PS8 Questions

Ben Francis

Ring

The homework said to copy and paste relevant changes to ring.cpp. The critical change was:

```
  int left  = myrank - 1;
  int right = myrank + 1;
  if (0 == myrank) {
    left  = 1;
    right = myrank - 1;
  }
  if (myrank == mysize - 1) {
    right = 0;
  }
```

Note that now we always have the left and right as the process to the left and right, except for 0 and mysize-1, which are the boundaries; in these cases
the neighboring value should "wrap around", where 0 wraps back to mysize-1 on its left and mysize-1 wraps around to 0 on its right.

Norm

QUESTION 1

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

Commentary: We used all reduce to make sure each process gets the same global rho.

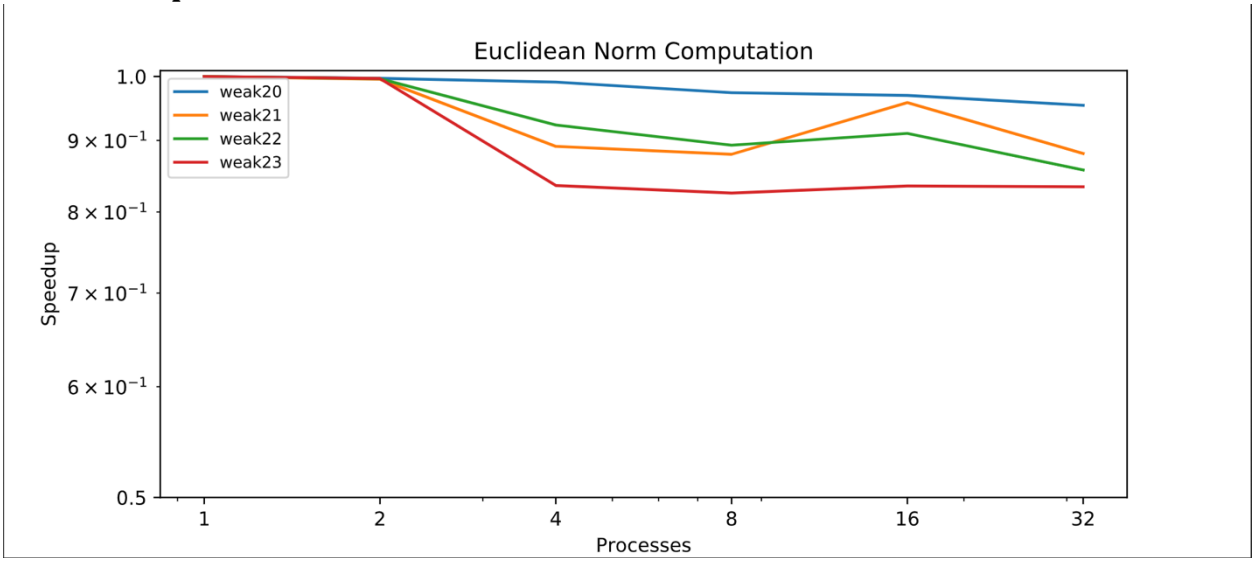
```
double mpi_norm(const Vector& local_x) {
  double global_rho = 0.0;

  // Write me -- compute local sum of squares and then REDUCE
  // ALL ranks should get the same global_rho  (that was a hint)

  double local_sum = 0.0;
  for (size_t i = 0; i < local_x.num_rows(); ++i) {
    local_sum += local_x(i) + local_x(i);
  }

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

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

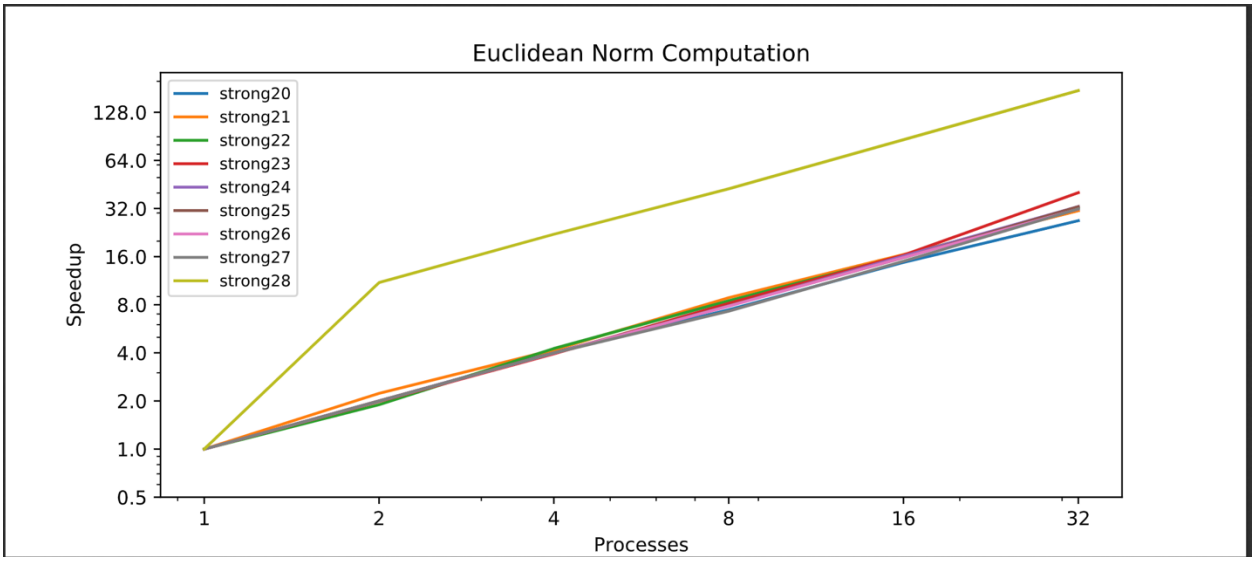
QUESTION 2

* 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).

Weak.bash plot



Strong.bash plot



Yes, these look like what we would expect. For the weak scaling, we are not increasing the problem size, and as such we do not see speedup with increased processes. Communication cost even incurs some losses. Flattening of performance is due to the intrinsically sequential part of the problem asymptotically approaching a certain percentage of the total computational cost For the strong, we do increase the data size, and so we correspondingly do see the (linear) speedup, since there is correspondingly more work for each process to do, so the percentage of the computation that is intrinsically sequential does not asymptotically increase.

QUESTION 3

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

There is no maximum useful size, as expected (good result for parallel computing!). This supports Gustafson's law. Similarly, the capping of speedup in the weak plot seems to support Ahmdahl's law; the difference, as expected, being the problem size.

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.

(Commentary in code)

```
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)
    double rho = 0;
    double global_rho = 0;

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

    // Number of rows in this sub-grid process
    size_t numRows = y.num_x();
    size_t numCols = y.num_y();
```

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

    // Buffers to receive boundary rows
    std::vector<double> prevRow(numCols, 0.0);
    std::vector<double> nextRow(numCols, 0.0);

    // Make sure it's not the top sub-grid process (no neighbor "on top" to swap with)
    if (myrank > 0) {
      MPI::COMM_WORLD.Sendrecv(&y(0,0), numCols, MPI::DOUBLE, myrank - 1, 0,
                               &prevRow[0], numCols, MPI::DOUBLE, myrank-1, 0);
    }

    // Make sure it's not the bottom sub-grid process (no neighbor "on bottom" to swap
with)
    if (myrank < mysize - 1) {
      MPI::COMM_WORLD.Sendrecv(&y(numRows-1,0),numCols, MPI::DOUBLE, myrank + 1, 0,
                               &nextRow[0], numCols, MPI::DOUBLE, myrank + 1, 0);
    }

    // First row (need to use prevRow)
    size_t i = 0;
    for (size_t j = 1; j < numCols - 1; ++j) {
      y(i, j) = (prevRow[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));
    }
    for (size_t i = 1; i < numRows - 1; ++i) {
      for (size_t j = 1; j < numCols - 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));
      }
    }
    // Last row (need to use nextRow)
    i = numRows - 1;
    for (size_t j = 1; j < numCols - 1; ++j) {
      y(i, j) = (x(i - 1, j) + nextRow[j] + x(i, j - 1) + x(i, j + 1)) / 4.0;
      rho += (y(i, j) - x(i, j)) * (y(i, j) - x(i, j));
    }

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


    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;
}
```

QUESTION 5

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

Commentary: Note the bounds for i, which do not to include the ghost cells on the row boundaries.

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

  // Parallelize me
  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, &global_sum, 1, MPI::DOUBLE, MPI::SUM);

  return global_sum;
}
```

QUESTION 6

* What is your code for `ir`? (Cut and paste the code here.)

Commentary: Replaced dot with mpi_dot. A will already use the parallelized mpiStencil multiply since it is an mpiStencil.

```
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);
```

```
    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;
}
```

QUESTION 7

* What is your code for `cg`? (Cut and paste the code here.)

Commentary: Replaced dot with mpi_dot, on two occasions. A will already use the parallelized mpiStencil multiply since it is an mpiStencil.

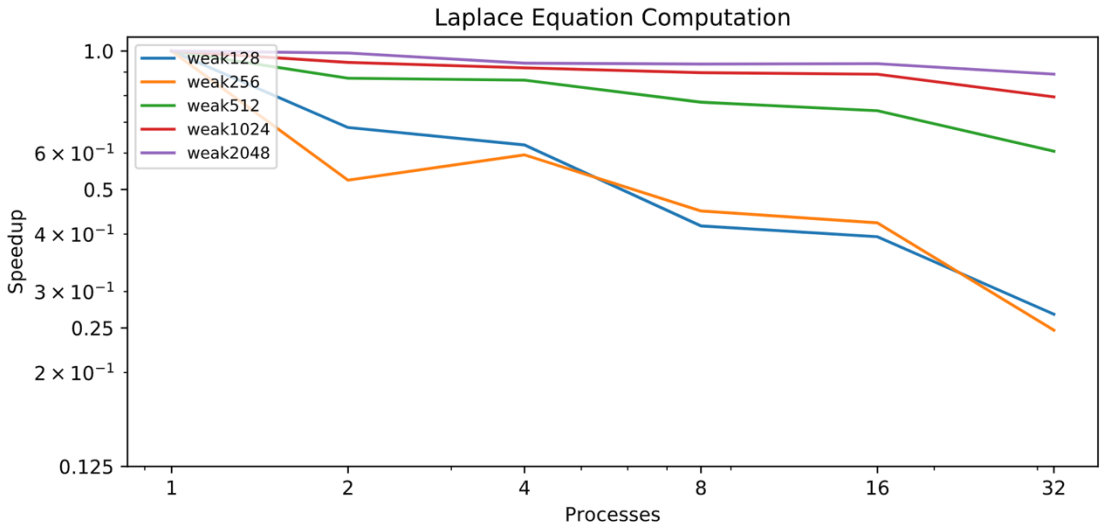
```
size_t cg(const mpiStencil& A, Grid& x, const Grid& b, size_t max_iter, double tol,
bool debug = false) {
  size_t myrank = MPI::COMM_WORLD.Get_rank();

  Grid r = b - A*x, p(b);
  double rho = mpi_dot(r, r), rho_1 = 0.0;

  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;
}
```

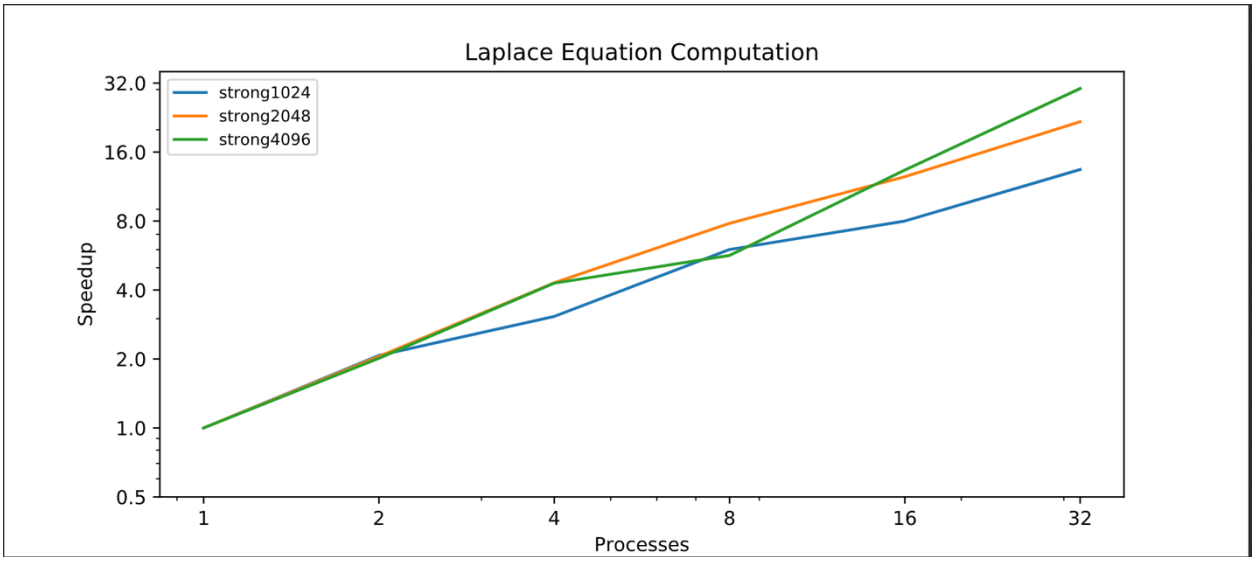
QUESTION 8

* 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).

Weak.bash plot



Strong.bash plot



The weak version doesn't scale at all, often being worse. This is because the problem size is not being scaled accordingly. Ahmdahl's law explains that the intrinsically sequential part of the program eventually caps all parallel benefit, even with additional processes. Furthermore, as seen here, communication problems may even become burdensome.

Yes, they look like what we would expect.

QUESTION 9

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

Again, there isn't a maximum where it stops being useful; the trend is linear throughout, which again is in accordance with Gustafson's law. The analysis is identical to the previous case with "norm".