// calculate all ranges

int stripe = size / mpi\_size; // general stripe size

int remainder = size % mpi\_size; // in case if size is not divisible

int stripes[mpi\_size]; // stripes[i] is rank i’s stripe

int stripe\_begins[mpi\_size]; // stripe\_begins[i] is rank i’s

int stripe\_ends[mpi\_size]; // stripe\_begins[i] is rank i’s

// once all ranks receive “size”, i.e., a simulation size, they can compute

// their stripe\_begins[my\_rank] and stripe\_ends[my\_rank].

stripes[my\_rank] = ...;

stripe\_begins[my\_rank] = ( rank < remainder ) ?

stripe \* my\_rank + rank : stripe \* rank + remainder;

stripe\_ends[my\_rank] = ...;

// for simplicity, you may use different variables like:

int stripe\_begin = stripe\_begins[my\_rank];

int stripe\_end = stripe\_ends[my\_rank];

// then, your schroedinger computation will be

for( int i = stripe\_begin; i <= stripe\_ends; i++ ) {

for ( int j = 1; j < size - 1; j++ ) {

...;

}

}

// before the above schroedinger computation, you need to exchange boundary

// arrays with left and right neighbors.

MPI\_Send( z[q][stripe\_begin], size, MPI\_DOUBLE, my\_rank - 1, 0,

MPI\_COMM\_WORLD ); // send to my left neighbor

MPI\_Send( z[q][stripe\_end], size, MPI\_DOUBLE, my\_rank + 1, 0,

MPI\_COMM\_WORLD ); // send to my right neighbor

// note that rank 0 has no left neighbor and rank n-1 has no right

// neighbor.

// Finally, recall that, if two processes send a big message to each other

// at a time, they may exhaust each other’s socket buffer, which falls into

// a deadlock. Let even ranks initiate sends first so that odd ranks receive

// messages. Thereafter, they should change their turns.