Parallel Programing: HW6

Andres Imperial

February 29, 2020

Implementation

I implemented my game of life by splitting rows among the given processes. They take their rows plus their adjacent rows to calculate the change in the cells.

Timing: Processors: time 1 : 136 2 : 115 3 : 143 4 : 183 8 : 271

As we can see the increase in number of processors doesn't mean an increase of speed, perhaps due to the overhead of sending the information and the creations of new variables.

Code:

Listing 1: main.cpp

```
#include <chrono>
#include <cmath>
#include <complex>
#include <cstdio>
#include <fstream>
#include <mpi.h>
#include <time.h>
#include <vector>

using namespace std;
#define MCW MPI_COMM_WORLD
int HEIGHT = 20;
int WIDTH = 40;

int getNeighbor(int row, int col, vector<int> &board) {
  // use modulus to get wrapping effect at board edges
  if (row < 0 || row >= HEIGHT) {
    return 0;
  }
  return board[((row + HEIGHT) % HEIGHT) * WIDTH + ((col + WIDTH) % WIDTH)];
}
```

```
int getCount(int row, int col, vector<int> &board) {
  int count = 0;
  std::vector<int> deltas{-1, 0, 1};
  for (int dc : deltas) {
    for (int dr : deltas) {
      if (dr || dc) {
        count += getNeighbor(row + dr, col + dc, board);
      }
    }
  }
  return count;
}

int tick(vector<int> &board, int row, int col) {
  int count = getCount(row, col, board);
  bool birth = !board[row * WIDTH + col] && count == 3;
  bool survive = board[row * WIDTH + col] && (count == 2 || count == 3);
  return birth || survive;
}

void showCell(int cell) { std::cout << (cell ? "*" : "␣"); }

void showCells(vector<int> &board) {
  for (int row = 0; row < HEIGHT; ++row) {
    std::cout << "|";
    for (int col = 0; col < WIDTH; ++col) {
      showCell(board[row * WIDTH + col]);
    }
    std::cout << "|\n";
  }
}

int main(int argc, char **argv) {
  int rank, size;
  int data;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MCW, &rank);
  MPI_Comm_size(MCW, &size);

  auto start = std::chrono::high_resolution_clock::now();
  std::vector<int> allRows(HEIGHT * WIDTH, 0);

  int numRows = HEIGHT / size;
  if (rank < static_cast<int>(HEIGHT) % size) {
    ++numRows;
```

```
}

if (rank == 0) {
  srand(time(NULL));
  for (int row = 0; row < HEIGHT; ++row) {
    for (int col = 0; col < WIDTH; ++col) {
      for (auto &&val : allRows) {
        val = rand() % 2;
      }
    }
  }
  showCells(allRows);
}

int numGens = 10;
for (int genCount = 0; genCount < numGens; ++genCount) {
  if (rank == 0) {
    // The rows they need to compute plus border rows above and below if
    // applicable
    auto start = numRows * WIDTH;
    for (int i = 1; i < size; ++i) {
      int tempRows = HEIGHT / size;
      if (i < static_cast<int>(HEIGHT) % size) {
        ++tempRows;
      }

      auto dataSize = (tempRows + 2) * WIDTH;
      if (i == size - 1) {
        dataSize = (tempRows + 1) * WIDTH;
      }
      MPI_Send(&allRows[start - (1 * WIDTH)], dataSize, MPI_INT, i, 0, MCW);
      start += (tempRows * WIDTH);
    }
  } else {
    // Recieve data
    auto dataSize = (numRows + 2) * WIDTH;
    if (rank == size - 1) {
      dataSize = (numRows + 1) * WIDTH;
    }
    std::vector<int> temp(dataSize, 0);
    MPI_Recv(&temp[0], dataSize, MPI_INT, 0, 0, MCW, MPI_STATUS_IGNORE);
    auto original = temp;
    for (int row = 1; row < numRows + 1; ++row) {
      for (int col = 0; col < WIDTH; ++col) {
        temp[row * WIDTH + col] = tick(original, row, col);
      }
```

```
    }
    // Send data to thread 0
    MPI_Send(&temp[WIDTH], numRows * WIDTH, MPI_INT, 0, 0, MCW);
  }

  if (rank == 0) {
    std::vector<int> myWork(numRows * WIDTH);
    for (int row = 0; row < numRows; ++row) {
      for (int col = 0; col < WIDTH; ++col) {
        myWork[row * WIDTH + col] = tick(allRows, row, col);
      }
    }
    // Recieve data
    vector<vector<int>> finalRows(size, vector<int>{});
    finalRows[0] = myWork;

    for (int i = 1; i < size; ++i) {
      int tempRows = HEIGHT / size;
      if (i < static_cast<int>(HEIGHT) % size) {
        ++tempRows;
      }

      std::vector<int> temp(tempRows * WIDTH);
      MPI_Recv(&temp[0], tempRows * WIDTH, MPI_INT, i, 0, MCW,
               MPI_STATUS_IGNORE);
      finalRows[i] = temp;
    }
    // put data from final rows to all rows
    allRows.clear();
    for (int i = 0; i < size; ++i) {
      for (auto val : finalRows[i]) {
        allRows.push_back(val);
      }
    }
    showCells(allRows);
    cout << endl;
  } else {
  }
}

MPI_Finalize();

if (rank == 0) {
  auto stop = std::chrono::high_resolution_clock::now();
  auto duration =
      std::chrono::duration_cast<std::chrono::milliseconds>(stop - start);
```

```
    printf("Duration: %li\n", duration.count());
  }

  return 0;
}
```

How to run:

```
mpic++ main.cpp -o main
mpirun -np <n> -oversubscribe ./main
```

Output

Timing subject to change dependent on machine used for execution.