AMS 530 Principles of Parallel Computing
Project 4 Parallel 2D Poisson Equation

Wenhan Gao
Department of Applied Mathematics and Statistics, Stony Brook University, Stony Brook, NY 11794, USA
`wenhan.gao@stonybrook.edu`

1 Problem Description

Physical laws often take the form of partial differential equations (PDEs), e.g., the Hamilton-Jacobi-Bellman (HJB) equation [1] in control theory and the Schrődinger equation [2] in quantum mechanics.

There are various traditional numerical methods for solving PDEs; for example, the Finite Difference Method [3]. However, when the grid size is large, it can be computationally intensive. Therefore, it is imperative to parallelize the algorithm to distribute the work to multiple processors.

Consider a 2D Poisson's equation with Dirichlet boundary conditions:

$$\begin{aligned} \Delta u(x,y) &= f(x,y), && \text{in } \Omega := (0,1)\times(0,1), \\ u(x,y) &= g(x,y), && \text{on } \partial\Omega, \end{aligned}$$

where Δ is the Laplacian operator, $f(x) = \sin(x-y)$, and $g(x,y) = 1$.

In this report, the PDE above will be solved by FDM in parallel on three different grids, 100×100, 200×200, and 400×400, using $P = 2, 10, 20$ processors.

2 Algorithm Description and Pseudo-code

2.1 Sequential FDM: Five-point Stencil

The unit square domain can be discretized as an $N \times N$ matrix:

$$x_i = i \cdot h,\ y_j = j \cdot h,\ i,j \in [0, N],$$

where the step size $h = dx = dy = \frac{1}{N}$, and

$$u_{i,j} = u\left(x_i, y_j\right),\ f_{i,j} = f\left(x_i, y_j\right) = sin(x_i - y_j).$$

By central difference approximation:

$$\frac{u_{i+1,j} + u_{i-1,j} - 2u_{i,j}}{h^2} + \frac{u_{i,j+1} + u_{i,j-1} - 2u_{i,j}}{h^2} = \sin\left(x_i - y_j\right).$$

For all unknown $u_{i,j}, i,j \in [1, N-1]$, we have a system of $(N-2)^2$ equations with $(N-2)^2$ unknown variables, which can be written in the form of $Au = b$. An example is shown in Figure 2.1.

Since this system of linear equations is sparse and strictly diagonally dominant, Jacobi method [4] can be applied to solve this system of equations. The updates for interior grid points ($u_{i,j}, i,j \in [1, N-1]$) are:

$$u_{i,j}^{n+1} = \frac{1}{4}(u_{i+1,j}^n + u_{i-1,j}^n + u_{i,j+1}^n + u_{i,j-1}^n - f_{i,j} \cdot h^2). \tag{2.1}$$

For the update of an interior grid point, we need the values from its 4 neighbours; The point itself and its 4 neighbouring points are called the Five-point Stencil (Figure 2.2). The algorithm starts with an initial guess solution, and the Jacobi updates are applied iteratively until interior gridvalues converge. In this work, we set the initial guess to be a zero matrix and set the stopping criteria to be: $\max(|(u^{n+1} - u^n|) \leq 10^{-8}$.

Figure 2.1: $Au = b$ Example; Image From Dr. Deng's AMS 530 Lecture Note

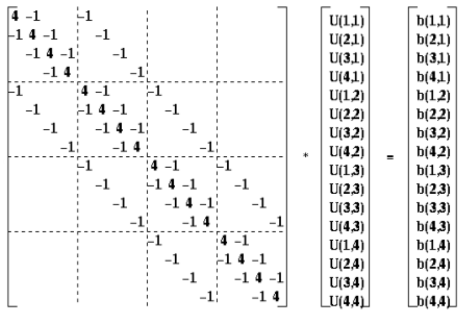
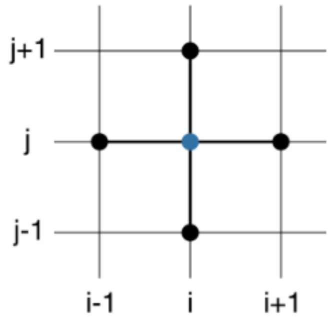


Figure 2.2: Five-point Stencil



2.2 Parallelization

Since unknown $u_{i,j}$s' updates are independent of each other, the Jacobi method can be parallelized very easily by its nature. In this report, we are going to parallelize the Jacobi method using so called "domain decomposition": the domain is decomposed into P "stripes". We assign each strip to one processor and have the Jacobi updates for interior grid points done on that processor locally. Notice that the values of $u_{i,j}$ on the boundary where the original grid is divided are updated with information from the neighboring stripe. Thus, it requires communication between neighboring processors to complete the Jacobi updates. This process is demonstrated pictorially in Figure 2.3. Now, in each iteration, the parallel Jacobi method will first perform information exchange and then perform local Jacobi updates. In the end, we gather all local stripes to get the numerical solution to the PDE. The pseudo-code is in Algorithm 2.1.

2.3 Main Structure of the Program

The program contains two python scripts. One named main.py is the main program and the other named utilities.py contains helper functions such as information exchange and Jacobi iteration. The main.py script is also in charge of recording running time. We take the maximum running time across all processors as the global elapsed running time. Another script named sequential.py is used to record the running time for the sequential case, i.e., $P = 1$, in order to make the speed-up curves. The main program will also produce a plot of the approximate solution to verify the correctness of the program, and the plot (.png) is saved to the current directory.

Note that main.py only works when $P \geq 2$, if there is only one processor, one should run sequential.py.

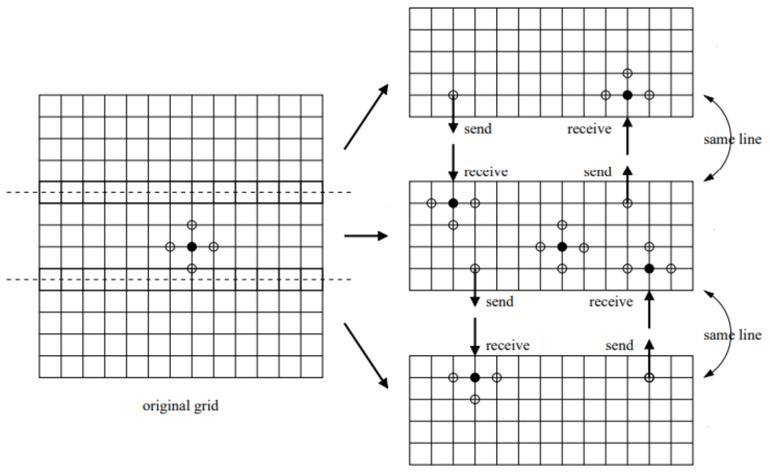
2.3.1 Slurm Configuration

To run the program on Seawulf with interactive shells, we load a distribution of Python 3.0 managed through Anaconda and also the mpi4py module for parallel implementations. Upload both python scripts and run the following commands to run the program:

module load slurm

srun -N 1 -n 4 -p short-28core –pty bash (here we request 4 processors with -n 4, one can change this number)

Figure 2.3: Domain Decomposition by Stripes; Image Adapted From [5]



Algorithm 2.1: Parallel FDM

```
Result: $u \in \mathbb{R}^{N \times N}$
Require: $N$ (grid size); size; rank; $L = 1$ (length of the unit square domain); $\epsilon = 10^{-8}$ (stopping threshold);
         $h = \frac{1}{N}$ (finite different step size); $f \in \mathbb{R}^{N \times N}$ (2d array containing grid values of $f$);
1  Initialize $u \in \mathbb{R}^{N \times N}$ with 0-s;
2  Scatter $u$ and $f$ to all processors as sub_$u$ and sub_$f$;
3  Loop
4      For all processor, except processor 0, send the first row of sub_$u$ (boundary information) to its preceding
         processor;
5      For all processor, except the last processor, send the last row of sub_$u$ (boundary information) to its succeeding
         processor;
6      For all processor, except processor 0, receive boundary information from its preceding processor;
7      For all processor, except the last processor, receive boundary information from its succeeding processor;
8      For all processor, perform local Jacobi updates for all local interior points as described in Equation 2.1;
9      diff $= \max(|(\text{sub\_}u^{n+1} - \text{sub\_}u^{n}|)$;
10     $\delta =$ allreduce($\delta$, MPI_MAX) ;  ▷ Now, $\delta = \max(|(u^{n+1} - u^{n}|)$ is the max in changes of all $u_{i,j}$
11     if $\delta \leq \epsilon$ then
12         break ;                                   ▷ stopping criteria met, break the loop
13     end
14 EndLoop
15 Gather all strips of $u$, sub_$u$-s, to the root processor;
```

```
module load anaconda/3

module load mpi4py/3.0.3

mpirun python main.py
```

3 Results and Analysis of Program Performance

First of all, I want to verify that the numerical solution generated by our program is correct. As far as I know, this PDE cannot be solved analytically by any known methods, so I used Wolfram Mathematica to solve this PDE and use its numerical approximation as the “true" solution. The Mathematica output is shown in Figure 3.2. The solutions given in this work are quite similar under $N = 100, 200, 400$, and the solutions are exactly the same with different numbers of processors. Therefore, we present the solution when $N = 100$ in Figure 3.3. In Figure 3.4, the convergence of the Jacobi iteration under $N = 100$ is shown. By comparing the Wolfram solution and my solution, it is obvious that my program is giving the correct approximation to the PDE solution.

Next, the timing and speed-up results with respect to different numbers of processors and different numbers of grid sizes are presented in Table 1 and 2, respectively. Also, the speed-up curves are presented in Figure 3.1. The speed-up factor is defined as $S(P,N) = \frac{T(1,N)}{T(P,N)}$, where $T(P,N)$ is the running time with P processors.

Overall, the program is robust and efficient. The speed-up with respect to the number of processors becomes more remarkable when the grid size increases, it is reasonable to expect the speed-up factor gets even better with the increase of the grid size; asymptotically, it is even possible to observe a linear speed-up. From the speed up results, we observed that:

- The speed-up gets better when the scale gets larger; this happens because the communication cost becomes less significant compared to the computation cost, which is distributed to multiple processors.
- In some cases, the speed-up is actually worse with more processors; for example, the case when $P = 20, N = 100$ is worse than $P = 2, N = 100$ and $P = 10, N = 100$. This happens because the computation task itself is already simple enough with only a few processors, and when more processors are used for this task, it takes additional time for communication such as creating the communication pipeline, etc..

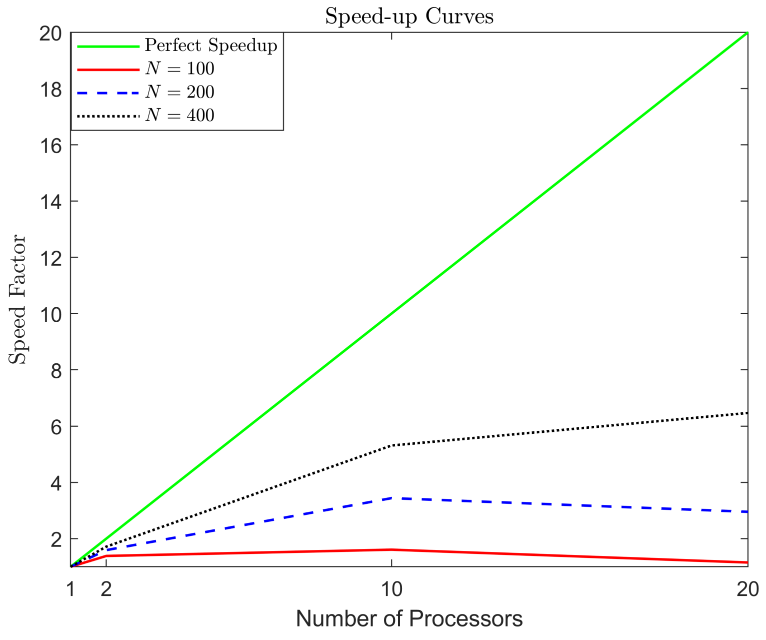
	$P = 1$	$P = 2$	$P = 10$	$P = 20$
$N = 100$	2.679693	1.934774	1.666728	2.326345
$N = 200$	28.899304	18.155115	8.402078	9.783527
$N = 400$	354.117078	206.414760	66.682876	54.755312

Table 1: Running Time in Seconds

	$P = 2$	$P = 10$	$P = 20$
$N = 100$	1.385	1.607	1.152
$N = 200$	1.592	3.440	2.954
$N = 400$	1.716	5.310	6.467

Table 2: Speed-up Factors

Figure 3.1: Speed-up Curves



Source Code for Submission

Source code will be submitted along with this report. Source code can also be found in this git repo (Clickable Link).

References

[1] Juan Li and Qingmeng Wei. Optimal control problems of fully coupled fbsdes and viscosity solutions of Hamilton-Jacobi-Bellman equations. *SIAM Journal on Control and Optimization*, 52(3):1622 – 1662, 2014.

[2] Dirac and P. A. M. *The principles of quantum mechanics*. Clarendon Press, Oxford, 1981.

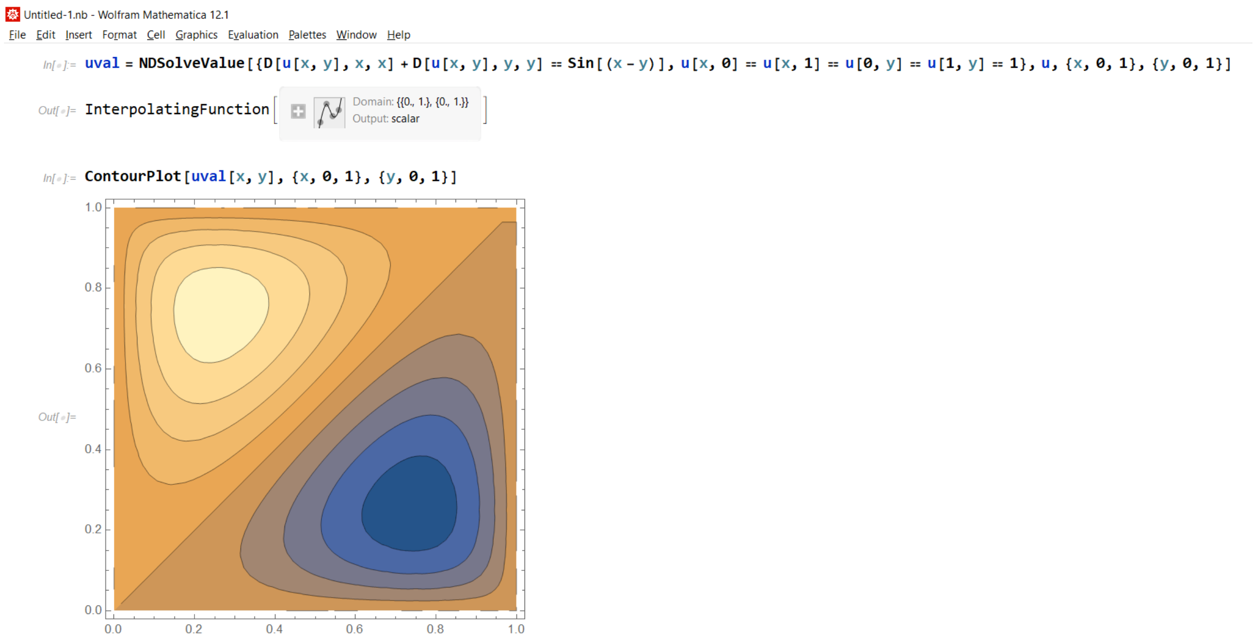
[3] Andrew R. and D. F. Griffiths. *The finite difference method in partial differential equations*. Wiley, Chichester, [England], 1980.

[4] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. Other Titles in Applied Mathematics. SIAM, second edition, 2003. ISBN 978-0-89871-534-7. doi: 10.1137/1.9780898718003. URL `http://www-users.cs.umn.edu/~{}saad/IterMethBook_2ndEd.pdf`.

[5] URL `https://www.sharcnet.ca/help/images/4/4f/Parallel_Numerical_Solution_of_PDEs_with_Message_Passing.pdf`.

Appendix

Figure 3.2: Wolfram Mathematica Output


Untitled-1.nb - Wolfram Mathematica 12.1
File Edit Insert Format Cell Graphics Evaluation Palettes Window Help
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0

```
In[ ]:= uval = NDSolveValue[{D[u[x, y], x, x] + D[u[x, y], y, y] == Sin[(x - y)], u[x, 0] == u[x, 1] == u[0, y] == u[1, y] == 1}, u, {x, 0, 1}, {y, 0, 1}]

Out[ ]= InterpolatingFunction[ Domain: {{0., 1.}, {0., 1.}} Output: scalar ]

In[ ]:= ContourPlot[uval[x, y], {x, 0, 1}, {y, 0, 1}]

Out[ ]=
```

Figure 3.3: My Program Output

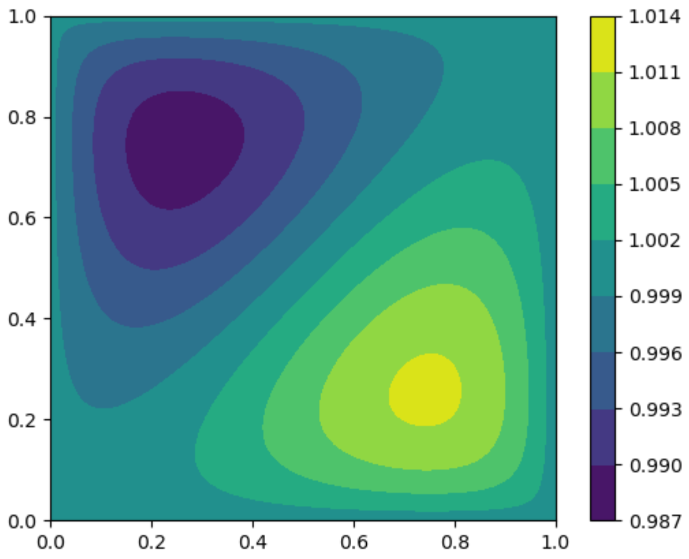


Figure 3.4: Convergence of Jacobi Method