration in subsequent computations with the $k$-th iteration[1]. The general iteration formula is

$$x_i^k = \frac{1}{a_{i,i}}[b_i - \Sigma_{j=1}^{i-1} a_{i,j}x_j^k - \Sigma_{j=i+1}^{n} a_{i,j}x_j^{k-1}]$$

which assume the computation of $x_i$ in natural order i.e. sequentially. In this case, we can re-write our sequential code into

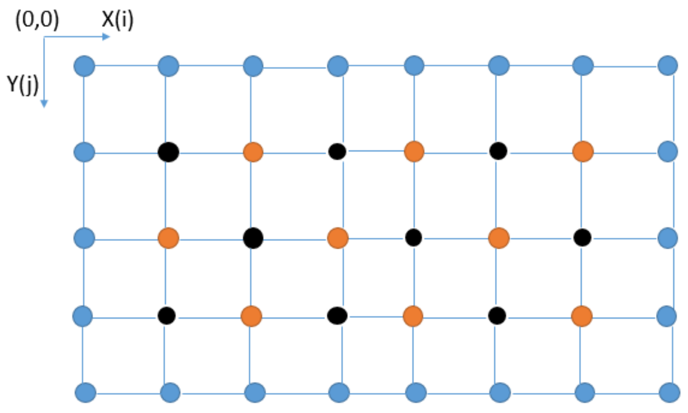
```
for (n = 0; n < MAXITER; n++)
  for (i = 1; i < SIZE; i++)
    for (j = 1; j < SIZE; j++)
      matrix[i][j] = 0.25 * (matrix[i-1][j] + matrix[i+1][j] + matrix[i][j-1]
              + matrix[i][j+1]);
```

In this way, the program is using the most recent values available and it produces a faster convergence but relies upon the sequential order of the computation.

### 3.3 Red-Black Ordering

Another approach that we consider in our experiment is Red-Black Ordering on interior points distribution. This methods also inherit the assumption of the computation order in sequential like Gauss-Seidel relaxation. Additionally, at each point (i,j) in the domain is denoted a red or a black according to whether i+j is odd or even. Then the red point calculations use values from the previous iteration and the black point calculations use values from the current iteration of the red points.

Figure 4: Red/Black Interior Nodal Distribution



## 4 Parallel Implementation

In our Parallel approach, we experiment using OpenMP on the basic Jacobi iteration method and Red-Black Ordering method, and using MPI for Gauss-Seidel relaxation method. We observe the following considerations for parallalization.

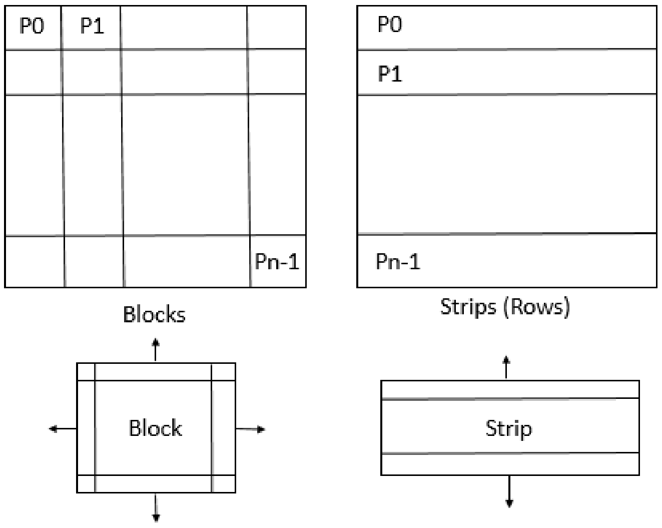
## 4.1 Consideration

**Eventual Result** - From sequential code, we observe all the points can be visited simultaneously without any change to the algorithm because when the iteration become large and convergence enough, the computed result will be the same.

**Single Program Multiple Data** - It is a single program that manages data distribution and number of processes is executed.

**Partitioning** - For partitioning, we would ideally allocate more than one point to each process as there would be always more points than available processors. Therefore, the mesh of the domain could be partitioned into blocks or strips(rows).

Figure 5: Partitioning Heat Distribution and Communication consequences of partitioning



As shows in Figure 5, the block partition cause the four edges to exchange the data points. Each process would send and receive four messages in each iteration. The communication cost for the block partition is

$$t_{commblock} = 8(t_{startup} + \frac{n}{\sqrt{p}} t_{data})$$

whereas the equation is valid for $p \geq 9$ when at least one block has four communication edge. For Row strip partition, there are two edges where data points are exchanged and the communication time is as the following equation. In nutshell, the communication time will be cirtically influenced by the startup time[4].

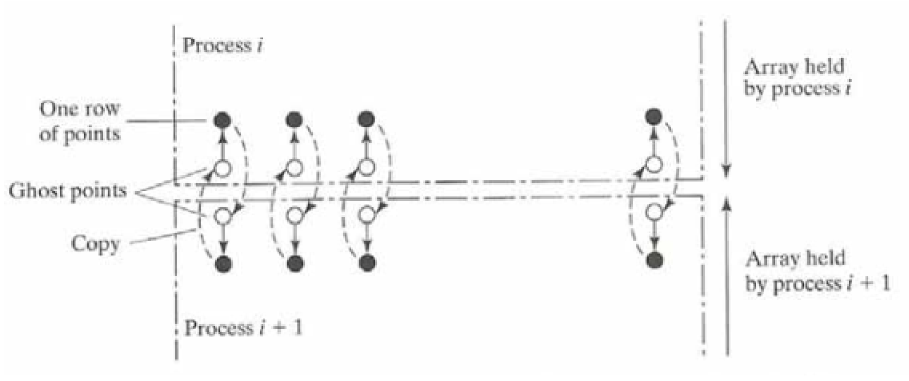
$$t_{commstrip} = 4(t_{startup} + nt_{data})$$

**Row Distribution** - In our experiment, we attempt only Row wise decomposition. We observe that Rows are distributed as evenly as possible to each processor for proper load balancing and minimize the idle processes.

**Ghost Row** - In order to minimize the communication and for the programming convenience, the Ghost Rows (see Figure 6) are used. These rows are immediately above and below of the processing row. Rows are communicated between adjacent process at the starting of each iteration. To avoid safety and deadlock, sends/receives are non-blocking using `MPI_Isend/MPI_Irecv`.

**No Global Matrices** - There is no global matrix at any point during the solving of the iterative heat equation. It processes in such that each process knows its assigned rows and at most 2 more one-dimensional ”ghost rows” of a size corresponding to the number of points in the x-direction.

Figure 6: Configurating array into contiguous rows for each process, with ghost points[4]



### 4.2 Implementation Details

For OpenMP implementation, it is quite straight forward. We parallelize the rows in each process and compute the result accordingly. For MPI implementation, instead of using 2D array, we use contiguous 1D array to construct the points. Then we use `MPI_Scatterv/MPI_Gatherv` to let each processes acquire their respective chunk size for computation. Since there might be uneven combination for number of points over number of processors, using `MPI_Scatterv`, we divide the task by calculating the displacement and assignment to each process as even as possible.

```
MPI_Scatterv(matrix, chunksize, chunkid, MPI_DOUBLE, localmatrix,
  chunksize[myid], MPI_DOUBLE, 0, MPI_COMM_WORLD)
```

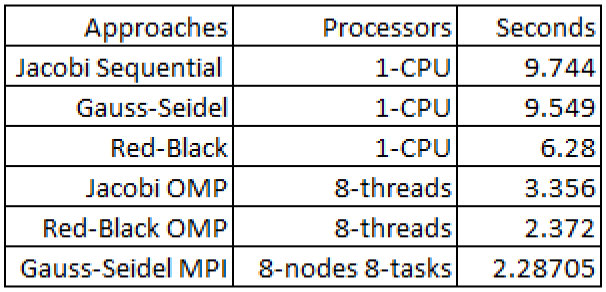
For ghost rows, we use non-blocking sends and receives MPI routines. We then use `MPI_Wait` for local synchronization in each processes to communicate with another processes.

```
MPI_Isend(localmatrix, size, MPI_DOUBLE, myid, 0, MPI_COMM_WORLD, &request)
MPI_Irecv(before, size, MPI_DOUBLE, myid, 0, MPI_COMM_WORLD, &request)
MPI_Wait(&request, &status)
```

## 5 Conclusions

For run time analysis, we choose $1000 \times 1000$ grid points with maximum iteration to 1000 for the heat point starting at $(500, 500)$. In Figure 7 compare the run time between sequential version of 3 different methods. We also observe that it could further improve with using partition like block decomposition to achieve better run time.

Figure 7: Run Time Comparison



| Approaches | Processors | Seconds |
|---:|---:|---:|
| Jacobi Sequential | 1-CPU | 9.744 |
| Gauss-Seidel | 1-CPU | 9.549 |
| Red-Black | 1-CPU | 6.28 |
| Jacobi OMP | 8-threads | 3.356 |
| Red-Black OMP | 8-threads | 2.372 |
| Gauss-Seidel MPI | 8-nodes 8-tasks | 2.28705 |

In summary, we observe that it is particularly more challenging to develop program on MPI environment than OpenMP. Especially, for a problem like Heat Distribution which requires local synchronization as discussed in ”Ghost Rows”.

## 6 References

[1] Aaron Harwood. *COMP90025 Parallel and Multicore Computing.* Lecture series — Introduction, Architectures, Classes of parallel algorithms. The University of Melbourne, 2016.

[2] Quinn Michael J. *Parallel Programming in C with MPI and OpenMP.* International series of monographs on physics. McGraw Hill, 2003. ISBN: 007-282256-2.

[3] Michael C. Wendl. *Theoretical Foundations of Conduction and Convection Heat Transfer.* International series of monographs on physics. DOI 10.13140/RG.2.1.1875.3120. Wendl Foundation, 2012. URL: `http://wendl.weebly.com/textbook.html` (visited on 10/21/2016).

[4] Barry Wilkinson and Michael Allen. *Parallel Programming, Techniques and Applications Using Networked Workstations and Parallel Computers 2nd Edition.* Prentice Hall, 2005. ISBN: 0-13-140563-2.