CSCI 4576-5576 High Performance Scientific Computing
Lab02 Report
Varun Hoskere and Roseanna Neupauer

Parallelization design description :

The goal of this assignment was to solve the two-dimensional, steady-state temperature diffusion equation for a unit square, using multiple processors. The domain is discretized into rows and columns, with an example shown in Figure 1 for 8 rows and 8 columns. In Figure 1, each circle represents a node where temperature is calculated. The circles are numbered either as (column, row) shown above the node or as a node number ranging from 1 to N_n, where N_n is the total number of nodes. The diffusion equation is discretized to be solved on this grid, and the solution is given by $\mathbf{A}\boldsymbol{\phi} = \mathbf{b}$, where $\boldsymbol{\phi}$ is an N_n -length vector of simulated temperature at each node, $\mathbf{A}$ is an N_n x N_n matrix of coefficients, and $\mathbf{b}$ is an N_n-length vector that incorporates the boundary temperatures.

The domain is discretized to be solved on multiple processors. An illustration of the discretization is shown in Figure 1 where each shaded region is solved on a different processor. The grids for each processor are shown in Figure 2. They include the grid of the main domain, plus a one-cell border of ghost cells (shaded in gray in Figure 2). Each processor solves a sub-problem, given by $\mathbf{A}^p\boldsymbol{\phi}^p = \mathbf{b}^p$, where
where $\boldsymbol{\phi}^p$ is an (nRealx + 2)(nRealy+2)-length vector of simulated temperature at each node in the subdomain for processor p, where nRealx and nRealy are the number of nodes from the full grid in the x and y directions that are passed to processor p, $\mathbf{A}^p$ is an (nRealx + 2)(nRealy+2) x (nRealx + 2)(nRealy+2) matrix of coefficients, and $\mathbf{b}^p$ is an (nRealx + 2)(nRealy+2)-length vector that incorporates the boundary temperatures. The boundary temperatures include temperatures on all boundaries of the sub grid, which may include boundaries of the real domain or connections with adjacent sub grids for other processors. Let processor p share a sub grid boundary with processor q. The temperatures of one row or column of the main grid is calculated by both processor p and processor q (see Figure 2). Then part of the solution $\boldsymbol{\phi}^q$ is used as boundary temperatures for processor p. The boundary temperature are assigned to the row or column ghost cells in processor p that are adjacent to processor q, and they are taken from row or column in processor q that is one row or column on the inside of the cells that calculated on both processors.

All processors solve their sub-problem by $\mathbf{A}^p\boldsymbol{\phi}^p = \mathbf{b}^p$ using Jacobi iteration. After each iteration, information is exchanged between the processors (following the arrows in Figure 2), and the sub-problem is solved again. The process repeats until the temperatures at cells that are calculated on multiple processors all converge to the same value within a designated tolerance.

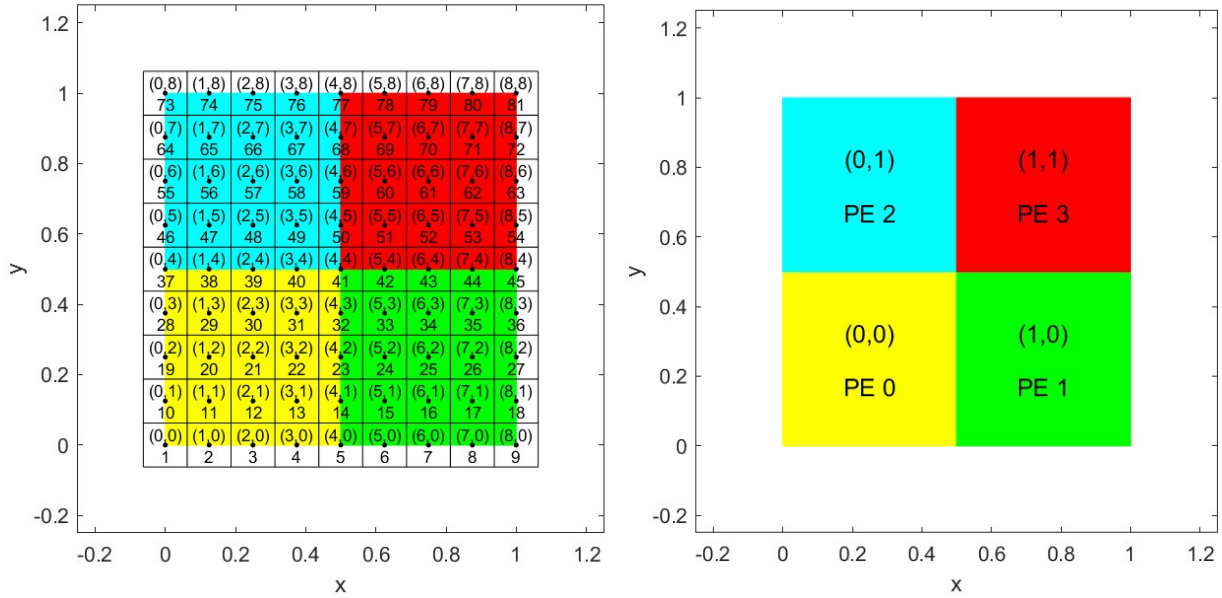

Figure 1. Physical domain of problem with discretization (left) and distribution to processers (right). Colors represent regions that are evaluated on a one processor. Small black circles in left plot represents nodes where temperature is calculated, labeled both with (row, column) above the node and natural number below the node. Text on right represents processor number as (row, column) and natural number.

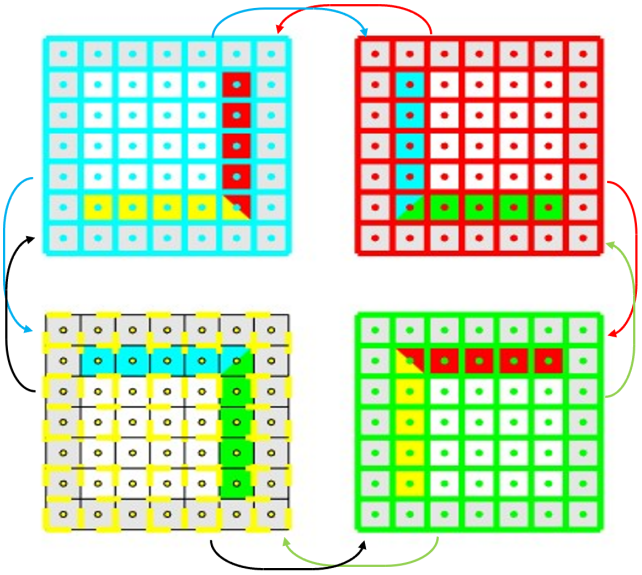

Figure 2. Sub grid for each processor. Grids are color-coded to match processors in Figure 1. Gray cells are ghost cells that are outside of the main grid region shown in Figure 1. The cells that are calculated by two processors are shaded with the color of the processor on which they are duplicated. For example, the red-shaded cells in the green grid are the same nodes as the greed-shaded cells in the red grid. The arrow represents cells from one processor that are passed from one processor to another. The tail of the arrow is located at the row or column that is sent, and the head of the arrow is at the row or column that is received.

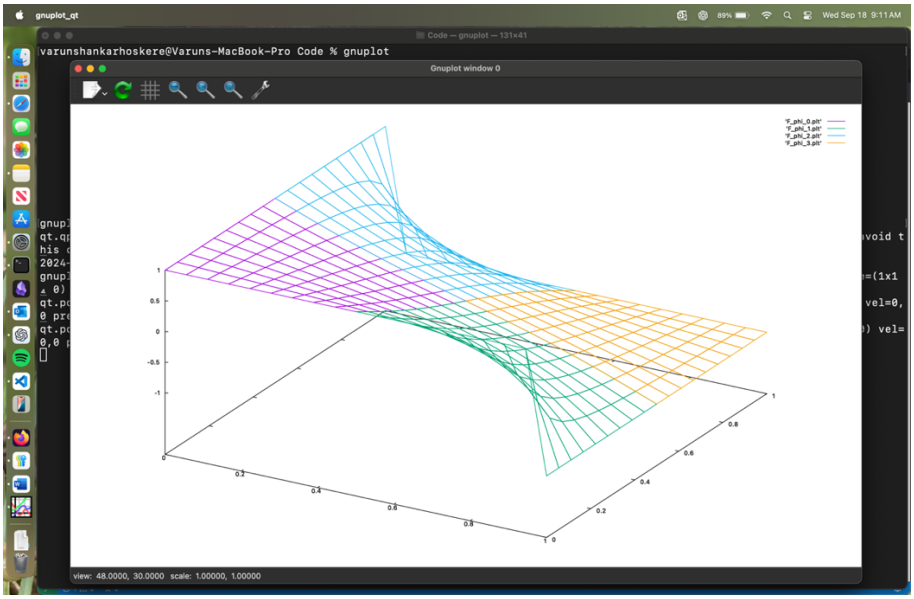

Figure 3 : Plot for test case when running on 4 PEs (2 in the x- direction and 2- in the y- direction)

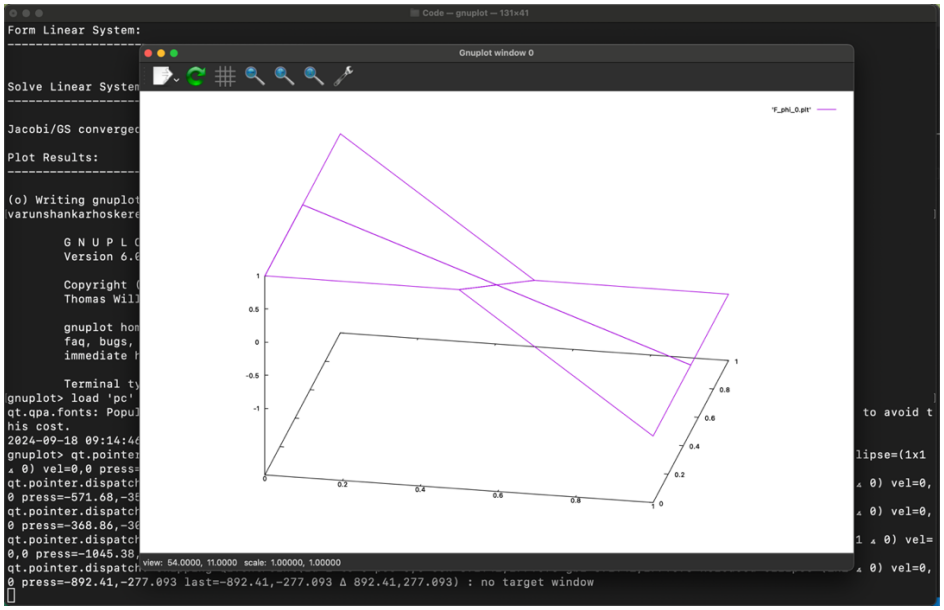

Figure 4 : Plot for test case when running on 1 PE

Results of two simulations are shown in Figure 3 (4 processors) and Figure 4 (1 processor). Figure 3 shows that the solutions from the four processors are identical at the sub-grid boundaries, demonstrating that the parallelization was successful. Comparison of Figures 3 and 4 shows that both simulations produced the same results (although the spatial discretization is different), further demonstrating that the parallelization was successful.

Self-Evaluation:

After discussing the approach with Varun, we both modified our own codes. My code compiled but had a run-time error that occurred in the step that incorporates the received information into the $\mathbf{b}^p$ vector. After getting help from the professor regarding passing arrays rather than array addresses into MPI_Isend and MPI_Irecv, the code ran through one iteration, but it got stuck on the sending or receiving on the second iteration. I am still trying to figure out the issue.

- Neupauer

After discussing the approach with Neupauer, I came to realize that I had many misunderstandings. I was not clear on whether the phiSend_n() vector was for sending to the North or sending from the North, and similarly for the other directions.

We spent a lot of time working on the indexing for the Solution vector. Both of us had different thoughts on how to iterate the vector and on how the mapping between cells and indexes worked. Initially we had not used the pid() function to map the cells to their indexes. We disagreed on whether it was even needed, but eventually realized that we had to use it one way or another.

After some initial work, the plots I was seeing were discontinuous along the North-South direction and continuous along the East-West direction. After discussing this during office hours, Dr. Runnels pointed out that my computation of the North-South neighbors had been switched up. After correcting that silly mistake, my plots became continuous along all directions, and I was able to finish the requirements for his lab.

- Varun Hoskere

Appendix A: `GridDecomposition` routine from `mpiInfo.h`

```
void GridDecomposition(int _nPEx, int _nPEy, int nCellx, int nCelly)
  {

    nRealx = nCellx;
    nRealy = nCelly;

    // Store and check incoming processor counts

    nPEx = _nPEx;
    nPEy = _nPEy;

    if (nPEx * nPEy != numPE)
    {
      if (myPE == 0)
        cout << "Fatal Error:  Number of PEs in x-y directions do not add up to numPE" << endl;
      MPI_Barrier(MPI_COMM_WORLD);
      MPI_Finalize();
      exit(0);
    }

    // Get the i-j location of this processor, given its number.  See figure above:

    jPE = int(myPE / nPEx);
    iPE = myPE - jPE * nPEx;

    // Set neighbor values

    nei_n = nei_s = nei_e = nei_w = -1;

    if (iPE > 0)
    {
      nei_w = myPE - 1;
    }
    if (iPE < nPEx - 1)
    {
      nei_e = myPE + 1;
    }
    if (jPE < nPEy - 1)
    {
      nei_n = myPE + nPEx;
    }
    if (jPE > 0)
    {
      nei_s = myPE - nPEx;

    }

    // TO-DO in Lab : nei_s
    // TO-DO in Lab : nei_e
    // TO-DO in Lab : nei_w
    // done

    countx = nRealx + 2;
    county = nRealy + 2;

    phiL = new double[county];
    phiR = new double[county];
```

```
    phiT = new double[countx];
    phiB = new double[countx];

    phiSend_n = new double[countx];
    phiSend_s = new double[countx];
    phiSend_e = new double[county];
    phiSend_w = new double[county];

    phiRecv_n = new double[countx];
    phiRecv_s = new double[countx];
    phiRecv_e = new double[county];
    phiRecv_w = new double[county];

    tag = 0;
  }
```

Appendix B: ExchangeBoundaryInfo routine from mpiInfo.h

```
void ExchangeBoundaryInfo(VD &Solution, VD &b)
  {
    sLOOP phiSend_n[s] = 0.;
    sLOOP phiSend_s[s] = 0.;
    tLOOP phiSend_e[t] = 0.;
    tLOOP phiSend_w[t] = 0.;

    // ---------------------------------------------
    // Parallel communication on PE Boundaries       ** See fd.h for tLOOP and sLOOP macros **
    // ---------------------------------------------

    // (1.1) Put values into communication arrays

    // s and t are maps from the actual cell to its place in the vector.
    // don't forget to map the s , t to the actual cell number with pid()

    sLOOP phiSend_n[s] = Solution[pid(s, nRealy-1)];         // needs the value to be sent to
thenorthern neighbour. sending from bottom most row
    sLOOP phiSend_s[s] = Solution[pid(s, 2)];              // needs the value to be sent to the
southern neighbour 2*countX + s

    tLOOP phiSend_w[t] = Solution[pid(2, t)];          // needs the value to be sent to the western
neighbour
    tLOOP phiSend_e[t] = Solution[pid(nRealx-1, t)];             // needs the value to be sent to
the easter neighbour i=2

    // (1.2) Send them to neighboring PEs

    // < ? >
//                                    vector , szie,   type,
    if (nei_n >= 0) err = MPI_Isend(phiSend_n, countx, MPI_DOUBLE, nei_n, 0, MPI_COMM_WORLD,
&request);
    if (nei_s >= 0) err = MPI_Isend(phiSend_s, countx, MPI_DOUBLE, nei_s, 0, MPI_COMM_WORLD,
&request);
    if (nei_e >= 0) err = MPI_Isend(phiSend_e, county, MPI_DOUBLE, nei_e, 0, MPI_COMM_WORLD,
&request);
    if (nei_w >= 0) err = MPI_Isend(phiSend_w, county, MPI_DOUBLE, nei_w, 0, MPI_COMM_WORLD,
&request);

    // (1.3) Receive values from neighobring PEs' physical boundaries.

    if (nei_n >= 0)
    {
      err = MPI_Irecv(phiRecv_n, countx, MPI_DOUBLE, nei_n, 0, MPI_COMM_WORLD, &request);
      MPI_Wait(&request, &status);
    }
    if (nei_s >= 0)
    {
      err = MPI_Irecv(phiRecv_s, countx, MPI_DOUBLE, nei_s, 0, MPI_COMM_WORLD, &request);
      MPI_Wait(&request, &status);
    }
    if (nei_e >= 0)
    {
      err = MPI_Irecv(phiRecv_e, county, MPI_DOUBLE, nei_e, 0, MPI_COMM_WORLD, &request);
      MPI_Wait(&request, &status);
    }
    if (nei_w >= 0)
```

```
    {
      err = MPI_Irecv(phiRecv_w, county, MPI_DOUBLE, nei_w, 0, MPI_COMM_WORLD, &request);
      MPI_Wait(&request, &status);
    }

    // (1.4) Update BCs using the exchanged information
    // b is the boundary, so update there

    if (nei_n >= 0) sLOOP b[pid(s, nRealy+1)] = phiRecv_n[s];
    if (nei_s >= 0) sLOOP b[pid(s, 0)] = phiRecv_s[s];
    if (nei_e >= 0) tLOOP b[pid(nRealx+1, t)] = phiRecv_e[t];
    if (nei_w >= 0) tLOOP b[pid(0, t)] = phiRecv_w[t];
  }

  int pid(int i, int j) { return (i + 1) + (j) * (nRealx + 2); }
};
```