**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**

Chart, line chart

Description automatically generated

**Strong.bash plot**

Chart, line chart

Description automatically generated

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**

Chart, line chart

Description automatically generated

**Strong.bash plot**

Chart, line chart

Description automatically generated

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”.