Compiling and usage

To compile just type make in the directory. Two executables are generated.

“serial” is an executable serial version used for comparing performance. Use it like so: “./serial NCOLS NROWS”

“parallel” is an executable parallel version. Use it like so: “mpirun -n NNODES ./parallel NCOLS NROWS XDIM YDIM”. XDIM is how many nodes the x dimension is split between, and YDIM is how many nodes the y dimension is split between. XDIM and YDIM can be set to 0 and MPI will decide the grid’s shape, but I found that it often gave strange grid shapes. An example call with 1000x1000 heat grid, 10 nodes and a 5x5 topology looks like: “mpirun -n 10./parallel 1000 1000 5 5”.

The parallel version includes a serial version that can be used to check the results. This can be enabled by uncommenting the “compare()” call found in parallel.c.

At the end of execution all nodes send their partial result to the root node, which places the partial results into a final result grid. Printing this is disabled due to the larger size however. It can be printed by uncommenting the print function call.

Cartesian Topology

A cartesian topology is used to organise the nodes. Each node uses MPI_Cart_rank to figure out what node should receive data. For example: if the node at x = 1, y = 1 in the grid wants to send data to the node “left” of it, it will use MPI_Cart_rank to figure out which node is at x = 0, y = 1.

Except if the node is at one of the edges, each node will send data to the nodes in the grid that are above, below, to the left and to the right of it.

I tested with many different heat grid sizes and topology shapes and found my code works with any grid size or topology shape as long as the grid handled by each node is the same size. My code cannot handle the situation where work is unevenly distributed between each grid.

Derived Datatypes

When sending data up or down in the grid an ordinary buffer of doubles is used, since the data is flat and contiguous. However when data is sent left or right in the grid the data is a column in the grid so it is not flat or contiguous. MPI_Type_vector is in this case to allow the columns to be sent easily.

Performance

To see how the work scales when communication costs are low I timed the work using 6 nodes all running on on a 6 core CPU. The heat grid size was 9’000 columns and 10’000 rows, while the topology had an x dimension size of 3 and a y dimension size of 2.
Serial version: 86.6 seconds
Parallel version: 11.5 seconds (7.5x faster)
This shows that when communication costs are low the work scales extremely well. On a side note I am not fully sure how it became 7.5x faster when split between only 6 nodes. My guess is less L1 and L2 cache pressure when distributed over many cores.

I then used the HPC cluster to see how the work scales when nodes are on separate computers. I repeated this test twice, once passing -N to salloc, and then passing -n to salloc. This was to see how communication costs are lowered when some of the nodes are on the same CPU. With 16 nodes, a heat grid size of 3’200 columns and 3’200 rows, and a topology with an x and y dimension size of 4 I got the following results:
Serial: 10.46 seconds
Parallel with -N passed to salloc: 3.83 seconds (2.7x faster)
Parallel with -n passed to salloc: 1.98 seconds (5.2x faster)
The scaling in both cases is worse than the scaling when I used 6 nodes all on a single 6 core CPU, but the scaling is two times worse when all nodes are on separate CPUs. This shows how communication costs can have a significant effect on the performance of the application.

Finally I tested again on the HPC cluster with 16 nodes allocated by passing -n to salloc. With a heat grid size of 9’600 columns and 9’600 rows, and a topology with an x and y dimension size of 4 I got the following results:
Serial: 99.87 seconds
Parallel: 13.28 seconds (7.5x faster)
This shows that scaling improves as the workload increases, due to proportionally less time being spent in the communication part.