HIGH PERFORMANCE SCIENTIFIC COMPUTING - LAB 2

Shriprajwal Krishnamurthy, Nick Rovito

September 19, 2024

Parallelization Design

In this lab, we parallelize a diffusion solver using the finite difference (FD) method and an iterative jacobi solver. When the computational domain is large, it is often necessary to divide the domain to several processes to prevent excessive computing time. Here, we implement a parallel communication design on a 2D grid using MPI.

We solve the Laplace equation on a unit square (Eq. 1). The domain is written as Ω and the boundaries are $\Gamma = \partial\Omega$. The solution function is u, representing the temperature profile in the domain. The boundary conditions are given for this problem in Equation 2.

$$\nabla^2 u = 0 \tag{1}$$

$$\begin{aligned} u(\underline{\boldsymbol{x}}) &= 1 \ \forall \ \underline{\boldsymbol{x}} \ \in \ \Gamma_B \\ u(\underline{\boldsymbol{x}}) &= 1 \ \forall \ \underline{\boldsymbol{x}} \ \in \ \Gamma_L \\ u(\underline{\boldsymbol{x}}) &= -1 \ \forall \ \underline{\boldsymbol{x}} \ \in \ \Gamma_T \\ u(\underline{\boldsymbol{x}}) &= -1 \ \forall \ \underline{\boldsymbol{x}} \ \in \ \Gamma_R \end{aligned} \tag{2}$$

We provide the number of processes and the number of elements in the x and y directions owned by each process. This creates a grid of processes through which we must share data. We accomplish this by sending solution values within the real domain on one process to neighboring processors' ghost nodes. These sent values become Dirchlet boundary conditions for adjoining processes. If we let $nRealx$ and $nRealy$ be the number of cells in the simulated physical domain and $u_p(e_x, e_y)$ be the nodal solution for process p, these boundary conditions can be written as:

$$\begin{aligned} \Gamma_{B,north} &= u_p(e_x, nRealy - 1) \\ \Gamma_{T,south} &= u_p(e_x, 2) \\ \Gamma_{R,\ west} &= u_p(2, e_y) \\ \Gamma_{L,\ east} &= u_p(nRealx - 1, e_y) \end{aligned} \tag{3}$$

Here, $\Gamma_{B,north}$ denotes the lower boundary of the processor north to one sending the solution. Each process sends and receives these boundary conditions. If the process contains elements that are on the physical domain boundary, it's boundary conditions follow those described in Equation 2.

The process ID for the north (nei_n), east (nei_e), south (nei_s), and west (nei_w) processors are given as:

$$\begin{aligned} nei_n &= myPE - nPEx \\ nei_e &= myPE + 1 \\ nei_s &= myPE - nPEx \\ nei_w &= myPE - 1 \end{aligned} \tag{4}$$

where $myPE$ is the current processor and $nPEx$ is the number of processes in the x direction. These values are only assigned if there is a neighboring process.

Results

Figure 1 shows the finite difference solution using jacobi iterations when run on 1 processor. Figure 2 shows the finite difference solution using jacobi iterations when run on 4 processors. In both cases we use 10 cells in the x and y directions per process. This means that the 4 processor case is 4x more resolved.

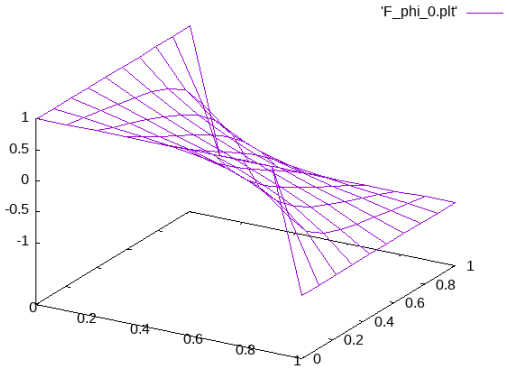

Figure 1: The MPI FD solution using 1 process and 10 nodes in the x and y directions.

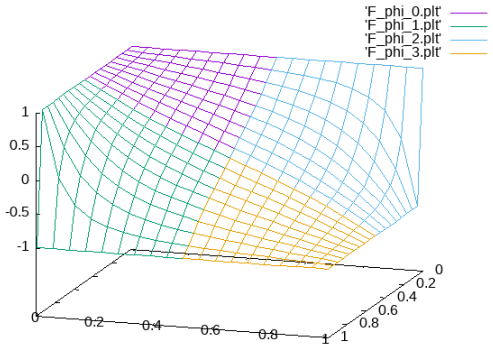

Figure 2: The MPI FD solution using 4 processes and 10 nodes in the x and y directions on each process.

Self Evaluation

Overall we think the lab went well. We ran into a bunch of errors (mpi_ERR_RANK: invalid rank, Double Free, etc) that took quite some time to debug. We wrote a small script in the *GridDecomposition* function that writes the neighboring processor ids to help us debug this section. The most important part of the lab was to realize that we need to set the column/row as 2 in phiSend(sLOOP phiSend_s[s] = Solution[pid(s, 2)];).

APPENDIX A: GridDecomposition Routine

```
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;  //init to -1 (if no neighbor)

    if(iPE > 0){
        nei_w = myPE - 1;  // West neighbor
      }

    if(iPE < nPEx - 1){
        nei_e = myPE + 1;  // East neighbor
    }

    if(jPE > 0){
        nei_s = myPE - nPEx;  // South neighbor
    }

    if(jPE < nPEy - 1){
        nei_n = myPE + nPEx;  // North neighbor
    }

    // For rank debugging
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    std::string filename = "neighbors_rank_" + std::to_string(rank - 1) + ".dat";
    std::ofstream file;
    file.open(filename, std::ios::out | std::ios::trunc);
    if (file.is_open()) {
            file << "Rank = " << rank << std::endl;
            file << "North neighbor: " << nei_n << std::endl;
            file << "South neighbor: " << nei_s << std::endl;
            file << "East  neighbor: " << nei_e << std::endl;
            file << "West  neighbor: " << nei_w << std::endl;
            file.close();
    }

    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

```
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
    sLOOP phiSend_n[s] = Solution[pid(s, nRealy - 1)];
    sLOOP phiSend_s[s] = Solution[pid(s, 2)];
    tLOOP phiSend_w[t] = Solution[pid(2, t)];
    tLOOP phiSend_e[t] = Solution[pid(nRealx - 1, t)];

    // (1.2) Send them to neighboring PEs

    // int MPI_Isend(const void* buffer, int count, MPI_Datatype datatype, int recipient, int tag,
    MPI_Comm communicator, MPI_Request* request);

    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
    // Set b vector from adjoining processes
    if (nei_n >= 0) sLOOP b[pid(s, nRealy + 1)] = phiRecv_n[s] ;  // recieving from n
    if (nei_s >= 0) sLOOP b[pid(s, 0)]          = phiRecv_s[s] ;  // recieving from s
    if (nei_w >= 0) tLOOP b[pid(0, t)]          = phiRecv_w[t] ;  // recieving from w
    if (nei_e >= 0) tLOOP b[pid(nRealx + 1, t)] = phiRecv_e[t] ;  // recieving from e

}
```