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.

}

**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:

Chart, line chart

Description automatically generated

Weak scaling plot:

Chart, line chart

Description automatically generated

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)

1. **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;

}

1. **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;

}

}

}

1. **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:

Chart, line chart

Description automatically generated

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:

Chart, line chart

Description automatically generated

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.