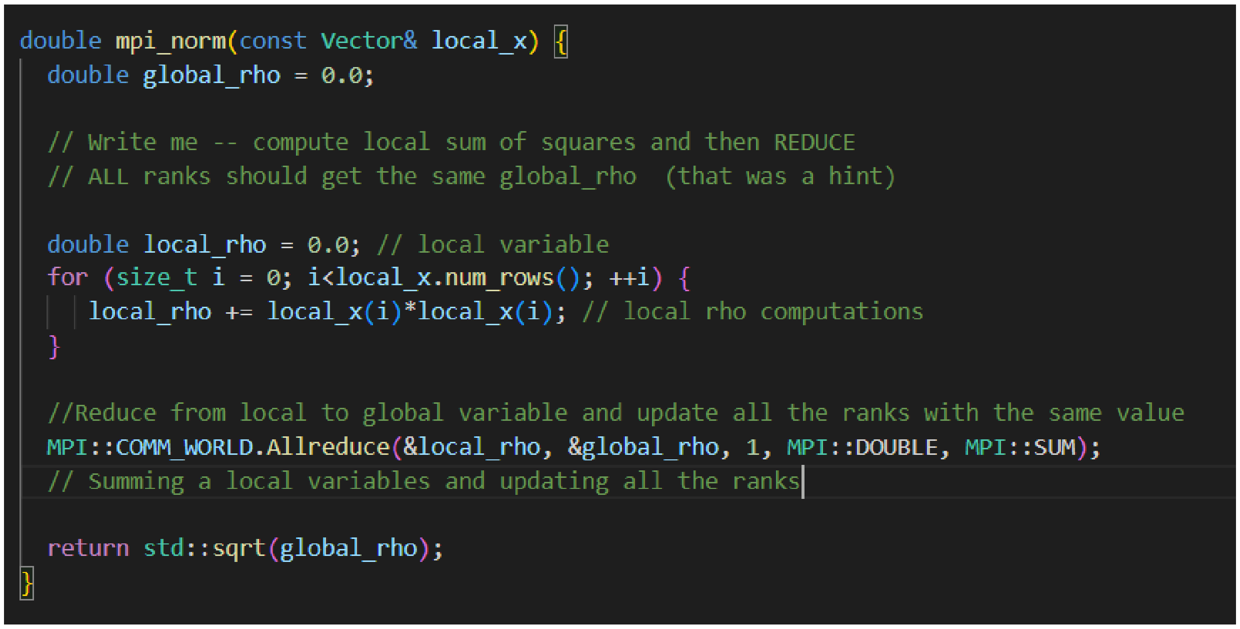
PS8 Questions

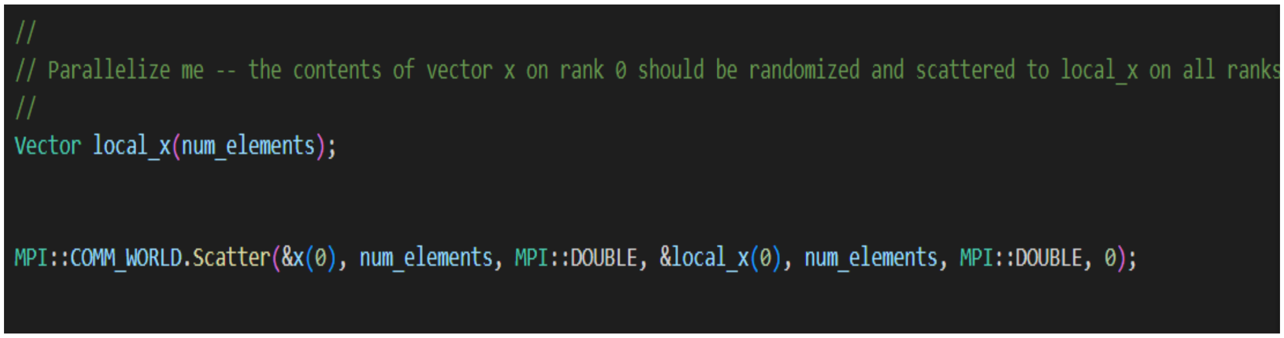
Norm

* 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.



```
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)

  double local_rho = 0.0; // local variable
  for (size_t i = 0; i<local_x.num_rows(); ++i) {
    local_rho += local_x(i)*local_x(i); // local rho computations
  }

  //Reduce from local to global variable and update all the ranks with the same value
  MPI::COMM_WORLD.Allreduce(&local_rho, &global_rho, 1, MPI::DOUBLE, MPI::SUM);
  // Summing a local variables and updating all the ranks

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

In the snippet below MPI::Scatter is used to split the global vector x to local vectors stored in ranks. The address of x and local_x is given by location of the first element of corresponding vector. That address is passed as input to scatter.

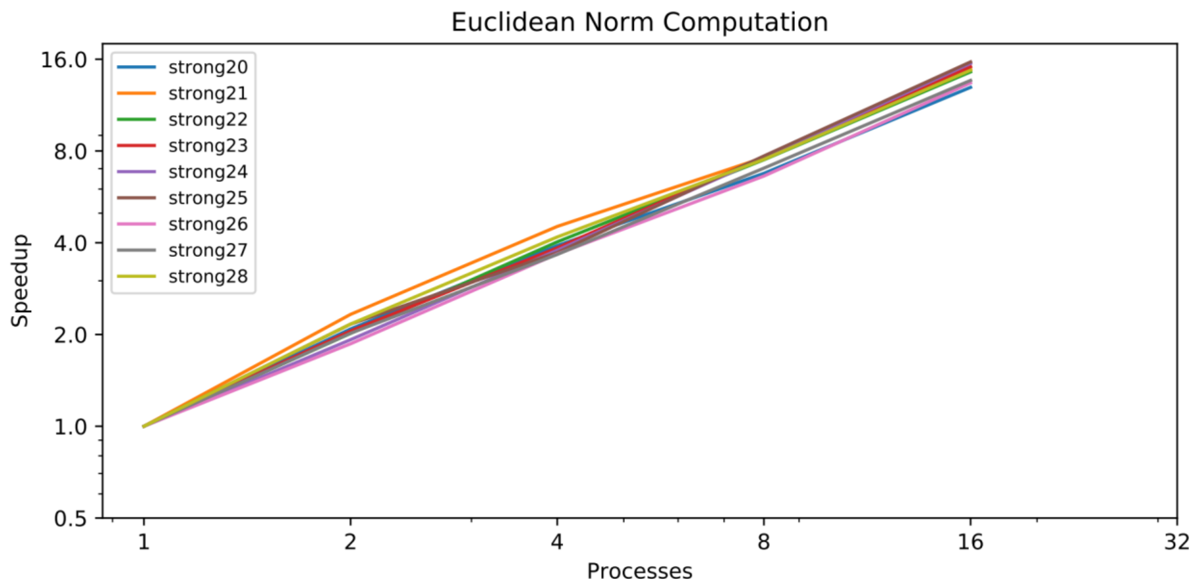


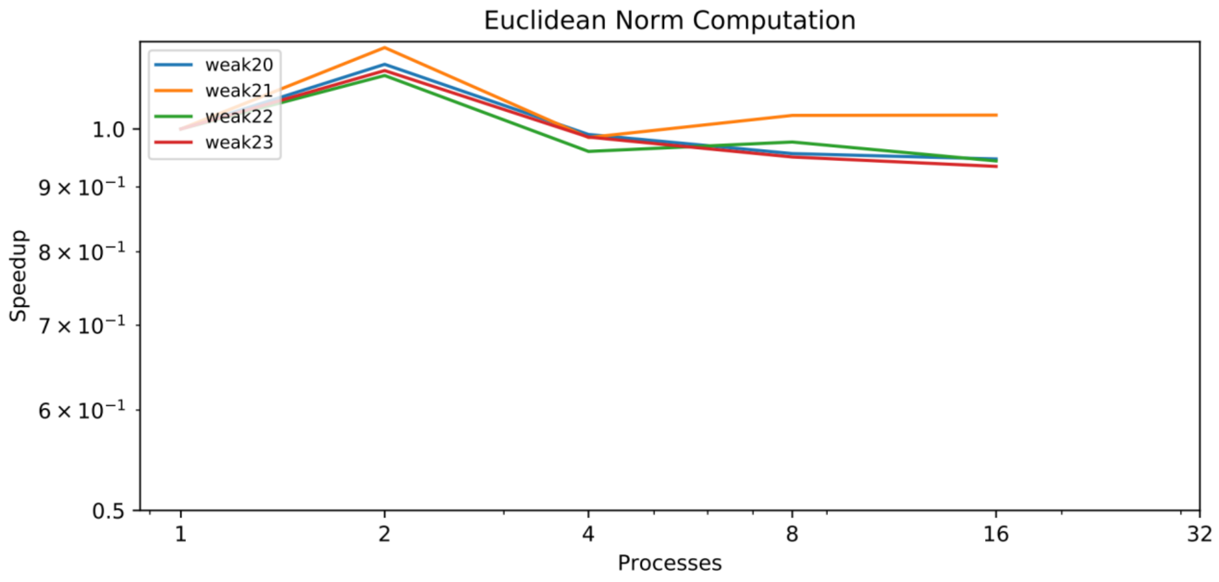
```
//
// Parallelize me -- the contents of vector x on rank 0 should be randomized and scattered to local_x on all ranks
//
Vector local_x(num_elements);


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

* 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 speed up is given by 1/(s+p/N)= N/(N*s+p), where s is serial part execution time, p is the parallelizable part execution time and N is the number of processes. Assuming s to be fixed, p should be inversely proportional to N (assuming latency can be hidden), then speed up can be written as 1(s+a/N^2), where a is some constant. We expect initial speed up with N, followed by plateauing.

Weak scaling speed up is given by s+p*N. Since p should be inversely proportional to N (assuming latency can be hidden), the speedup curve is expected to be a constant.



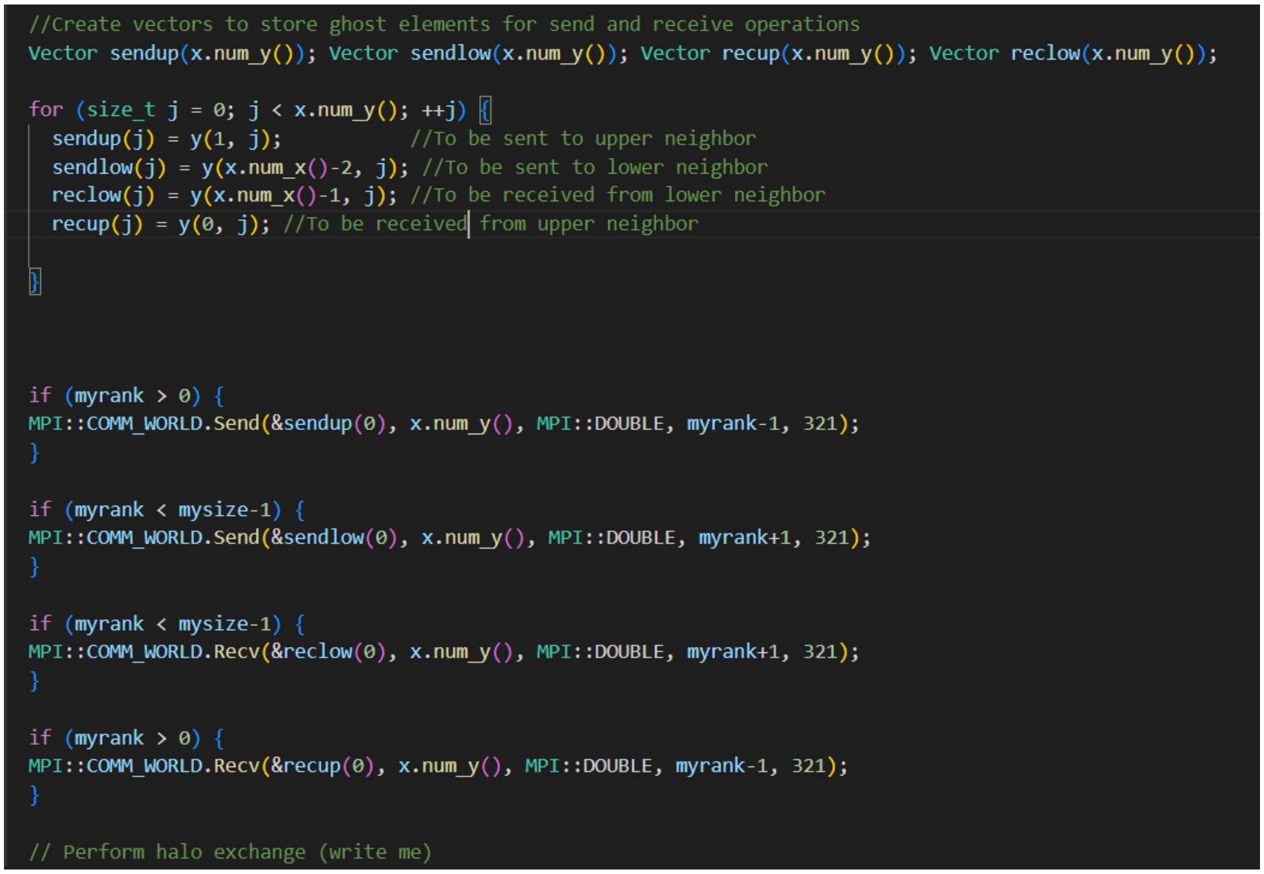


* For strong scaling, at what problem size (and what number of nodes) does parallelization stop being useful? Explain.

For problem size 27, and nodes 16, the performance drops significantly(more or less) compared to problem size 26. This could be because communicate time becomes much larger than compute time, and the latency can't be hidden any longer.

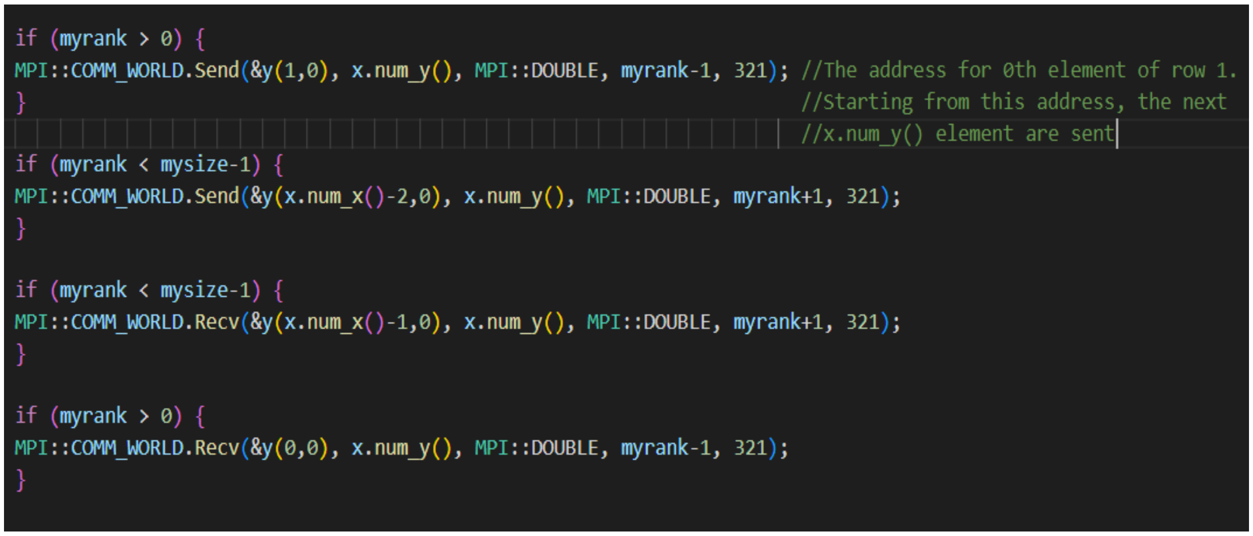
Solving Laplace's Equation
----

* 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.



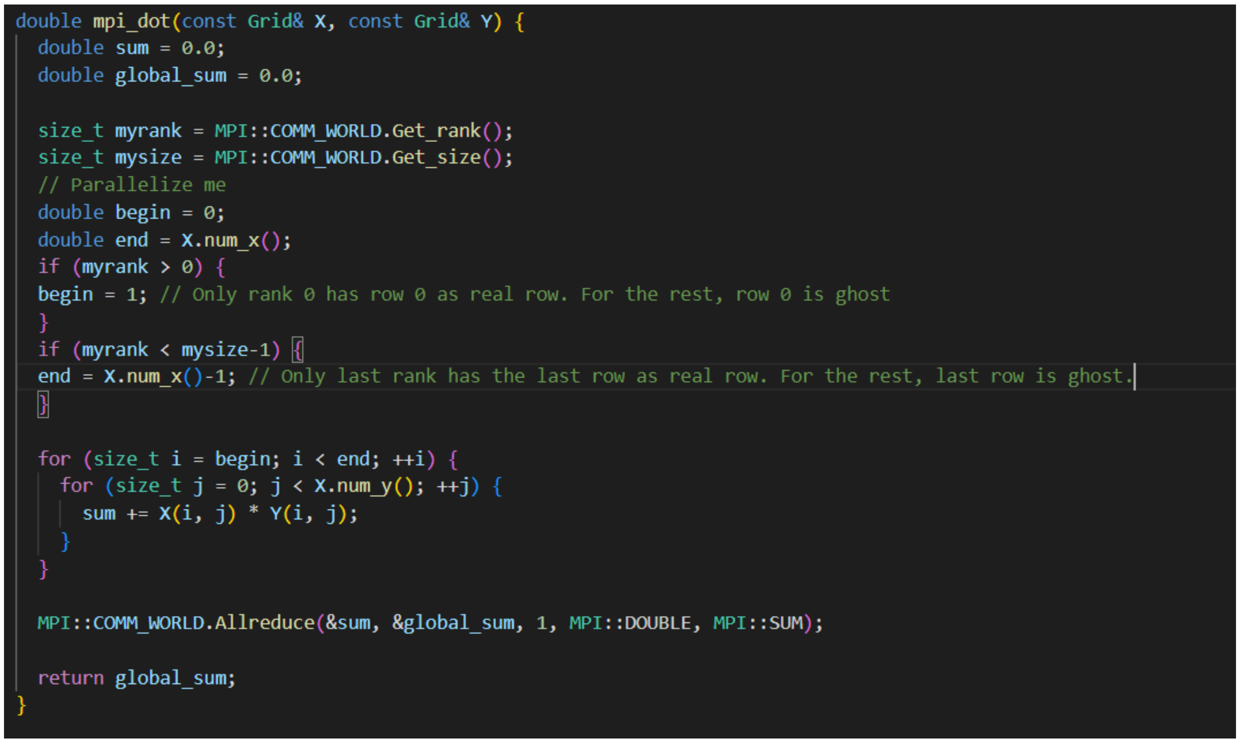
```
//Create vectors to store ghost elements for send and receive operations
Vector sendup(x.num_y()); Vector sendlow(x.num_y()); Vector recup(x.num_y()); Vector reclow(x.num_y());

for (size_t j = 0; j < x.num_y(); ++j) {
  sendup(j) = y(1, j);          //To be sent to upper neighbor
  sendlow(j) = y(x.num_x()-2, j); //To be sent to lower neighbor
  reclow(j) = y(x.num_x()-1, j); //To be received from lower neighbor
  recup(j) = y(0, j); //To be received from upper neighbor

}



if (myrank > 0) {
MPI::COMM_WORLD.Send(&sendup(0), x.num_y(), MPI::DOUBLE, myrank-1, 321);
}

if (myrank < mysize-1) {
MPI::COMM_WORLD.Send(&sendlow(0), x.num_y(), MPI::DOUBLE, myrank+1, 321);
}

if (myrank < mysize-1) {
MPI::COMM_WORLD.Recv(&reclow(0), x.num_y(), MPI::DOUBLE, myrank+1, 321);
}

if (myrank > 0) {
MPI::COMM_WORLD.Recv(&recup(0), x.num_y(), MPI::DOUBLE, myrank-1, 321);
}

// Perform halo exchange (write me)
```

We used the same scheme for mult as for jacobi.
Another, implementation for mult(shown below) we tested was directly sending elements from y, instead of creating Vectors to store the elements.



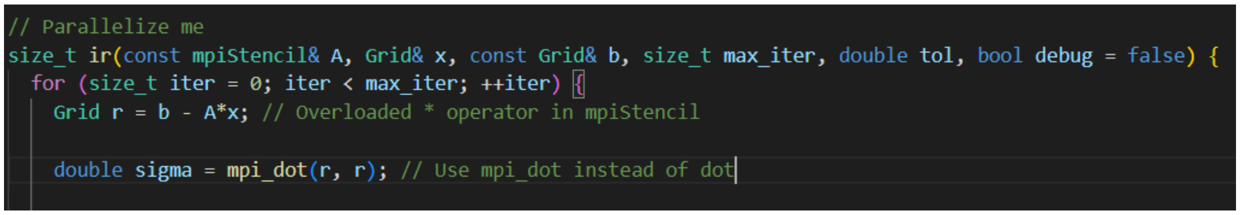
```
if (myrank > 0) {
MPI::COMM_WORLD.Send(&y(1,0), x.num_y(), MPI::DOUBLE, myrank-1, 321); //The address for 0th element of row 1.
}                                                                      //Starting from this address, the next
                                                                       //x.num_y() element are sent
if (myrank < mysize-1) {
MPI::COMM_WORLD.Send(&y(x.num_x()-2,0), x.num_y(), MPI::DOUBLE, myrank+1, 321);
}

if (myrank < mysize-1) {
MPI::COMM_WORLD.Recv(&y(x.num_x()-1,0), x.num_y(), MPI::DOUBLE, myrank+1, 321);
}

if (myrank > 0) {
MPI::COMM_WORLD.Recv(&y(0,0), x.num_y(), MPI::DOUBLE, myrank-1, 321);
}
```

* 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.



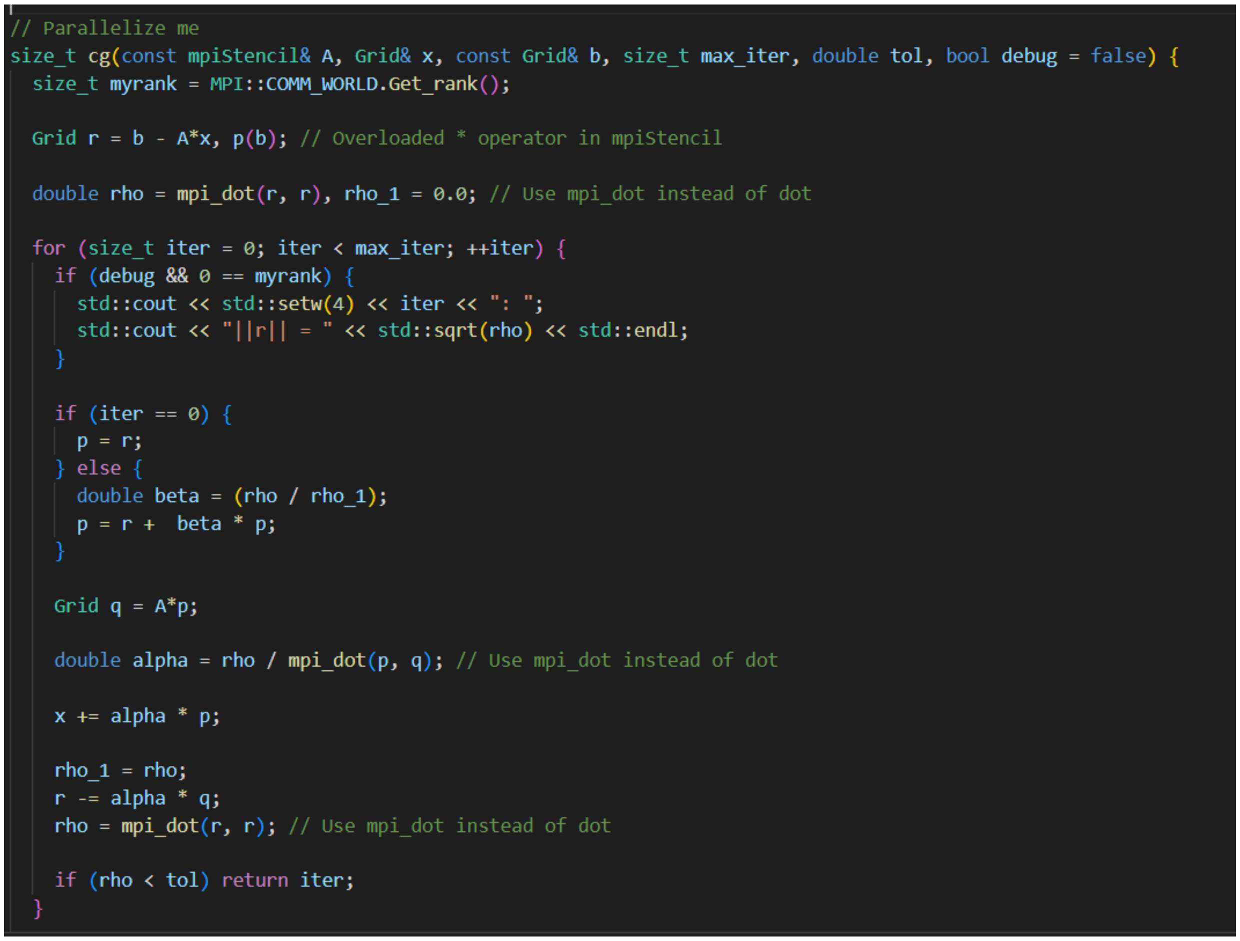
```
double mpi_dot(const Grid& X, const Grid& Y) {
  double sum = 0.0;
  double global_sum = 0.0;

  size_t myrank = MPI::COMM_WORLD.Get_rank();
  size_t mysize = MPI::COMM_WORLD.Get_size();
  // Parallelize me
  double begin = 0;
  double end = X.num_x();
  if (myrank > 0) {
  begin = 1; // Only rank 0 has row 0 as real row. For the rest, row 0 is ghost
  }
  if (myrank < mysize-1) {
  end = X.num_x()-1; // Only last rank has the last row as real row. For the rest, last row is ghost.
  }

  for (size_t i = begin; i < end; ++i) {
    for (size_t j = 0; j < X.num_y(); ++j) {
      sum += X(i, j) * Y(i, j);
    }
  }

  MPI::COMM_WORLD.Allreduce(&sum, &global_sum, 1, MPI::DOUBLE, MPI::SUM);

  return global_sum;
}
```

* 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; // Overloaded * operator in mpiStencil

    double sigma = mpi_dot(r, r); // Use mpi_dot instead of dot
```

* (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.



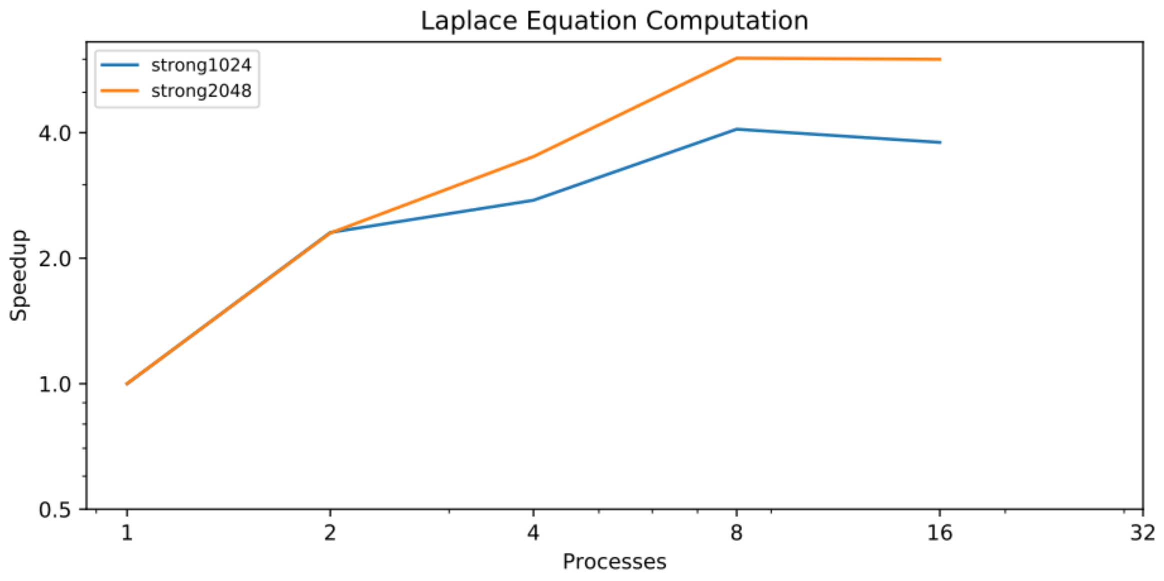
```cpp
// Parallelize me
size_t cg(const mpiStencil& A, Grid& x, const Grid& b, size_t max_iter, double tol, bool debug = false) {
  size_t myrank = MPI::COMM_WORLD.Get_rank();

  Grid r = b - A*x, p(b); // Overloaded * operator in mpiStencil

  double rho = mpi_dot(r, r), rho_1 = 0.0; // Use mpi_dot instead of dot

  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); // Use mpi_dot instead of dot

    x += alpha * p;

    rho_1 = rho;
    r -= alpha * q;
    rho = mpi_dot(r, r); // Use mpi_dot instead of dot

    if (rho < tol) return iter;
  }
```

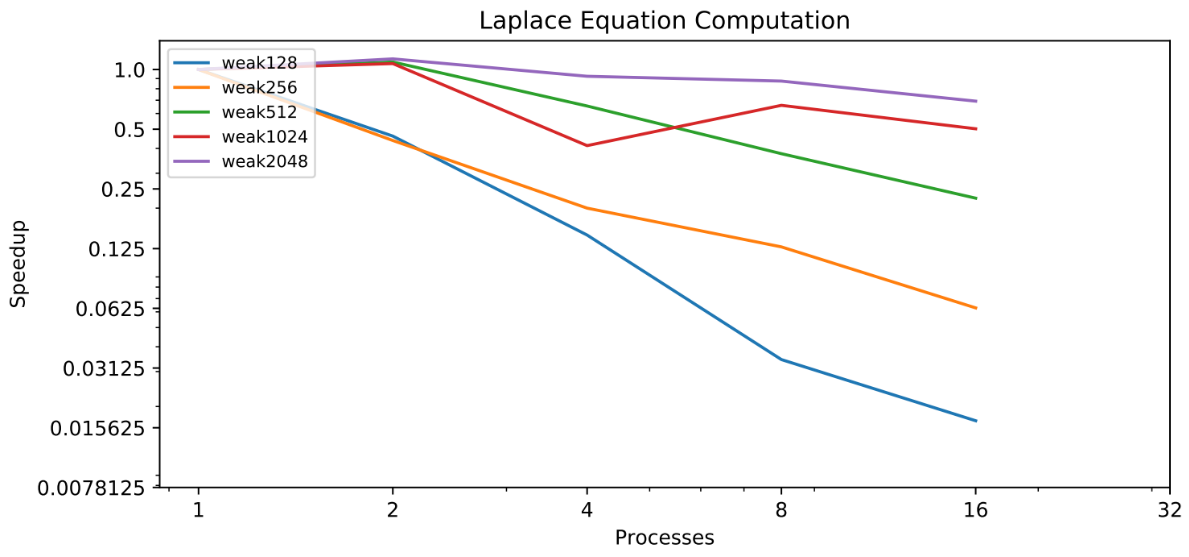
Scaling
----

* 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).

The strong and weak plots are shown below.
The strong.bash stopped showing time values for problem size 4096 and node 2 onwards. The table for 4096 is shown below.

size	procs	time
4096	1	25915
4096	2	-----
4096	4	-----
4096	8	-----
4096	16	-----





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

Since the code did not record time for 4096 and nodes greater than 2, it is possible that time to run was prohibitively long (>5:00) and was killed. The possible cause is large ghost cells. Sending and receiving large ghost cells, could be expensive and if communicate time >> compute time, then no speedup would be achieved over a single node, and might make the program slower.