PS8 Questions

To-Do 5 (ring.cpp):

```cpp
// Modification made in here, we set initial value of left and right to be 0:
// ----------------------------
int left  = 0;
int right = 0;
// if (0 == myrank) {
//   left  = 1;
//   right = 1;
// }
// ----------------------------

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

  // Modification made in here, we always try to send to the next process from our current process, and
  // when the current process is the last process, we let the next process to be the initial process so
  // that we have achieved the ring feature:
  // ----------------------------
  left = (myrank + 1) % mysize;
  // ----------------------------

  if (0 == myrank) {

    // Modification made in here, when current rank is 0, we try to receive from the last process
    // which is process (mysize - 1)
    // ----------------------------
    right = mysize - 1;
    // ----------------------------

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

    // Modification made in here, for processes other than the process 0, we try to always
    // receive from the current rank - 1:
    // ----------------------------
    right = myrank - 1;
    // ----------------------------

    MPI::COMM_WORLD.Recv(&token, 1, MPI::INT, right, 321);
    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)
  }
}
```

Norm

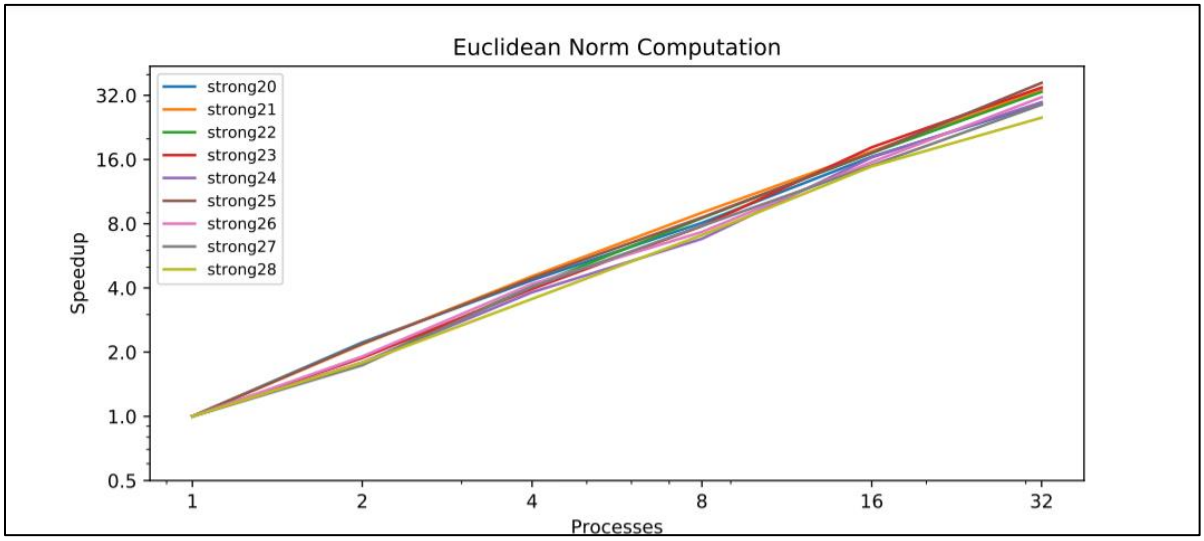
* Question 1: What is your code for `mpi_norm`? (Cut and paste the code here.)

```
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)

  // Modification made starting here:
  // ------------------------------------------------------------------------------
  // We set a double variable local_rho to be the norm for each partitioned local_x
  double local_rho = 0.0;
  for (size_t i = 0; i < local_x.num_rows(); ++i) {
    local_rho += local_x(i) * local_x(i);
  }
  // Then we use MPI Allreduce to sum up all the local_rho together as the global_rho
  MPI::COMM_WORLD.Allreduce(&local_rho, &global_rho, 1, MPI::DOUBLE, MPI::SUM);
  // ------------------------------------------------------------------------------

  return std::sqrt(global_rho);
}
```

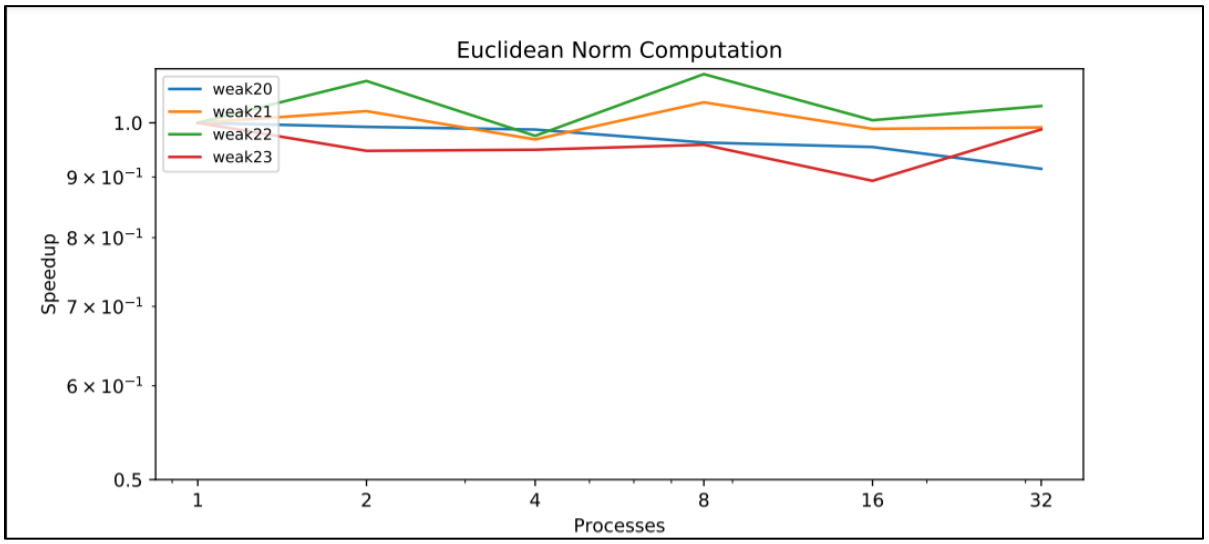
(I also made a modification where I used Scatter to distribute the vector in the main function):

```
  // Parallelize me -- the contents of vector x on rank 0 should be randomized and
  //scattered to local_x on all ranks
  Vector local_x(num_elements);

  // Modification made in here:
  // We use Scatter to distribute the vector.
  // ------------------------------------------------------------------------------------------------
  MPI::COMM_WORLD.Scatter(&x(0), num_elements, MPI::DOUBLE, &local_x(0), num_elements, MPI::DOUBLE, 0);
  // ------------------------------------------------------------------------------------------------
```

strong.pdf for mpi_norm.exe:



weak.pdf for mpi_norm.exe:



* 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).

Answer: For strong.pdf, the plot looks as expected. We see that as more processes added, the speedup we get is strongly scaled with the number of processes we have which is expected for strong scaling. For weak.pdf, the plot also looks as expected. We see that as more processes added, for all the problem sizes, the speedup is consistent around 1 which is expected for a weak scaling.

* Question 3: For strong scaling, at what problem size (and what number of nodes) does parallelization stop being useful? Explain.

Answer: From the plot we see that for all the problem sizes, the speedup scales almost consistently to the number of process added. This implies that adding more processes can be still useful in terms of performance speedup for all the problem sizes we tested by looking at the plot.

Grid

* Question 4: What is your code for halo exchange in `jacobi`? (Cut and paste the code here.) If you used a different scheme for extra credit in `mult`, show that as well.

```
size_t jacobi(const mpiStencil& A, Grid& x, const Grid& b, size_t maxiter, double tol, bool debug = false)
    ... (Code in here are omitted for space, only modified parts are shown)
    // Parallelize me (rho)
    double rho = 0.0;

    // Modification made in here:
    // Similar to mpi_norm, we set a local rho, then use Allreduce to sum up local_rho to rho:
    // ------------------------------------------------------------------------------
    double local_rho = 0.0;
    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;
        local_rho += (y(i, j) - x(i, j)) * (y(i, j) - x(i, j));
      }
    }
    MPI::COMM_WORLD.Allreduce(&local_rho, &rho, 1, MPI::DOUBLE, MPI::SUM);
    // ------------------------------------------------------------------------------

    ... (Code in here are omitted for space, only modified parts are shown)
    // Perform halo exchange (write me)

    // Modification made in here:
    // We are sending and receiving with corresponding x elements in a grid to update the ghost
    // cells just as depicted in the lecture notes. (with slide "Updating Ghost Cells" from Lecture 19.")
    // The approach we used in here is the first one on slide "Some other solutions" from Lecture 19:
    // ------------------------------------------------------------------------------------------------
    if (myrank != 0){
      MPI::COMM_WORLD.Send(&x(1, 0), x.num_y(), MPI::DOUBLE, myrank - 1, 123);
      MPI::COMM_WORLD.Recv(&x(0, 0), x.num_y(), MPI::DOUBLE, myrank - 1, 321);
    }
    if (myrank != mysize - 1){
      MPI::COMM_WORLD.Recv(&x(x.num_x() - 1, 0), x.num_y(), MPI::DOUBLE, myrank + 1, 123);
      MPI::COMM_WORLD.Send(&x(x.num_x() - 2, 0), x.num_y(), MPI::DOUBLE, myrank + 1, 321);
    }
    // ------------------------------------------------------------------------------------------------

  }
  return maxiter;
}
```

Q4 cont. (For **extra credit**, I have used a different approach for mult() in mpiStencil.hpp):

```
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();

  // Write me Ghost cell (halo) update goes here
  // Boundaries are row 0 and row x.num_x() - 1

  // Modification made in here:
  // Similar to what we did in jacobi function, the idea is the same, but for Extra Credit,
  // I used Irecv and Isend as a different approach. We are sending and receiving with
  // corresponding x elements in a grid to update the ghost cells just as depicted in the
  // lecture notes. (with slide "Updating Ghost Cells" from Lecture 19.")
  // ------------------------------------------------------------------------------------------
  if (myrank != 0) {
    MPI::Request recv_north = MPI::COMM_WORLD.Irecv(const_cast<double*>(&x(0, 0)),
                                                    x.num_y(), MPI::DOUBLE, myrank - 1, 321);
    MPI::Request send_north = MPI::COMM_WORLD.Isend(&x(1, 0), x.num_y(), MPI::DOUBLE, myrank - 1, 123);
    recv_north.Wait();
    send_north.Wait();
  }
  if (myrank != mysize-1) {
    MPI::Request recv_south = MPI::COMM_WORLD.Irecv(const_cast<double*>(&x(x.num_x() - 1, 0)),
                                                    x.num_y(), MPI::DOUBLE, myrank + 1, 123);
    MPI::Request send_south = MPI::COMM_WORLD.Isend(&x(x.num_x() - 2, 0), x.num_y(),
                                                    MPI::DOUBLE, myrank + 1, 321);
    recv_south.Wait();
    send_south.Wait();
  }
  // ------------------------------------------------------------------------------------------

  // SPMD stencil application
  ... (Code in here are omitted for space, only modified parts are shown)
}
```

* Question 5: What is your code for `mpi_dot`? (Cut and paste the code here.)

```
double mpi_dot(const Grid& X, const Grid& Y) {
  double sum = 0.0;

  // Modification made in here:
  // Depend on what rank we are at, we set the corresponding begin and end value to
  // deal with the proper boundary conditions when rank = 0 and rank = mysize - 1.
  // Then, like mpi_norm we set a local sum and use Allreduce to sum up for sum.
  // ------------------------------------------------------------------------------
  size_t myrank = MPI::COMM_WORLD.Get_rank();
  size_t mysize = MPI::COMM_WORLD.Get_size();
  size_t begin = (myrank == 0) ? 0 : 1;
  size_t end = (myrank == mysize - 1) ? X.num_x() : X.num_x() - 1;
  double local_sum = 0.0;
  for (size_t i = begin; i < end; ++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;
}
```

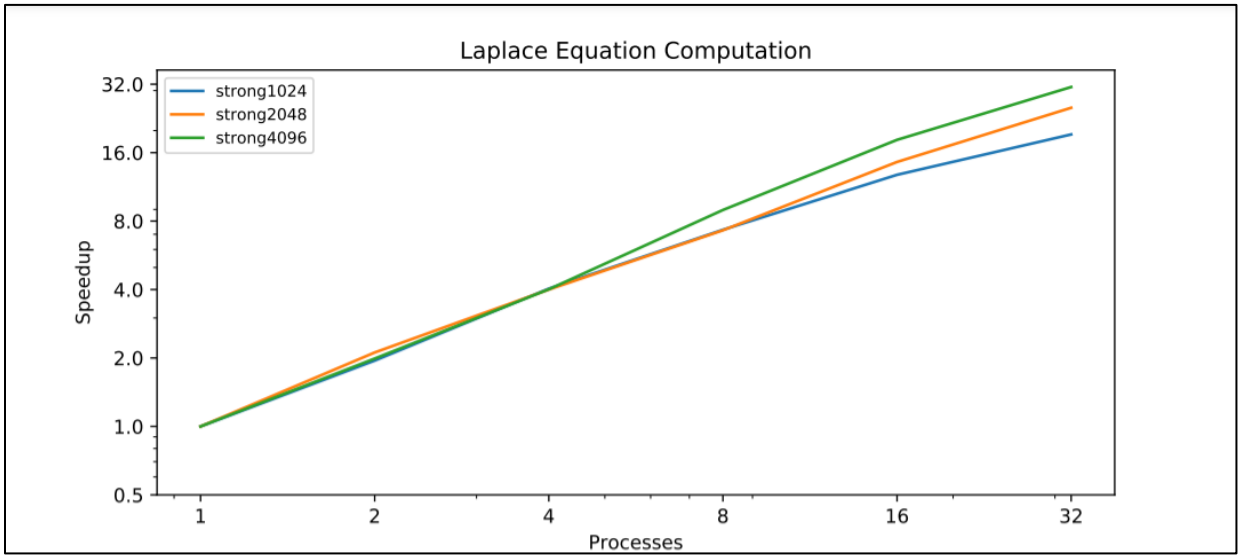
* Question 6: What changes did you make for `ir` in ` mpiMath.hpp `? Copy and paste relevant code lines that contain your edits to your report. Provide comments in the code near your edits to explain your approach.

```
// 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;

    // Modification made in here:
    // We change dot(r, r) to mpi_dot(r, r), since mpi_dot(r, r) utilized parallelization:
    // ---------------------------------------
    double sigma = mpi_dot(r, r);
    // ---------------------------------------

    ... (Code in here are omitted for space, only modified parts are shown)
  }
  return max_iter;
}
```

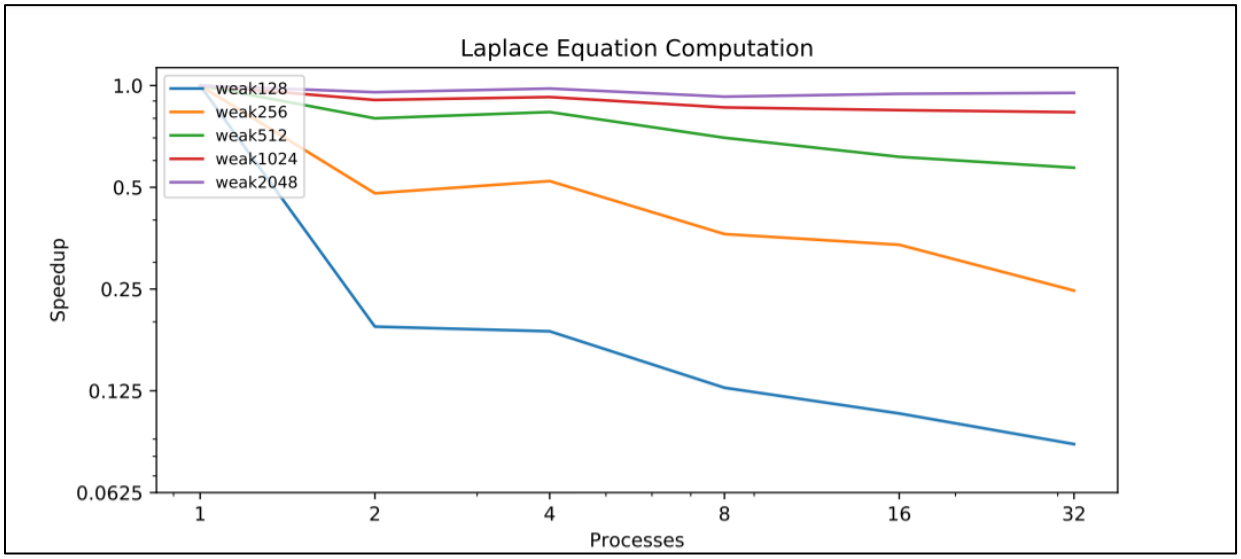
* Question 7: (583 only) What changes did you make for `cg` in `mpiMath.hpp`? Copy and paste relevant code lines that contain your edits to your report. Provide comments in the code near your edits to explain your approach.

Answer: This is for Amath 583 only, I am from Amath 483.

strong.pdf for grid.exe:



weak.pdf for grid.exe:



* 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).

Answer: For strong.pdf, the plot looks as expected. For strong scaling, we see that as more processes added, the speedup we get is strongly scaled with the number of processes we have which is expected for strong scaling. For weak.pdf, the plot also looks as expected. For weak scaling, we see that as the problem size increases, with more processes added, the speedup is more and more close and consistent to be 1 which is expected for weak scaling. For smaller problem sizes such as N = 128, as more processes added, we see the speedup dropped. Because the problem size is so small that the time took to add a new process surpasses the benefits of speedup brought by distributing the problem to pieces. This is reasonable and as expected.

* Question 9: For strong scaling, at what problem size (and what number of nodes) does parallelization stop being useful? Explain.

Answer: For problem size N = 1024, we see a turning point when the process number equals to 10, after that we see a drop in the speedup as more processes added. This implies that for N = 1024, after 10 processes, adding additional process stop being useful.

For problem size N = 2048, we see a turning point when the process number equals to 16, after that we see a drop in the speedup as more processes added. This implies that for N = 2048, after 16 processes, adding additional process stop being useful.

For problem size N = 4096, there is not an obvious turning point from the plot. We see that the speedup increased by adding more processes is generally consistent from 1 process to 32 processes. So, from what is provided by our plot, adding more than 32 processes for problem size N = 4096 can still be useful.