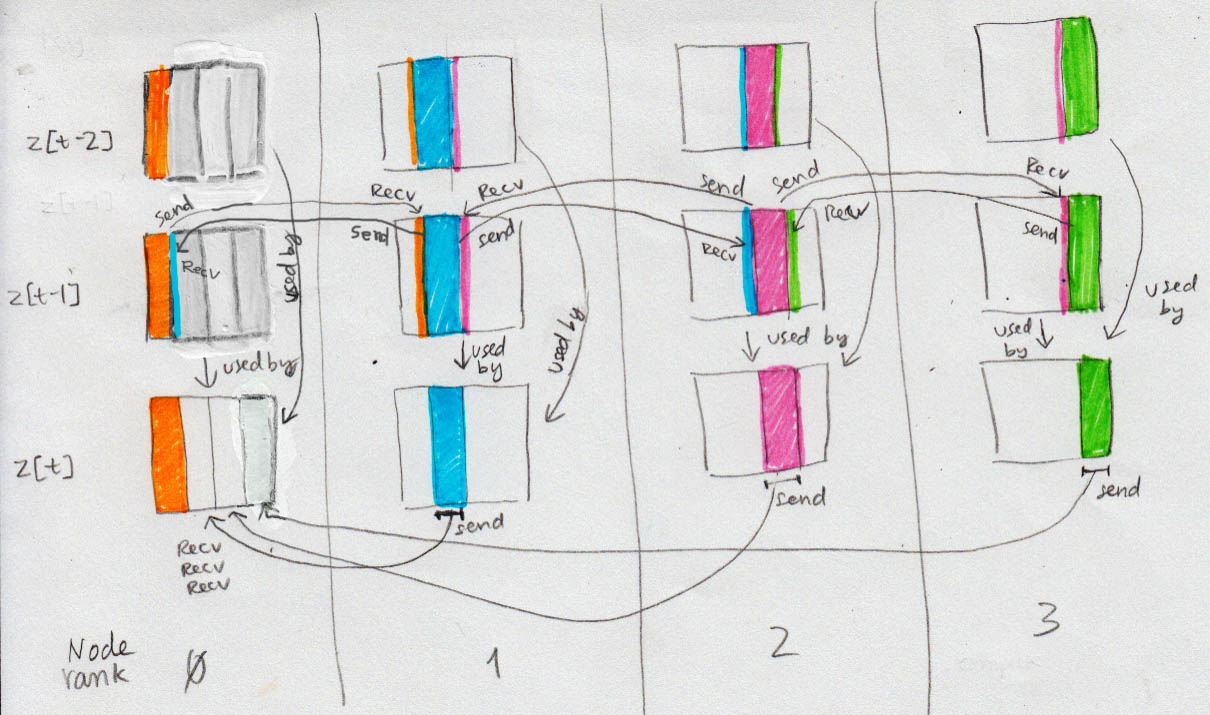
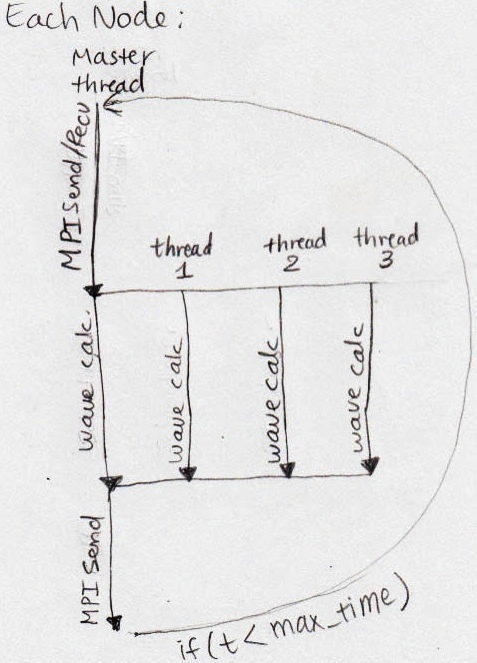
Documentation

Parallelization strategy

Example of what 4-node parallelization look like. N-node parallelization uses the same logic.



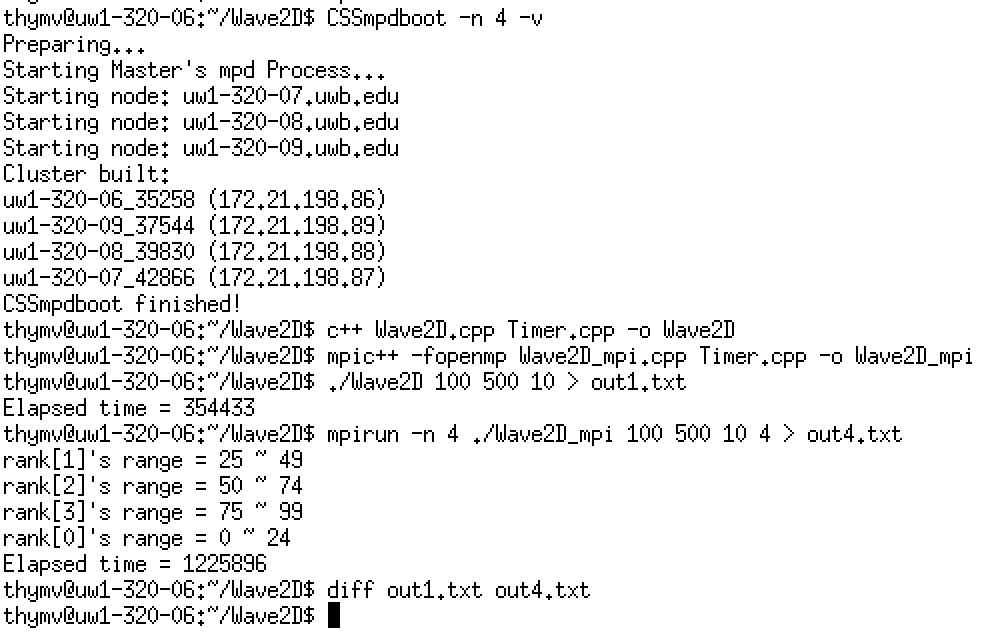
Each node has a copy of z[0] at t = 0 and computes wave heights for t = 1 to t = max\_time for a number of elements which is equal to (total number of elements)/(total number of nodes). For example, when running Wave2D\_mpi with 4 computing nodes and matrix with size 100x100, each node computes elements residing in z[t][rank\*25] to z[t][rank\*25+25-1] at the same time. In order to do Schroedinger equation, first node needs elements residing in z[t-1][25], last node needs elements from z[t-1][74], and each middle node needs elements in z[t-1][rank\*25-1] and z[t-1][rank\*25+25]. Thus, before computation for every t, each node sends edge parts of its z[t-1] to left and/or right neighbors and receive edge parts of its left and/or right neighbors. It does so by making call MPI\_Send and MPI\_Recv in an an appropriate order that does not cause blocking. After receiving the needed neighbor's information, the node divides the wave heights calculation among its OpenMP threads. If the number of threads is specified as 4, each node creates 4 threads to complete wave height calculations for the elements it is in charge of. After that, the slave threads terminate, and if t is at specified time interval, the master thread (of each non-master node) sends the node's computed elements to master node, which will print out computed heights for the whole matrix at time t.

(Cont. next page)

Execution Output

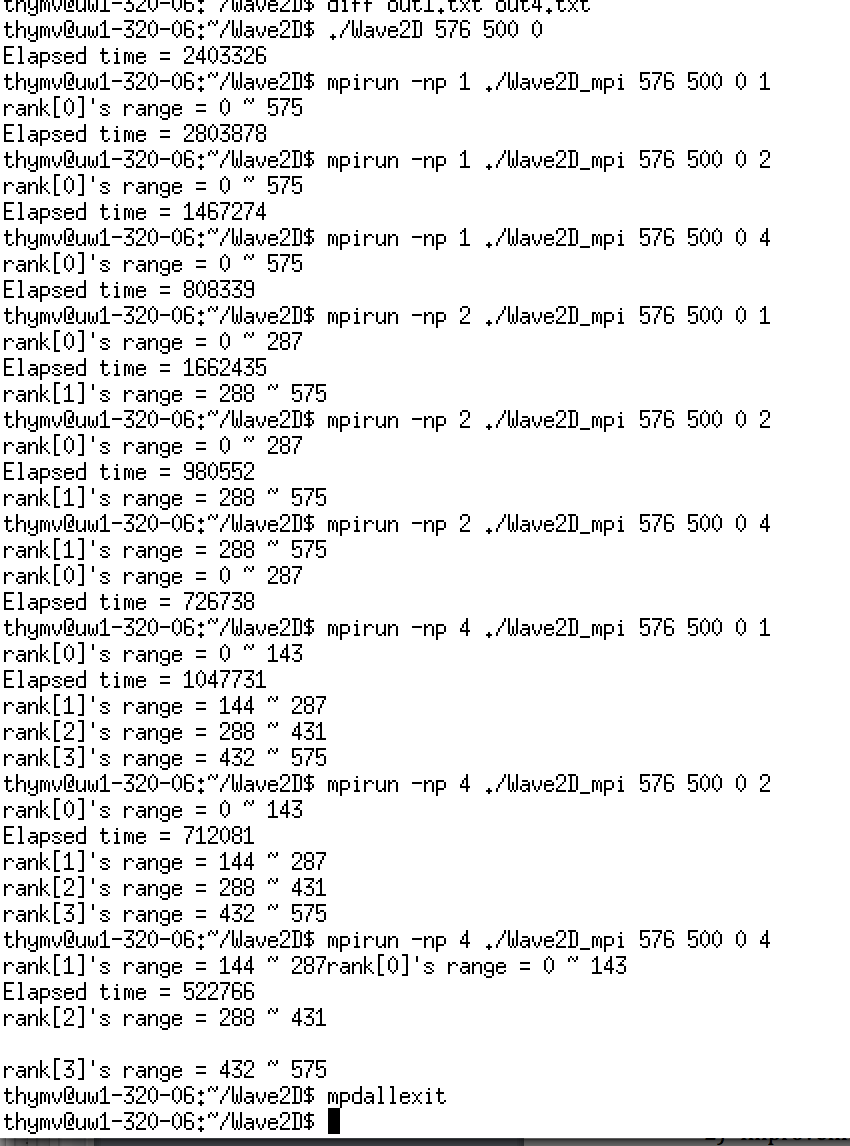
Part 1:

There is no difference between Wave2D and Wave2D\_mpi outputs.



(cont. next page)

Part 2:



Performance analysis

1. Improvement with four machines:

(Wave2D) / (Wave2D\_mpi 4 nodes, 1 thread per node)

= 2403326 / 1047731

= 2.294

( Wave2D\_mpi 1 node, 1 thread ) / (Wave2D\_mpi 4 nodes, 1 thread per node)

= 2803878 / 1047731

= 2.676

1. Improvement with four machines with multithreading

(Wave2D) / (Wave2D\_mpi 4 nodes, 4 threads per node)

= 2403326 / 522766

= 4.592

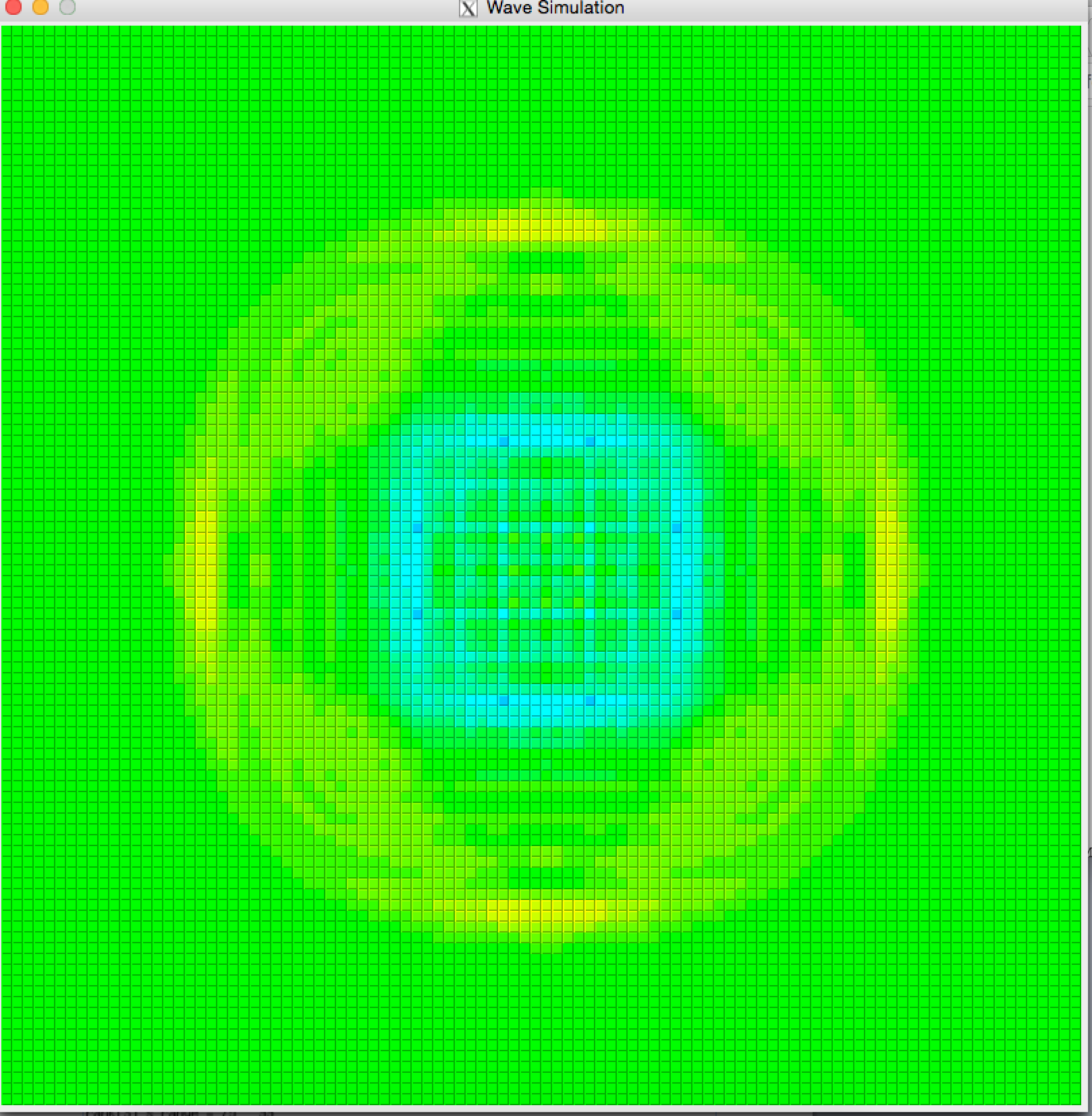
(Wave2D\_mpi 1 node, 1 thread) / (Wave2D\_mpi 4 nodes, 4 threads per node)

= 2803878 / 522766

= 5.363

(Cont. next page)

1. Graphical output:



(Cont. next page)

Discussion

The current parallelization code is limited to working for cases when the size of a square matrix's side is divisible by the number of computing nodes. Workload (column boundary) would need to be logically redefined if the requirement asks for accommodation of non-square matrix and arbitrary matrix size. A possible modification to accommodate arbitrary size: If int k = (int size) % (int number of computing nodes), k number of nodes needs to compute 1+(int size/(int number of computing nodes)) number of columns, and the remaining nodes work on (int size)/(int number of computing nodes) number of columns.

User currently can input any number of computing nodes and threads per node. Using more computing nodes and threads does not always result in better performance, especially when the size of the matrix is small. When there are too many computing nodes and each has a small number of elements to calculate values for, inter-node communication overhead (and potentially thread creation and termination overhead) becomes significant and takes away the benefit of multiple-node parallelization. As seen in execution output on page 3 of this report, execution time lapse for 100x100 matrix on 4-node-with-4-threads/node is longer than on 1-node-1-thread.

At the beginning of execution, many elements close to the matrix's edge (not limited to ones touching the edge) are zero until some time has passed. However, we still use computationally-intensive Schrödinger formula to calculate these elements that we already know will yield 0.0. To improve performance, we can insert code that temporarily excludes computation for these elements, especially when current t value is small and matrix size is very large.