CSCI 5576: Lab Report 7

Kartik Sharma

Luna McBride

May 14, 2024

Contents

1 Introduction

The goal of Lab 8 is to use the knowledge that we have gained in class regarding MPI to parallelize a linear solver on our own. We will be implementing portions of the conjugate gradient algorithm making sure to parallelize it with MPI.

2 CG Parallelization Description

In order to convey how this CG function was implemented using parallelism, we will first take a look at the first task that we needed to complete which was implementing the `Dot` function. From the conjugate gradient method from Wikipedia:

$$a_k = \frac{r_k^T * r_k}{p_k^T * A * P_k}$$

The numerator of this conjugate gradient function tells us the Dot product that needs to occur in the CG function. Given that we have two Vectors Vec1 and Vec2 we can simply take the dot product of these two vectors and divide it by a factor that allows us to make sure that the specific process is only adding its contribution to the overall Dot product, so that when the `MPI_Allreduce` is called we are not double/quadruple adding up on the boundary nodes. Below is a function that represents what is going on to calculate the dot product where n is the length of the vector size:

$$\sum_{i=0}^{n} \frac{vec1[i] * vec2[i]}{scalarfactor[i]}$$

In order to accomplish this we will initialize a vector with the size of vec1 and make sure that all of its values correspond to 1. Then with this newly initialized vector, we will send it over to the function PEsum which will turn all the values in this vector to be equal to how many PEs share that partiucular node in the vector, which is also a mesh. For example in `Figure 1` below after calling PEsum on this newly created vector in perspective of PE_0, we can see that for `case 1` the mesh values in the vector will be 1, and for case 2 all those nodes will get a value of 2 since 2 PEs share those nodes, and for case 3 those nodes will get a value of 4 since 4 PEs share that specific node. This is because, when PEsum is called the neighboring nodes send the value 1 to each other and that is summed up. Because the vectors passed in this function and the newly created vector are meshes that correspond to the same point for that given process, to compute the Dot product we can compute the Dot product normally and then just divide by this scaling factor to make sure that this process only adds its contribution to the overall sum of the Dot product when `MPI_Allreduce` is called. Therefore, for case 1 we will divide the dot product by 1, for case 2 we will divide the dot product by 2 and for case 3 we will divide the dot product by 4. This is just considering boundary nodes, all interior mesh nodes will just be divided by 1 since those are process-specific and won't have issues with double counting edges.

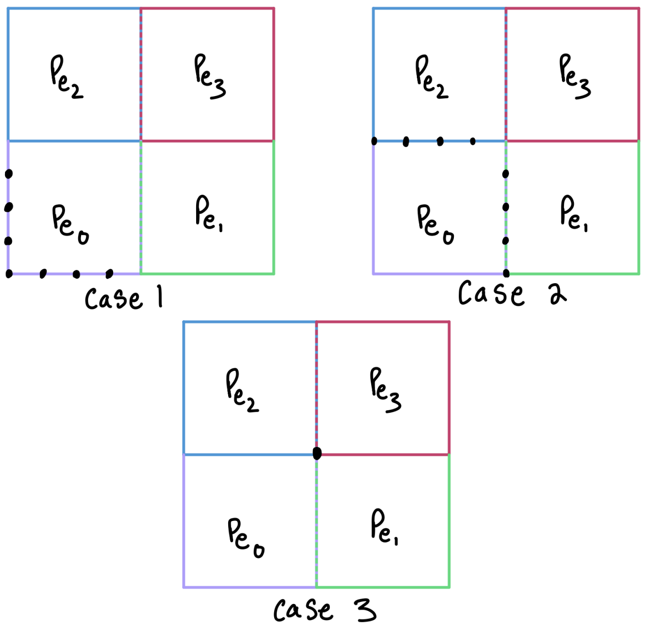


Figure 1: Cases of neighboring processes when doing Dot product in terms of Pe_0

Additionally, the `MatVecProd`, which is computing Acoef * p and storing that result into `prod`, is used in computing `A *` P_K in the equation above and throughout the conjugate gradient method. All that is needed for parallelization in this function is to make sure that `prod` is synchronized across all neighboring PE's, so it handles PE boundaries which is just to call PESum on prod.

The next portion necessary for parallelization is the b_PEsum variable, which is used to collect the right hand side values across all PEs. This simply needed to be filled by the current row values then sent to PEsum once again. PEsum is in the name, so it is not much of a stretch to believe that it would need to be sent to PEsum to account for the RHS values on all PEs.

The final portion required for us to parallelize was collecting all of the converged variables from the

PEs into a global variable, aptly named global_converged. This was achieved with an MPI Allreduce call, MPI_Allreduce(&converged, &global_converged, 1 , MPI_INT, MPI_MIN, MPI_COMM_WORLD);. This call is saying, in layman's terms, to take all of the converge variables from the different PEs and reduce them all into the global_converged variable.

3 Jacobi-CG Comparison:

3.1 Equivalent Output for CG and Jacobi

From plotting Jacobi (Figure 2) and CG (Figure 3) for the same problem size it can be conveyed that they are identical, which demonstrates that our code produces the same solution with differing the type of solver.

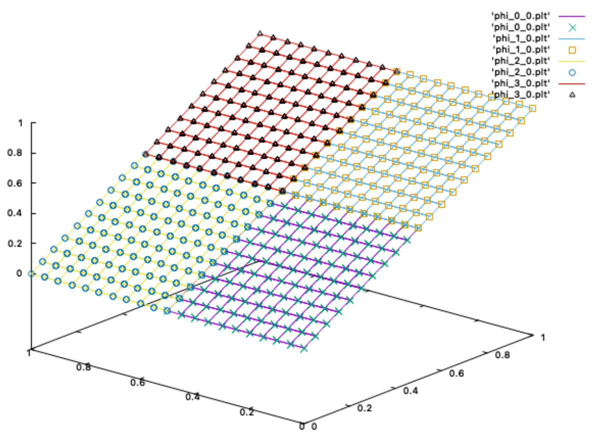


Figure 2: Plot from `./solvers -nPEx 2 -nPEy 2 -nCellx 10 -nCelly 10 -solver jacobi`

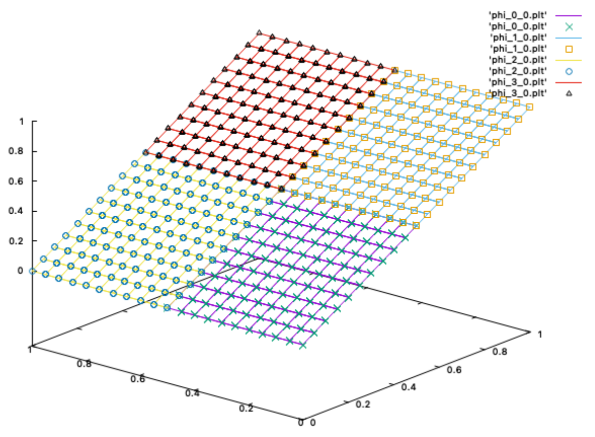


Figure 3: Plot from `./solvers -nPEx 2 -nPEy 2 -nCellx 10 -nCelly 10 -solver cg`

3.2 Execution and Iteration Time Comparison

To compare jacobi and conjugate gradient method number of iterations and runtime both solvers will be tested on 4 processes 2 in the x direction and 2 in the y direction. Each having 10 cells in both in the x and y directions. Below is a table that summarizes the results:

CG Interations	**Jacobi Interations**	**Jacobi Runtime (seconds)**	**CG Runtime (seconds)**
75	372	0.005261	0.003299

Table 1: Iterations and Runtimes for 4 PEs (2x,2y) and 10 Xcells and 10 Ycells

As we can see for this particular problem, CG has lower iterations to convergence and a faster runtime than Jacobi. However, both solvers have two different convergence criteria. This was not changed for the comparison where Jacobi has a convergence criteria of 1.e-04 and CG has a convergence criteria of 1.e-10. Even though CG's convergence criteria is lower than Jacobi's, it still was able to have a faster runtime and converge in a lower amount of iterations.

4 CG Speed-up Study:

Below is a table that shows how the runtimes are affected when the problem space is broken up into different numbers of PEs using the conjugate gradient method:

nCells	nPEs	Runtime
100 x 100	1	0.476075 seconds
50 x 50	2	0.12311 seconds
33.3 x 33.3	3	0.055964 seconds
25 x 25	4	0.029906 seconds

Table 2: Runtimes for a range of PE's and Cells for a fixed grid size of 100x100

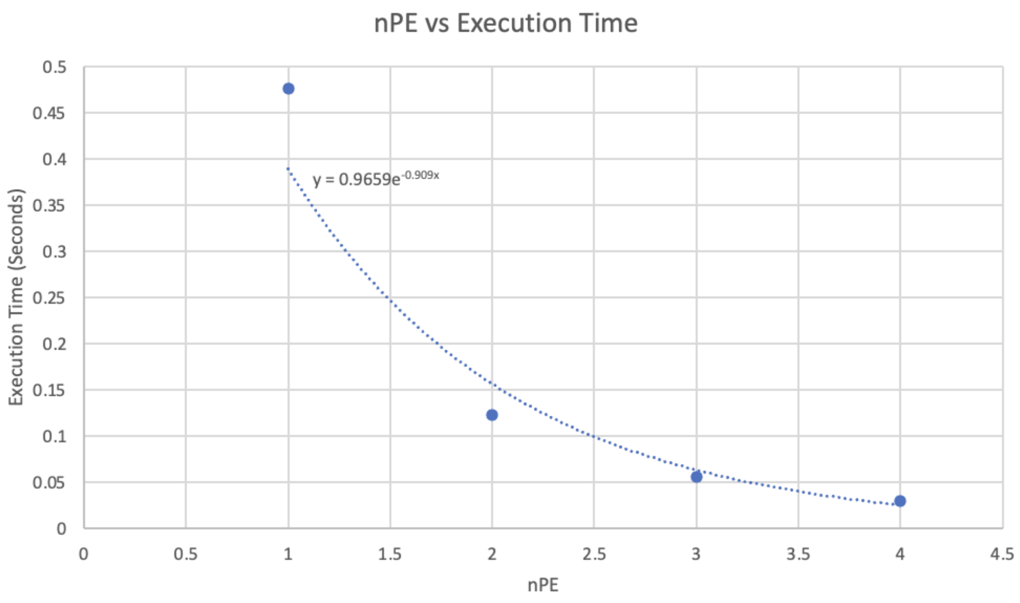


Figure 4: Plot of nPEs vs Execution Time for constant grid size

Looking at the table 2 above, when comparing the problem space where we have 1 PE completely solving a grid size of 100 and then splitting up the grid size to 2,3, and 4 PEs we can see that having more PEs to split the problem size does in fact reduce the runtime of the program. From Figure 4, the graph conveys that splitting the problem space into multiple PEs is best fitted by an exponential decay function. This conveys that as we add more PEs in the start to split the problem we get a huge reduction in execution time, however as we keep increasing the number of PEs the amount of gain that we receive starts to decrease.

5 Self Evaluation

5.1 Kartik

This lab was interesting because of the obstacle that was present to resolve the dot product function. After debugging and discussing with classmates, it was interesting to figure out the problem that was occurring was double/quadrupling values on the boundary. After the issue was found, it then took careful consideration to resolve the issue using code, which was very exciting. Overall, this lab was hard because of the obstacle mentioned but other than that it went well and as partners, we worked well together.

5.2 Luna

I feel like this was an interesting section to look into. CG is clearly the way to go long term, given that it proved to be much faster than Jacobi. I have been complaining about Jacobi for multiple labs now, so it is good to see it get the boot. My runs before changing anything were all slower than Kartik's to start with. I had also misunderstood the speedup this whole time in thinking I had to update the code to speed it up rather than showing how parallelizing the code sped up the code. I found this crazy loophole that removed some ifs and caused the code to go much faster. I noted this in the Appendix section, not just because it is interesting, but also so that the sunk cost is not actually sunk.

Another thing I noted while trying to find changes was that removing a variable was slower when that variable was to be put into another equation, which is different from how I expected based on the previous lab. p_dot_ap, for example, was only used once, so I attempted to just put that Dot function call into the alpha equation. This proved significantly slower, which I found surprising. It only makes sense that it is slower when being used in that other math specifically.

6 Appendix A: Source code listing

6.1 Main Code

```
//  ==
//  ||
//  || Jacobi Left The Same
//  ||
//  ==

double Dot(VD &vec1, VD &vec2, mpiInfo &myMPI)
{
    double local_dot = 0.;
    double global_dot = 0.;

    // 1. Create a vector of ones, since vec is a mesh of its process, so will this onesVec be a mesh
    VD onesVec;
    onesVec.resize(vec1.size(), 1.0); //since vec1 and vec2 are the same size doesn't matter which one
         we do

    // 2. Use PEsum on this vector that we created, since it is an artificial mesh and we are sending
        1's on PEsum on nodes that are shared with be incremented by the number of PEs that share it
    myMPI.PEsum(onesVec);

    // Compute the dot product on this PE.
    rLOOP
    {
        // 3. Adjust each entry by dividing by the count (i.e., the value in onesVec), which will make
            sure that each PE only adds its portion to the overall dot prodct

        local_dot += (vec1[r] * vec2[r]) / (onesVec[r]);
    }

    // call reduce to sum every process's indiviudal dot product and store that into global dot
        product and return that.
    MPI_Allreduce(&local_dot, &global_dot, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
    return global_dot;


}




//  ==
//  ||
//  || Utility routine: MatVecProd
//  ||
//  || Computes the matrix-vector product, prod = A*p where p is complete
//  || on each PE but A must be summed on PE boundaries.
//  ||
//  ==

void MatVecProd(VD &p , VD &prod , mpiInfo &myMPI)
{

  // Serial computation on this PE
  rowLOOP
    {
      prod[row] = 0.;
      colLOOP
        {
          int Acol = Jcoef[row][col];
          if ( Acol > 0 ) prod[row] += Acoef[row][col] * p[ Acol ];
        }
    }

  // Handle PE boundaries

  /* TO-DO */
  /*  ... */
  /*  ... */
  /*  ... */
  /* TO-DO */
  //since we are computing Acoef * p and storing that in prod, to handle boundary conditions on prod
       we can give it over to PEsum
  myMPI.PEsum(prod);


}

//  ==
//  ||
//  || Utility routine: Residual
//  ||
//  || Computes the residual, residual = b - A*Sol, where Sol is complete on
//  || each PE but b and A must be summed at PE boundaries.
//  ||
//  ==

void Residual(VD &residual , VD &Sol , VD &RHS, mpiInfo &myMPI)
{

  MatVecProd(Sol , residual , myMPI);

  rowLOOP residual[row] = RHS[row] - residual[row];

}



// ==
// ||
// || CG (Conjugate Gradient)
// ||
// || Solves the system using Conjugate-Gradient iteration.
// ||
// || See https://en.wikipedia.org/wiki/Conjugate_gradient_method
// ||
// ==

void CG(VD &Solution , mpiInfo & myMPI)
{
```

```
VD rnew; rnew.resize(nField + 1);
VD r;      r.resize(nField + 1);
VD p;      p.resize(nField + 1);
VD Ap;    Ap.resize(nField + 1);
VD Ax;    Ax.resize(nField + 1);

double p_dot_Ap;      // Stores matrix-vector product
double r_dot_r;       // Stores r dot r
double rnew_dot_rnew; // Stores rnew dot rnew
double alpha;         // Alpha in the above-referenced algorithm
double beta;          // Beta  "  "   "      "         "
double cur_delta;
double tol          = 1.e-04;
int global_converged = 0;
int iter            = 0;
int max_iter        = nField * 10;
int converged;

// (1) Initial guess and other initializations

rowLOOP Solution[row] = p[row] = r[row] = 0.;

Solution[0] = 0.;
p       [0] = 0.;
r       [0] = 0.;

// (2) Prepare for parallel computations on RHS

VD b_PEsum ;
b_PEsum.resize(nField + 1 ) ;


rowLOOP b_PEsum[row] = b[row]; //set b_PEsum values to be intialized to the right hand side which is
     b, cannot send it over to be sum empty.
myMPI.PEsum(b_PEsum); //Sum RHS vector(b_PEsum) on PE boundaries

// (3) Initialize residual, r, and r dot r for CG algorithm

Residual(r,Solution,b_PEsum,myMPI);

rowLOOP p[row] = r[row];

r_dot_r = Dot(r,r,myMPI);

// (4) CG Iterations

while ( global_converged == 0 && ++iter <= max_iter)
  {
    // (4.1) Compute alpha

    MatVecProd(p,Ap,myMPI);          // A*p (stored in Ap)
    p_dot_Ap = Dot(p,Ap,myMPI);      // p*Ap
    alpha    = r_dot_r / p_dot_Ap;

    // (4.2) Update solution and residual
```

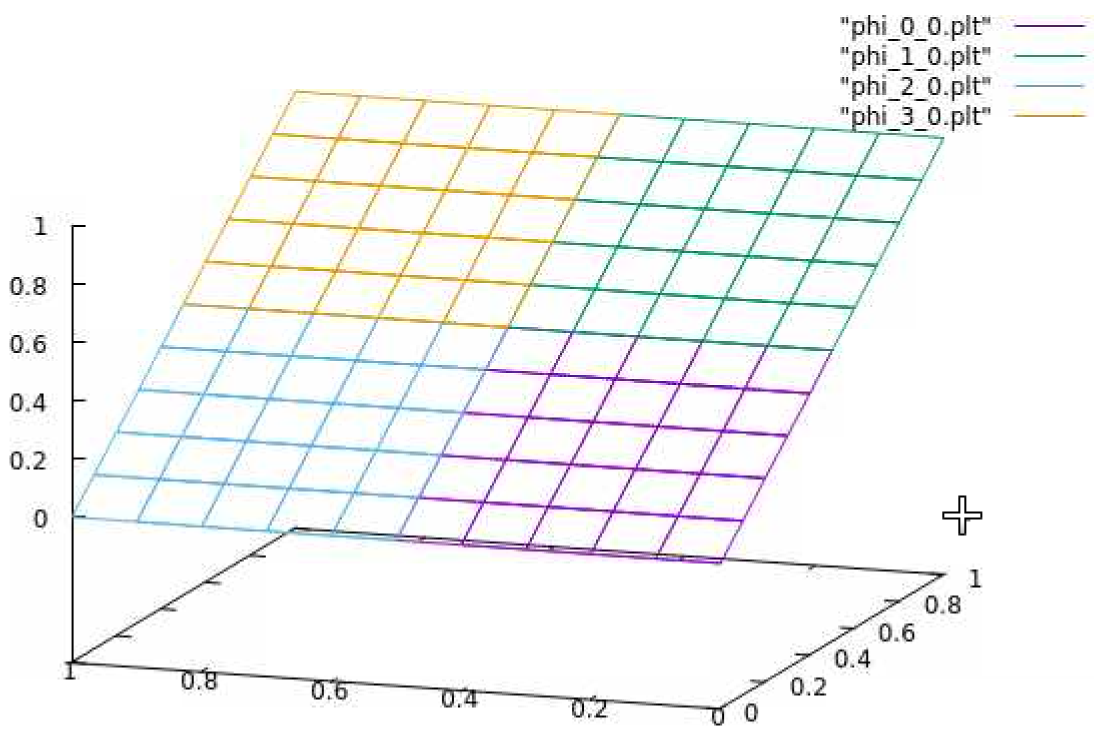
```
      rowLOOP Solution[row] = Solution[row] + alpha * p[row];
      rowLOOP rnew   [row] = r[row]      - alpha * Ap[row];

      // (4.3) Compute beta

      rnew_dot_rnew = Dot(rnew,rnew,myMPI);
      beta         = rnew_dot_rnew / r_dot_r;

      // (4.4) Update search direction

      rowLOOP p[row] = rnew[row] + beta*p[row];

      // (4.5) r "new" will be r "old" for next iteration

      rowLOOP r[row] = rnew[row];
      r_dot_r      = rnew_dot_rnew;

      // (4.6) Check convergence on this PE

      rowLOOP
        {
          cur_delta = fabs(alpha * p[row]);
          if ( cur_delta > tol ) converged = 0;
        }

      if ( fabs(r_dot_r) < 1.e-10)
        converged = 1;
      else
        converged = 0;

      // (4.7) Check convergence across PEs, store result in "global_converged"

     // Check convergence across PEs, store result in "global_converged"
    MPI_Allreduce(&converged, &global_converged, 1 , MPI_INT, MPI_MIN, MPI_COMM_WORLD); //no need to
        MPI barrier, since this call will act as a barrier.


    }

    // (5) Done - Inform user

    if ( global_converged == 1 ) if ( myMPI.myPE == 0 ) cout << "CG␣converged␣in␣" << iter << "␣
        iterations.\n" ;
    if ( global_converged == 0 ) if ( myMPI.myPE == 0 ) cout << "CG␣failed␣to␣converge␣aftger␣" <<
        iter << "␣iterations.\n" ;

}
```

6.2 Speedup Misunderstanding

There was originally a misunderstanding on speedup meaning ”find ways to make the code quicker” rather than ”show how the parallelization speeds things up”. There was still something interesting to note here that proved interesting:

```
      beta         = rnew_dot_rnew / r_dot_r;
      // (4.4) Update search direction

      rowLOOP {
        p[row] = rnew[row] + beta*p[row];

      }

      // (4.5) r "new" will be r "old" for next iteration

      rowLOOP r[row] = rnew[row];
      r_dot_r       = rnew_dot_rnew;

      // (4.6) Check convergence on this PE

      //rowLOOP
      //{
          //cur_delta = ;
          //if ( fabs(alpha * p[row]) > tol ) {
      //converged = 0;
      //break;
      //}
      //}

        if (fabs(r_dot_r) < 1.e-10)
          converged = 1;

      // (4.7) Check convergence across PEs, store result in "global_converged"

      //MPI_Barrier();
      MPI_Allreduce(&converged, &global_converged, 1, MPI_INT, MPI_MIN, MPI_COMM_WORLD);
    }

    // (5) Done - Inform user

    if ( global_converged == 1 ) if ( myMPI.myPE == 0 ) cout << "CG␣converged␣in␣" << iter << "␣
        iterations.\n" ;
    if ( global_converged == 0 ) if ( myMPI.myPE == 0 ) cout << "CG␣failed␣to␣converge␣aftger␣" <<
        iter << "␣iterations.\n" ;
     myTime.Finish(myMPI.myPE);

}
```

At this point in the code, the converged was only reset from 0 in one if statement, so the rowLOOP and else statement after were doing nothing. By removing those, as unintended as it may be, it gives the same result while going much quicker on 2x2 PEs. Here is a plot after this change to show that the end result does not change:



This is still interesting to note, despite it coming from a misunderstanding, because it shows a flaw in the underlying algorithm as it was given. This section of code was doing so many checks just to do nothing in the end besides make everything slower.