CS3210 Assignment 2 Report

Sriram Sami

November 12, 2016

Contents

1 Approach

1.1 Partitioning

The SETL problem can be partitioned in two ways: **Domain** and **Functional**.

1.1.1 Domain Decomposition

Generally, the whole SETL problem involves **searching for patterns** in the board, **evolving** the board, and repeatedly performing these steps for the given number of iterations.

One can note that, for any $k_1 \times k_2$ sized area of the board, where k_1 and k_2 are respectively less than the overall length x height ($size \times size$), we can add a buffer area of size $patternsize - 1$ (bounded by the size of the board) in all four directions around the $k_1 \times k_2$ area, and then no other infomation is necessary to perform a correct pattern match + evolution on that area (i.e. there is a way to split the board up into independant parts with some overlap in elements).

Hence this $MAX(k_1 + patternsize - 1, size) \times MAX(k_2 + patternsize - 1, size)$ area is the most basic and independant chunk of data in this problem, and we should also observe that as k_1 and k_2 increases, the ratio of overlapping area between these data blocks to the actual area being pattern-searched and evolved decreases i.e. the overhead decreases.

1.1.2 Functional Decomposition

Another way of approaching the problem is to look at the two main functions in the problem: *searching* and *evolving*. Multiple computation units could be focused on searching during a particular iteration whereas other computation units could perform evolution independantly of the pattern searching and communicate the results back to the pattern-searching components.

1.1.3 Evaluation

The issue with the functional decomposition approach is that the pattern searching components depend on the speed of the evolution components and it's highly likely that they will not take the same amount of time, meaning there is very likely going to be unused processor time on the evolution components' side even in the best case. Therefore, the domain-decomposition method seems the most useful in this particular scenario - since it can be parallelized independantly to many processors, but *within a single iteration.*

1.2 Communication

1. The root computation unit has to read the world file and pattern file and send the world and 4 patterns over to all other computation unit.
2. After each iteration, every computation unit must share the slice of world it has evolved to every other computation unit: can be done with *MPI_Alltoall*.
3. After all iterations are completed, each computation unit must send its list of matches back to the root computation unit.

1.3 Agglomeration

To reduce overlap in both the column and row dimensions - for this situation, I decided to a **row-based approach** to partitioning the data into tasks. This also simplified communication since contiguous blocks of data could be send from processor to processor - and the memory accesses had the added benefit of limiting cache misses due to their spatial proximity.

Hence the (simplified) overall work for one task is to:

1. If root task: read the game world and patterns and broadcast to all tasks
2. Receive the game world and the list of patterns
3. Receive the rows that the particular task is responsible for pattern-searching and evolving
4. Pattern-search the specified rows, append the results to the result list
5. Evolve the specified rows
6. Perform an MPI_Alltoall operation - all tasks should now have the most updated/evolved world
7. Proceed with the next iteration until all iterations are complete
8. Send the result list back to the root task itemIf root task: sort and display the results

1.4 Mapping

I decided to constrain the number of processors to be used to be only multiples of the world size (e.g. for a world size of 20: 1, 2, 4, 5 processors were acceptable. Note that certain processor counts are not acceptable as I also set a hard limit on how many rows a processor must have as larger than or equal to the pattern size to reduce overhead.) A trivial mapping of one processor to one task was used.

2 Results

2.1 NSCC

The execution context for NSCC was:

- **48 CPUs** (max for that instance)
- **50 mpiprocs**, **50 processors** (i.e. -np 50)
- **random3000.w world**
- **100 iterations**
- **glider5.p pattern**.