Name: Hongda Li Class: AMATH 583 SPRING 2021

6.1 Ring

todo 5

This is the code for passing message using MPI like a ring:

```
int main(int argc, char* argv[]) {
  MPI::Init();

  int token     = 0;
  size_t rounds = 1;
  if (argc >= 2) rounds = std::stol(argv[1]);

  int myrank = MPI::COMM_WORLD.Get_rank();
  int mysize = MPI::COMM_WORLD.Get_size();

  // Fix me
  int left  = 0;
  int right = 0;
  if (0 == myrank) {
    left  = myrank + 1;
    right = mysize - 1;
  }
  else {
    left = (myrank + 1)%mysize;
    right = myrank - 1;
  }

  // 0 -> 1 -> 2 -> 3 -> 0
  while (rounds--) {

    if (0 == myrank) {
      std::cout << myrank << ": sending  " << token << std::endl;

      MPI::COMM_WORLD.Send(&token, 1, MPI::INT, left, 321);         // myrank -> myrank + 1, send to rank: 1
      MPI::COMM_WORLD.Recv(&token, 1, MPI::INT, right, 321);        // Receive from: rank - 1

      std::cout << myrank << ": received " << token << std::endl;
      ++token;
    } else {
      MPI::COMM_WORLD.Recv(&token, 1, MPI::INT, right, 321);        // receive from: rank - 1
      std::cout << myrank << ": received " << token << std::endl;
      ++token;
      std::cout << myrank << ": sending  " << token << std::endl;
      MPI::COMM_WORLD.Send(&token, 1, MPI::INT, left, 321);          // myrank -> myrank + 1   (2 -> 3) send to: rank + 1
    }
  }

  MPI::Finalize();

  return 0;
}
```

Nothing spectacular.

6.2 Norm

6.2.2 Distributing the Vector

Question 1

What is your code for "mpi_norm"? (Cut and paste the code here.)

This is the changes I added inside

```
// Parallelize me -- the contents of vector x on rank 0 should be randomized and scattered to local_x on all ranks
// pass the pointer using &x(0), &x(0) + x.num_rows()
Vector local_x(num_elements);
MPI::COMM_WORLD.Scatter(myrank == 0? &x(0):nullptr, num_elements, MPI::DOUBLE, &local_x(0), num_elements, MPI::DOUBLE, 0);
```

This line of code scatter the content of the global vector from the root node with rank 0 to all the other nodes. num_elements is the size that got partitioned by MPI Scatter.

This is the code I used to compute the norm inside each of the individual node on the remote:

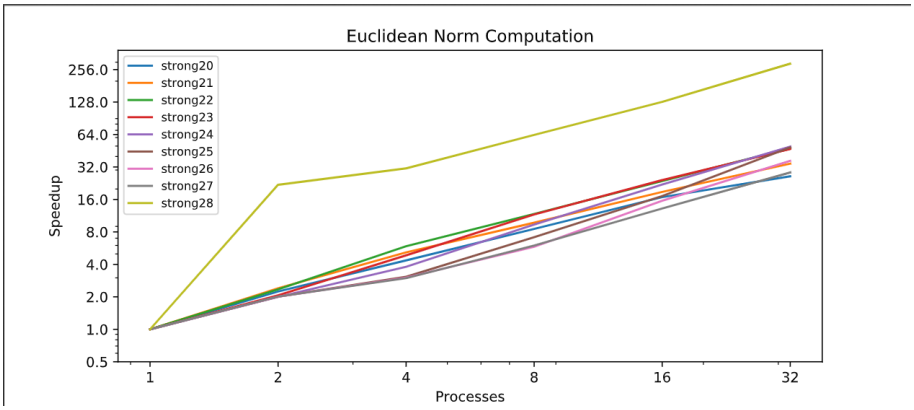
```
double mpi_norm(const Vector& local_x) {
  double global_rho = 0.0;
  // Write me -- compute local sum of squares and then REDUCE
  // ALL ranks should get the same global_rho  (that was a hint) --> All Reduce
  double PartialSum = std::inner_product(&local_x(0), &local_x(0) + local_x.num_rows(), &local_x(0), 0.0);
  MPI::COMM_WORLD.Allreduce(&PartialSum, &global_rho, 1, MPI::DOUBLE, MPI::SUM);
  return std::sqrt(global_rho);
}
```

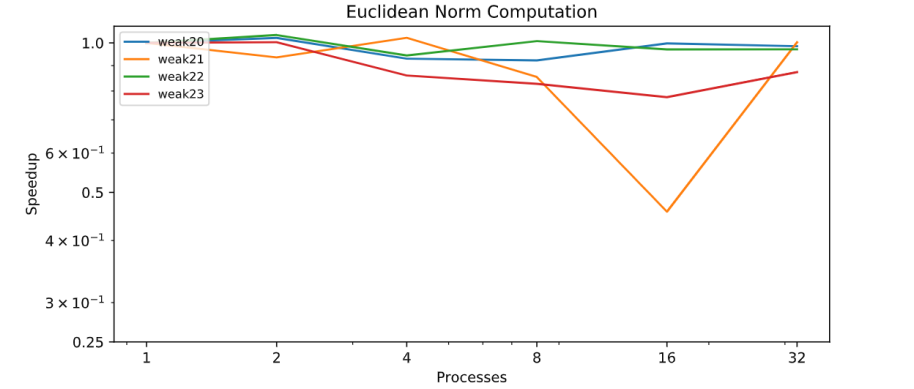
Where, we used the reduce all method after getting the partial sum. The MPI COMM will sum up all the partial sum and broadcast the results to all nodes. And all nodes will have the same answers.

Question 2

Per our discussions in lectures past about weak vs strong scaling, do the plots look like what you would expect? Describe any (significant) differences (if any).

This is the restuls from the strong and weak scaling:





The problem obeys Amdahl's law. Meaning that the amount of sequential communications works increase with problem size. This would imply no speed up on the speed up for weak scaling. This is shwon in the graph above, the size of the vector grows as the number of computations node increase. If the amount of communcations grow together with the nodes, we expect no speed up for weak scaling.
Strong scaling keep problem the same and increase the amount of parallelization, meaning that the amount of sequential works is a constant and it has nothing to do with size of the problem.
In this case, we cas see a linear increase for speed up using multiple processes. And the scaling get's better as the problem sizes increases. One explaionation could be that bigger problem size give smaller portion of overhead for computations.

Question 3

For strong scaling, at what problem size (and what number of nodes) does parallelization stop being useful? Explain.

The graph suggest that it gets better as the number of processes scale up and the number of problem size. We are not hitting the limit yet.
In theory, strong scaling doesn't have a limit to it. The speedup grows to infinity as computing resources goes to infinty.

6.3 Solving the Laplacian Equation

6.3.1

Question 4

What changes did you make for halo exchange in jacobi? Copy and paste relevant code lines that contain your edits to your report. Provide comments in the code near your edits to explain your approach.

This is the list of modifications I made for the Jacobi's Method:

1. I write a halo exchange function that operates on a given grid.
2. I use that function inside of the Jacobi's method.
3 I accumute the error for the stopping conditions in the forloop among all the processes.

This is the halo function's body:

```
void HaloUpdate(Grid&y)
{
    int myrank = MPI::COMM_WORLD.Get_rank();
    int mysize = MPI::COMM_WORLD.Get_size();
    size_t  Width = y.num_y();
    size_t Height = y.num_x();
    int Sendup = 1, SendDown = 0;


    // COMMUTE and Hallo Exchange
    if (mysize != 1)
    {

        if (myrank == 0 || myrank == mysize - 1)
        {
            // Bottom or top row for current process's grid
            if (myrank == 0)
            {
                // Send and receive bottom row with: rank  + 1
                MPI::COMM_WORLD.Recv(&y(Height - 1, 0), Width, MPI::DOUBLE, 1, Sendup);
                MPI::COMM_WORLD.Send(&y(Height - 2, 0), Width, MPI::DOUBLE, 1, SendDown);
            }
            else
            {
                // Send and receive top row with: rank - 1
                MPI::COMM_WORLD.Send(&y(1, 0), Width, MPI::DOUBLE, myrank - 1, Sendup);
                MPI::COMM_WORLD.Recv(&y(0, 0), Width, MPI::DOUBLE, myrank - 1, SendDown);
            }
        }
        else
        {
            // Bottom and the top row for current process's grid.
            MPI::COMM_WORLD.Send(&y(1, 0),   Width, MPI::DOUBLE,        myrank - 1, Sendup);
            MPI::COMM_WORLD.Recv(&y(Height - 1, 0), Width,MPI::DOUBLE,  myrank + 1, Sendup);
            MPI::COMM_WORLD.Send(&y(Height - 2, 0), Width, MPI::DOUBLE, myrank + 1, SendDown);
            MPI::COMM_WORLD.Recv(&y(0, 0),   Width, MPI::DOUBLE,        myrank - 1, SendDown);
        }
    }

}
```

And this is my modified Jacobi's method:

```
size_t jacobi(const mpiStencil& A, Grid& x, const Grid& b, size_t maxiter, double tol, bool debug = false)
{
    Grid y = b;
    swap(x, y); // x is now b, y is x
    // initial guess is b.
    for (size_t iter = 0; iter < maxiter; ++iter)
    {
        double rho = 0;
        for (size_t i = 1; i < x.num_x() - 1; ++i)
        {
            for (size_t j = 1; j < x.num_y() - 1; ++j)
            {
                y(i, j) = (x(i - 1, j) + x(i + 1, j) + x(i, j - 1) + x(i, j + 1)) / 4.0;
                rho += (y(i, j) - x(i, j)) * (y(i, j) - x(i, j)); // Delta
            }
        }
        double PartialRho = rho;
        // !! Accumulate Rho important reeeeee
        MPI::COMM_WORLD.Allreduce(&PartialRho, &rho, 1, MPI::DOUBLE, MPI::SUM);
        if (debug && MPI::COMM_WORLD.Get_rank() == 0)
        {
            std::cout << std::setw(4) << iter << ":␣";
            std::cout << "||r||␣=␣" << std::sqrt(rho) << std::endl;
        }
        if (std::sqrt(rho) < tol)
        {
            return iter;
        }
        swap(x, y); // x is x, y is b
        HaloUpdate(x);
    }
    return maxiter;
}
```

We need to update the error for the stopping conditions for the Jacobi's method, this is needed for stopping all the processes at the same time.

The boundaries for each of the processes are computed as the interior points of its neighbours.

This is done by a strictly synchronous appraoch, where, the top and the bottom grid partition only exchange it's boundary to one of it's neighbours, while all the internals grids are exchanging 2 ways of its neighbours.

Messages are categorized by tags, the sender and the receivers are sharing the same tag.

Question 5

What changes did you make for mpi_dot? Copy and paste relevant code lines that contain your edits to your report. Provide comments in the code near your edits to explain your approach.

For the dot product, we need to handle the following things:

1. Combine the partial sums from each of the process, and sync them up to all the processes.
2. Include the boundaries of the grid for computation as well, those values are part of the dot product.

Here is the code implementations for the grid dot product using MPI.

```
double mpi_dot(const Grid& X, const Grid& Y)
{
    assert(X.num_x() == Y.num_x() && X.num_y() == Y.num_y());

    double sum = 0.0;
    int Therank = MPI::COMM_WORLD.Get_rank();
    int Thesize = MPI::COMM_WORLD.Get_size();

    // Handle the top row and bottom row of process 0, rank - 1.
    bool IncludeTopBoundary = Therank == 0;
    bool IncludeBottomBoundary = Therank == Thesize - 1;
    size_t RowStart = IncludeTopBoundary? 0: 1;
    size_t RowEnd = IncludeBottomBoundary? X.num_x(): X.num_x() - 1;

    // Parallelize me
    for (size_t i = RowStart; i < RowEnd; ++i)
    {
        for (size_t j = 0; j < X.num_y(); ++j)
        {
            sum += X(i, j) * Y(i, j);
        }
    }
    double PartialSum = sum;
    MPI::COMM_WORLD.Allreduce(&PartialSum, &sum, 1, MPI::DOUBLE, MPI::SUM);

    return sum;
}
```

Question 6

What changes did you make for ir in mpiMath.hpp? Copy and paste relevant code lines that contain your edits to your report. Provide comments in the code near your edits to explain your approach.

For the iterative refinement, this is a list of things I do:

1. Use the MPI dot method inside the Iterative Refinement method.
2. Change the stencil muplication algorithm so that it compute the matrix vector product correctly, with the boundary sync up before computing things.

```
size_t ir(const mpiStencil& A, Grid& x, const Grid& b, size_t max_iter, double tol, bool debug = false) {
    for (size_t iter = 0; iter < max_iter; ++iter) {
        Grid r = b - A*x;

        double sigma = mpi_dot(r, r); // Already Communicated with other process.

        if (debug && MPI::COMM_WORLD.Get_rank() == 0) {
            std::cout << std::setw(4) << iter << ":␣";
            std::cout << "||r||␣=␣" << std::sqrt(sigma) << std::endl;
        }

        if (std::sqrt(sigma) < tol)
        {
            return iter;
```

```
        }
        x += r;

    }
    return max_iter;
}
void mult(const mpiStencil& A, const Grid& x, Grid& y)
{
    // !! Write me Ghost cell (halo) update goes here
    // !! Boundaries are row 0 and row x.num_x() - 1
    // Perform Stencil operations.

    // Sync up the boundary for each of the partition using the copied value of X grid.
    Grid Xcpy = x;
    HaloUpdate(Xcpy);
    for (size_t i = 1; i < x.num_x() - 1; ++i)
    {
        for (size_t j = 1; j < x.num_y() - 1; ++j)
        {
            y(i, j) = Xcpy(i, j) - (Xcpy(i - 1, j) + Xcpy(i + 1, j) + Xcpy(i, j - 1) + Xcpy(i, j + 1)) / 4.0;
        }
    }

}
```

This only thing I need to change is "mpi_dot(r, r)". It was origibally "dot(r, r)". Using the MPI version to compute the results will sync up the dot product for all the processes, so they have the right results to terminate at the right time.

Question 7

(583 only) What changes did you make for cg in mpiMath.hpp? Copy and paste relevant code lines that contain your edits to your report. Provide comments in the code near your edits to explain your approach.

This is a list of things I did for the Conjugate Gradient method:

1. I change the "dot" method into "mpi_dot". So that it's using the correct implementations with the MPI message board. Hence, the terminating conditions for CG and the compuations are all synced and correct.

The functions for dot and matrix vector multiplications are already shown before.

Here is the code for Conjugate Gradient:

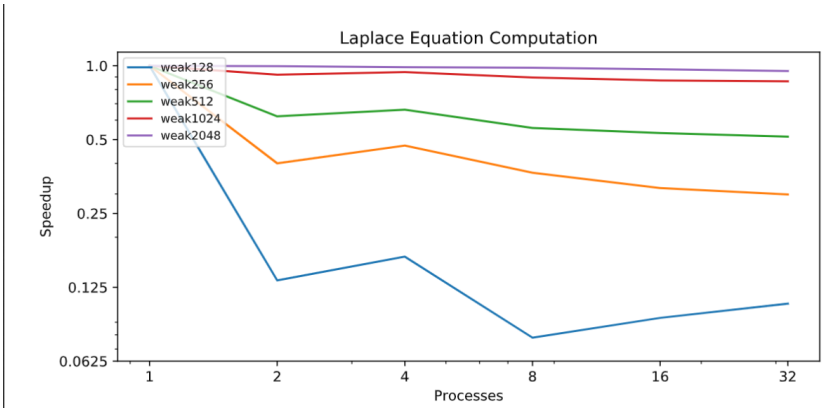
```
size_t cg(const mpiStencil& A, Grid& x, const Grid& b, size_t max_iter, double tol, bool debug = false) {
    size_t myrank = MPI::COMM_WORLD.Get_rank();

    Grid r = b - A*x, p(b);
    double rho = mpi_dot(r, r), rho_1 = 0.0; // Already communicated with other processes.

    for (size_t iter = 0; iter < max_iter; ++iter) {
        if (debug && 0 == myrank)
        {
            std::cout << std::setw(4) << iter << ":␣";
            std::cout << "||r||␣=␣" << std::sqrt(rho) << std::endl;
        }

        if (iter == 0)
        {
            p = r;
        } else {
            double beta = (rho / rho_1);
            p = r +  beta * p;

        }
        Grid q = A*p;
        double alpha = rho / mpi_dot(p, q);

        x += alpha * p;
        rho_1 = rho;
        r -= alpha * q;

        rho = mpi_dot(r, r);

        if (rho < tol) return iter;
    }

    return max_iter;
}
```

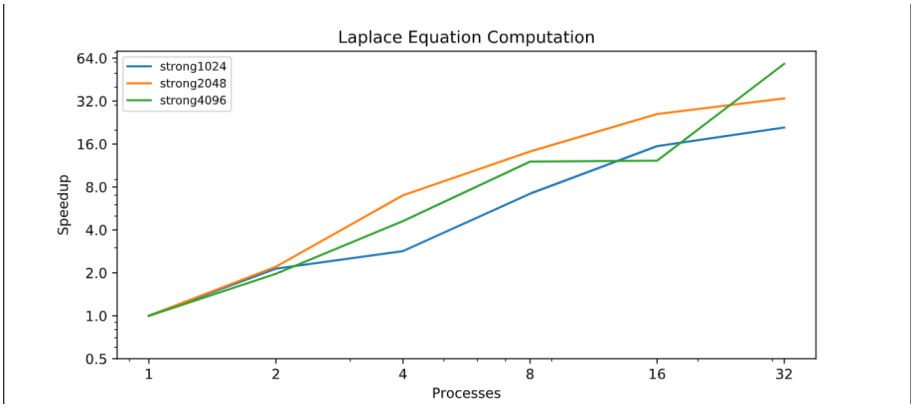
6.3.2 Scaling

Question 8

Per our discussions in lectures past about weak vs strong scaling, do the plots look like what you would expect? Describe any (significant) differences (if any).

This is the plots for weak and string scaling for running the Grid Executable:





The strong scaling has a speed up linear to the number of compute nodes and the weak scaling speed up stays approximating the same for higher number of iterations, and it goes down for smaller number of iterations.

This is expected as for the case of weak scaling, the problem size and the number of nodes increase together. Larger and larger problem size will result in more communications overhead. That is indicated as decrease in performance for fewer number of iterations.

The amount of communcations between processes grows together with the problem size, and therefore, there is no speed up in the weak scaling and it obeys Amdahl's law.

Question 9

For strong scaling, at what problem size (and what number of nodes) does parallelization stop being useful? Explain.

It's still scaling, I think we can add more nodes, all the way to 1024, maybe that will display the asymptoptic growth.

To improve the scaling, we need to consider using tree structure for comunications between processes, then the amount of sequential communcations time is growing like $\log(n)$ where n is the problem size