Rohan Tiwari
AMATH 583
PS8

1. Norm

1.1 What changes did you make for mpi_norm? 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 modified version of mpi_norm:

```
double mpi_norm(const Vector& local_x) {

  double global_rho = 0.0;


  double mysum = std::inner_product(&local_x(0),
&local_x(0)+local_x.num_rows(), &local_x(0), 0.0);

  // this is all reduce

  // this ensures the global summed value exists at all nodes. This
makes every node get the same final rho.

  MPI::COMM_WORLD.Allreduce(&mysum, &global_rho, 1, MPI::DOUBLE,
MPI::SUM);

  return std::sqrt(global_rho);

}
```

The allreduce will sum and send the result to all nodes so the have the same final answer.

In main function this code was added

```
// scatter contents of x only for node 0 into local_x on all ranks.

  if(myrank == 0){

    MPI::COMM_WORLD.Scatter(&x(0), num_elements, MPI::DOUBLE,
&local_x(0), num_elements, MPI::DOUBLE, 0);

  }

  else{

    MPI::COMM_WORLD.Scatter(nullptr, num_elements, MPI::DOUBLE,
&local_x(0), num_elements, MPI::DOUBLE, 0);
```

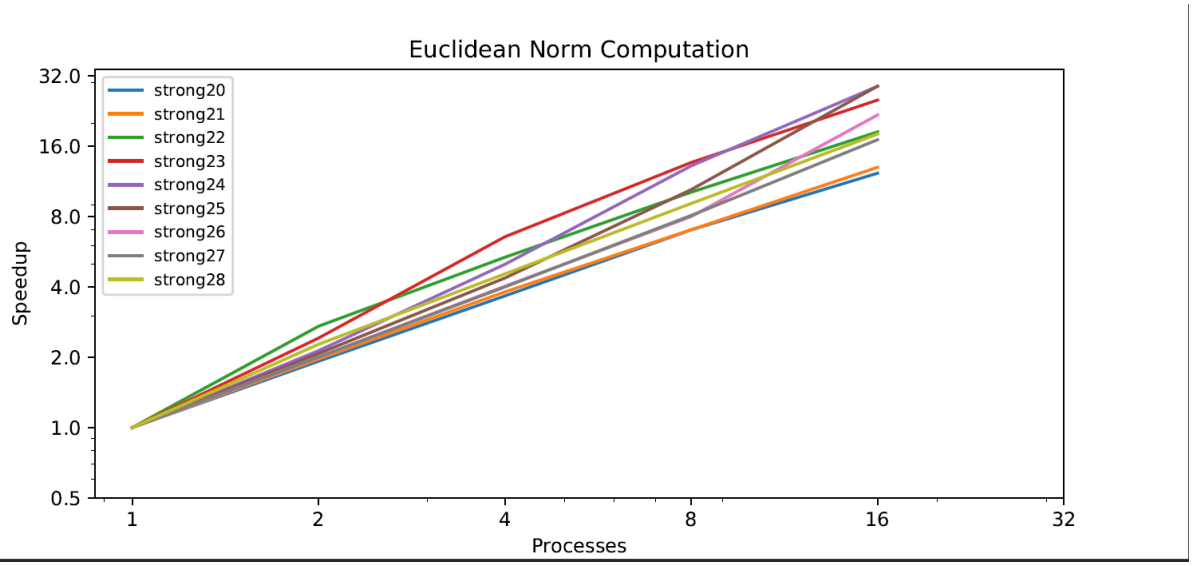
The above code scatters the data to all nodes. Num_elements is the size of the problem sent to each node.

```
  }
```

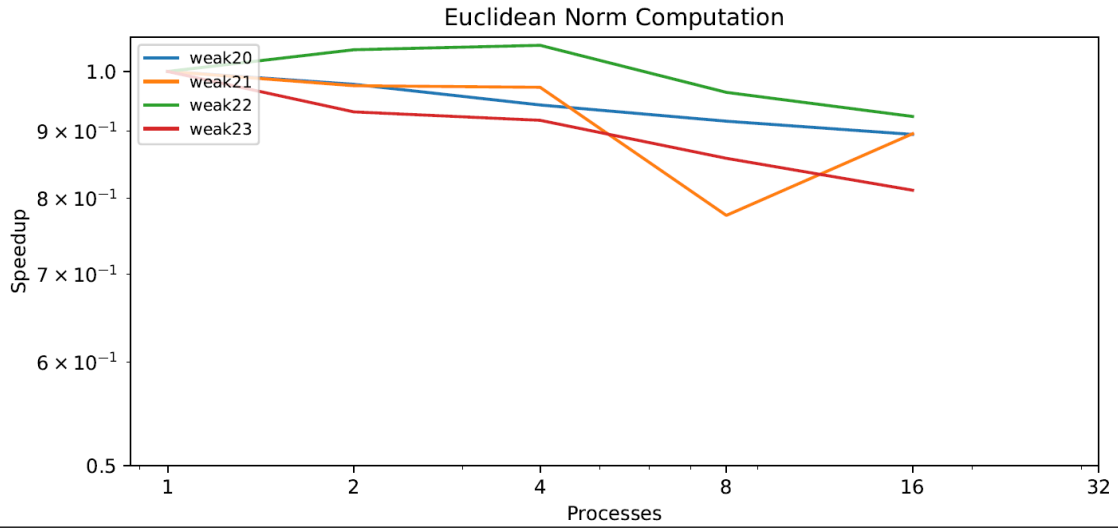
Rohan Tiwari
AMATH 583
PS8

1.2 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).

Strong scaling plot:



Weak scaling plot:



Graphs look as expected.

Strong scaling plot shows that speedup increases linearly with the number of processes while for weak scaling as number of processes is increased no speedup is observed (slowdown is observed).

For weak scaling the problem size is not increased proportionately to the number of processes, so the communication cost tends to be high adversely affecting the speedup (in fact we start seeing a slowdown). For strong scaling as the problem size is increased, there is incremental work for each additional process to do which hides the communication costs. Increasing problem size along with number of processes for strong scaling also limits the effect of sequential part of computation on overall speedup and we continue seeing speedup as the number of processes is increased. For Strong scaling, the speed up is slowly capped due to the sequential part of the problem which cannot be parallelized.

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

There is no upper limit observed as the lines keep going upward mostly. It looks like as number of processes is increased the speedup keeps growing. This indicates that there is enough work for each additional parallel process to take on and communication overhead doesn't slowdown things. This also means that the serial part isn't capping speed up yet (at least in the strong scaling plot above)

2. Laplace's equation

2.1 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. If you used a different scheme for extra credit in mult, show that as well.

Given below is the modified jacobi method in mpimath.hpp. Comments are added to the relevant lines.

Jacobi:

```
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);


  for (size_t iter = 0; iter < maxiter; ++iter) {


    // Parallelize me (rho)

    size_t myrank = MPI::COMM_WORLD.Get_rank();

    size_t mysize = MPI::COMM_WORLD.Get_size();


    double rho = 0.0;
```

```
    // each process will operate on parts of the overall grid
    //create empty vectors for boundary cases
    std::vector<double> buffer1(x.num_y(), 0.0);
    std::vector<double> buffer2(x.num_y(), 0.0);

    // check if this is not the last node to avoid going beyond the
lower boundary
    if(myrank < mysize - 1){
      MPI::COMM_WORLD.Sendrecv(&y(x.num_x()-1,0), x.num_y(),
MPI::DOUBLE, myrank+1, 0,
      &buffer1[0], x.num_y(), MPI::DOUBLE, myrank+1, 0);
    }

    // check if this isn't the first node to avoid going beyond the
upper boundary
    if(myrank > 0){
      MPI::COMM_WORLD.Sendrecv(&y(0,0), x.num_y(), MPI::DOUBLE,
myrank-1, 0,
      &buffer2[0], x.num_y(), MPI::DOUBLE, myrank-1, 0);
    }

    // handle inner part of the grid first
    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));
      }
    }

    //handle first row separately
    size_t i = 0;
```

```
    for(size_t j = 1; j < x.num_y() - 1; ++j){
      y(i, j) = (buffer1[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));
    }

    // handle last row separately
    i = x.num_x() - 1; // last row
    for(size_t j = 1; j < x.num_y() - 1; ++j){
      y(i, j) = (x(i - 1, j) + buffer2[j] + x(i, j - 1) + x(i, j + 1))
/ 4.0;
          rho += (y(i, j) - x(i, j)) * (y(i, j) - x(i, j));
    }

   // now run an all reduce to sum up and get global sum
   double global_rho = 0.0;
   MPI::COMM_WORLD.Allreduce(&rho, &global_rho, 1, MPI::DOUBLE,
MPI::SUM);
   rho = global_rho; // just assign back

    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);
```

```
  }


  return maxiter;
}
```

2.2 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.

The modified mpi_dot version is given below. The line modified is highlighted.

```
double mpi_dot(const Grid& X, const Grid& Y) {
  double local_sum = 0.0;
  double sum = 0.0;
  // Parallelize me
  // ignore the psuedo boundaries
  for (size_t i = 1; i < X.num_x()-1; ++i) {
    for (size_t j = 0; j < X.num_y(); ++j) {
      local_sum += X(i, j) * Y(i, j);
    }
  }
  MPI::COMM_WORLD.Allreduce(&local_sum, &sum, 1, MPI::DOUBLE,
MPI::SUM);
  return sum;
}
```

2.3 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.

Modified ir method is given below with the changes highlighted. The dot product method call is changed to use mpi version – mpi_dot. This makes it use the faster/parallel mpi version.

```
// Parallelize me
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;
```

```
    // change to mpi dot
    double sigma = mpi_dot(r, r);

    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;
}
```

2.4 (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.

The changes to the code are highlighted below. The dot product method call is changed to use the mpi version. This makes it use the faster/parallel mpi version.

```
// Parallelize me
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);
  // change to mpi_dot
```

```
  double rho = mpi_dot(r, r), rho_1 = 0.0;

  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;
    // change to mpi_dot
    double alpha = rho / mpi_dot(p, q);

    x += alpha * p;

    rho_1 = rho;
    r -= alpha * q;
    rho = dot(r, r);

    if (rho < tol) return iter;
  }

  return max_iter;
}
```

3. Extra Credit – Modified implementation of Mult

Given below is the modified scheme for mult. Instead of using the combined sendrecv, this uses separate send and receive routines.

```
void mult(const mpiStencil& A, const Grid& x, Grid& y){

  size_t myrank = MPI::COMM_WORLD.Get_rank();

  size_t mysize = MPI::COMM_WORLD.Get_size();


 if(mysize > 1){ // do not do for single node

    if(0 == myrank){

    // exchange last two rows

    MPI::COMM_WORLD.Recv(&y(x.num_x() - 1, 0), x.num_y(), MPI::DOUBLE,
myrank + 1, 1);

    MPI::COMM_WORLD.Send(&y(x.num_x() - 2, 0), x.num_y(), MPI::DOUBLE,
myrank + 1, 0);

    }

    else if(mysize - 1 == myrank){

    // exchange first two rows

      MPI::COMM_WORLD.Send(&y(1, 0), x.num_y(), MPI::DOUBLE, myrank -
1, 1);

      MPI::COMM_WORLD.Recv(&y(0, 0), x.num_y(), MPI::DOUBLE, myrank -
1, 0);

    }

    else{

      // exchange boundaries for this node

      // make sure message tags align correctly

      MPI::COMM_WORLD.Send(&y(1, 0),   x.num_y(), MPI::DOUBLE,
myrank - 1, 1);

      MPI::COMM_WORLD.Recv(&y(x.num_x() - 1, 0),
x.num_y(),MPI::DOUBLE,  myrank + 1, 1);

      MPI::COMM_WORLD.Send(&y(x.num_x() - 2, 0), x.num_y(),
MPI::DOUBLE, myrank + 1, 0);
```

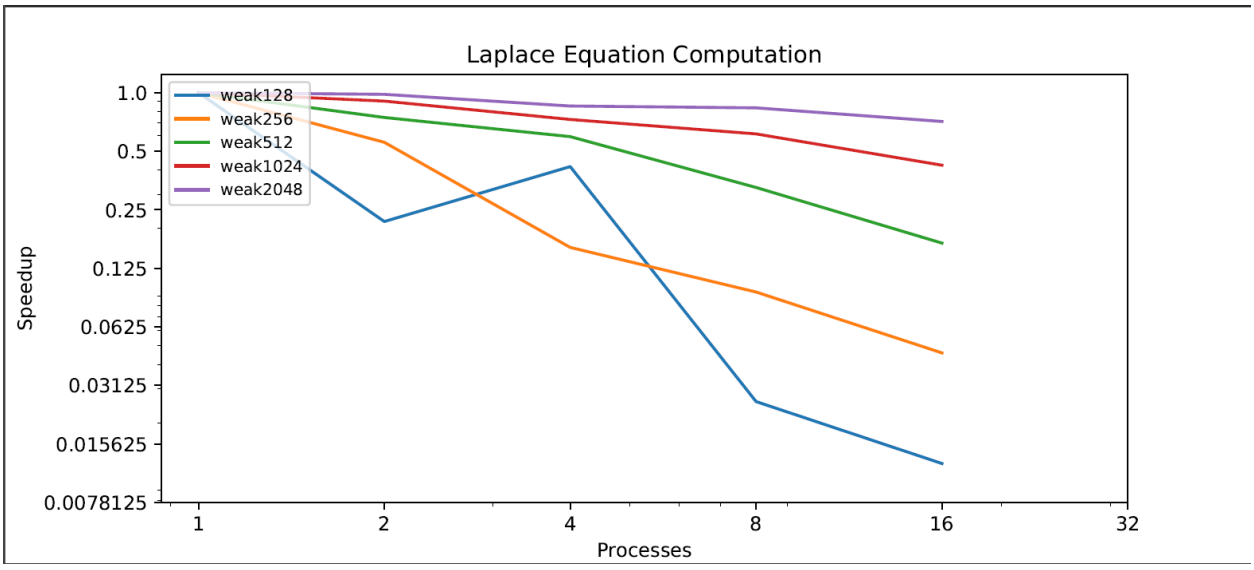
```
      MPI::COMM_WORLD.Recv(&y(0, 0),    x.num_y(), MPI::DOUBLE,
myrank - 1, 0);
    }
 }
 // stencil application
  for (size_t i = 1; i < x.num_x() - 1; ++i) {
      for (size_t j = 1; j < x.num_y() - 1; ++j) {
        y(i, j) = x(i, j) - (x(i - 1, j) + x(i + 1, j) + x(i, j - 1) +
x(i, j + 1)) / 4.0;
      }
  }

}
```

4. Scaling

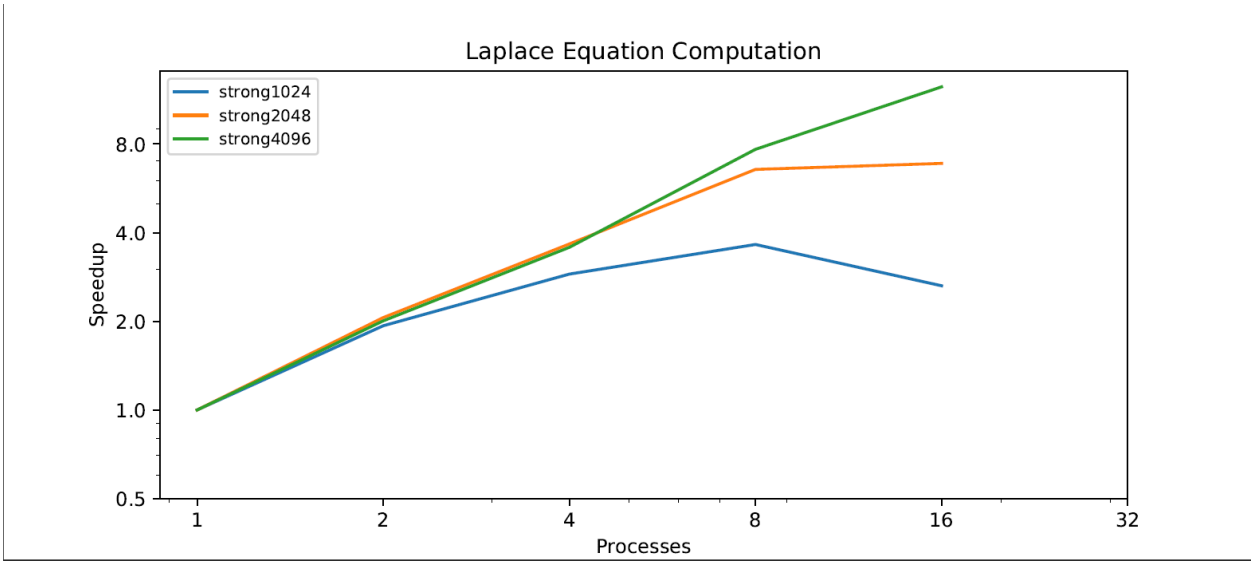
4.1 Per our discussions in lectures past about weak vs strong scaling, do the plots look like what you would expect? Describe any (significant) differences (if any).

Weak scaling plot:



For weak scaling, there is hardly any speedup implying that problem size is not scaled proportionately to number of processes and the communication costs between processes (as number of processes are increased) are high.

Strong Scaling plot:



For strong scaling, there is speedup observed until a certain number of processes after which there is a slowdown indicating the sequential part of the problem is capping the speedup which is as per Amdahl's law. Graphs look like what we would expect.

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

Looks like after 8 processes it stopped being useful. This is for all problem sizes but mainly for 2048 and 4096. It seems that after about 8 processes, the serial part starts to cap the speed up meaning there is not much incremental work to do for newly added parallel processes and hence no incremental speedup. It can be noticed that the speedup drops sharply for smaller sizes e.g. 1024 and not so much for larger sizes like 4096 which is understandable as smaller sizes are expected to not have enough work to give to the newly added parallel process hence communication costs might slow down parallel speedup.