Partitioning the Problem

I arrange the processors into a grid (as close to square as I can get with side lengths 2^n), and I give each processor a chunk of matrix C to calculate on its own. This means each processor needs full rows from matrix A and full columns from matrix B. I scatter the rows of matrix A down column 0 of my processor grid, then I broadcast those rows down each row of the processor grid. Same with B, I scatter then broadcast. At the end I gather the parts of C along the rows and columns of the processor grid.

Communication APIs

I found that my implementation was still mostly bottlenecked by computation. At least with n=4096 and np=4, tweaking the communication did not have a large effect on performance. In theory using non blocking operations to send out A and B concurrently should help, but I only saw a change of about 2GOPS. That's pretty minor because I get swings of +/- 5GOPS just rerunning the same code without modification. Also, I originally didn't scatter the columns of B directly. I scattered the rows, did a sort of transpose, then did an all-to-all send to get columns. This worked but caused me a lot of headache and after some digging I found that MPI_Type_vector() is a godsend for traversing the columns of a row-major matrix. It even gave me a ~5GOPS boost. The version I decided on is pretty simple and uses only blocking operations. Since you ask for code snippets, here are my scatters, broadcasts, and gathers:

```
// scatter a vertically
if (my_col == 0) {
  MPI_Scatter(a, local_rows * kK, MPI_FLOAT, local_a, local_rows * kK,
MPI_FLOAT, 0, col_comm);
}
// broadcast a horizontally
MPI_Bcast(local_a, local_rows * kK, MPI_FLOAT, 0, row_comm);
// scatter b horizontally
if (my_row == 0) {
  MPI_Datatype temp, column;
  MPI_Type_vector(kK, local_cols, kJ, MPI_FLOAT, &temp);
  MPI_Type_create_resized(temp, 0, local_cols * sizeof(float), &column);
  MPI_Type_commit(&column);
  MPI_Scatter(b, 1, column, local_b, kK * local_cols, MPI_FLOAT, 0,
row_comm);
  MPI_Type_free(&temp);
  MPI_Type_free(&column);
}
// broadcast b vertically
```

```
MPI_Bcast(local_b, kK*local_cols, MPI_FLOAT, 0, col_comm);
// gather chunks of c into rows
MPI_Datatype temp, chunk;
MPI_Type_vector(local_rows, local_cols, kJ, MPI_FLOAT, &temp);
MPI_Type_create_resized(temp, 0, local_cols * sizeof(float), &chunk);
MPI_Type_commit(&chunk);
float *c_buffer = new float[local_rows * kJ];
MPI_Gather(local_c, local_rows*local_cols, MPI_FLOAT, c_buffer, 1, chunk,
0, row_comm);
if (my_col == 0) {
  // gather rows into full matrix
  MPI_Gather(c_buffer, local_rows * kJ, MPI_FLOAT, c, local_rows * kJ,
MPI_FLOAT, 0, col_comm);
}
MPI_Type_free(&temp);
MPI_Type_free(&chunk);
delete[] c_buffer;
```

Problem Size

All tested with np=4 on aws m5.2xlarge:

n=1024:
Time: 0.0335667 s
Perf: 63.9766 GFlops

n=2048:
Time: 0.132184 s
Perf: 129.97 GFlops

n=4096:
Time: 0.909129 s
Perf: 151.176 GFlops

Performance increases nicely with problem size. With a fixed number of processors, the performance will top out at some point. So although going 1024->2048 doubles performance that is not a sustainable trend as n gets large with fixed np. All we need is for performance (and thus efficiency) to not drop off.

Scalability

All tested with n=4096 on aws m5.2xlarge:

np=1:
Time: 2.76075 s
Perf: 49.7833 GFlops

np=2:
Time: 1.57101 s
Perf: 87.4847 GFlops

np=4:
Time: 0.896402 s
Perf: 153.323 GFlops

np=8:
Time: 0.873159 s
Perf: 157.404 GFlops

np=16:
Time: 1.71505 s
Perf: 80.1371 GFlops

np=32:
Time: 3.08093 s
Perf: 44.6096 GFlops

Going 1->2->4 gives expected significant increases in performance. Going 4->8 does not increase performance, and I attribute that to the same thing I talked about in my report for lab 1; the m5.2xlarge may be able to do 8 programs in parallel but there are only 4 caches, which is crucial for tasks like this which revolve around memory usage. At least it doesn't slow down though. Then for 8->16->32 it takes considerable performance hits. This is expected because all my MPI operations are blocking. One process cannot proceed until it is done with a data transfer, which can't happen until another process gets swapped in. The fact that only 8 processes can run concurrently means that there will be lots of expensive waiting and context switches.

Note about performance: On aws I'm getting about 150GOPS for n=4096, np=4. However on gradescope it shoots up to about 200GOPS. Not sure why this happens. So for grading performance please use gradescope's numbers.

Compartison to OpenMP

Programming with OpenMP was definitely easier. I’m turning this in late because I had a rough time figuring out some segfaults. Managing memory transfers is a pain because it adds several new parameters to mess up. With OpenMP I could just have all the threads access the same shared memory. Also there is just more work to do. My MPI implementation is 2.5 times as long as the OpenMP one. With OpenMP I slapped a couple pragmas onto the sequential program and it worked spectacularly. MPI does have better performance for 4 processes/threads though. It doesn’t fare as well with more processes (OpenMP had ~200GOPS at 8 threads) but I think with bigger problems/computers it would be clear that MPI scales better.