CS3210 Assignment 2 Report

Sriram Sami

November 14, 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 take a **row-based approach** to partition the data into tasks. This also simplified communication since contiguous blocks of data could be sent 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 factors 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

The execution context in general was:

- **random3000.w world**
- **glider5.p pattern**
- **100 iterations**

This was specifically chosen to be as close to the real benchmark as possible.

2.1 NSCC

The additional specific execution contexts for NSCC were:

- **40 cores** (max for that instance)
- **40 mpiprocs**, **40 copies of the program** (i.e. -np 50)

In summary, after optimizations:

1. The sequential version of SETL took **125.20** seconds on the NSCC machine on average
2. The parallel version of SETL took **2.11** seconds on the NSCC cluster on average
3. The speedup obtained is thus **59.33x**, and the efficiency is **1.48**, given 40 execution units.

2.1.1 Optimizations

We note that the efficiency of this execution is superlinear - 1.48 > theoretical maximum of 1.

Only limited optimization had to be done (just finding the correct number of procesors) on the NSCC machine as speedup kept increasing until close to the limit of 48 cores. My version of SETL_par works beyond this limit but I did not count those results in this report. The results take this form:

Number of cores used	Time Taken (Parallel, s)
1	211.02
10	221.93
20	3.48
30	2.54
40	2.11
50	3.27

Figure 1: Performance characteristics of the NSCC cluster with increasing core counts

While it's unclear why performance dipped so much after the 10 core mark, perhaps memory copying or the some scheduling/thrashing barrier being overcome - it's clear that the optimal number is 40.

2.2 Lab

The additional specific execution contexts for the lab machine were:

- **8 cores** (-np 8), run from **only** the i7 node.

In summary, after optimizations:

1. The sequential version of SETL took **112.33** seconds on the lab machine on average
2. The parallel version of SETL took **9.16** seconds on the lab cluster (only the i7 was used) on average
3. The speedup obtained is thus **12.26x**, and the efficiency is **1.53**, given 8 execution units.

The first thing we note is that the efficiency of this configuration exceeds the theoretical maximum limit of 1, indicating that we achieved a superlinear speedup. The second thing we note is that only the i7 machine was used instead of taking advantage of the whole cluster. This was due to lower performance being observed under the given parameters when the work was distributed off the i7 machine, and this is very likely due to the overhead of sending data and synchronizing between the lab cluster machines.

2.2.1 Cluster vs i7 only tests

Some selected results from the combined testing of machinefile configurations and the number of processing units (-np) are shown below:

Number of cores used	Machinefile	Time Taken (Parallel, s)
8	Only i7	9.16
8	Lab cluster	84.94s
12	Lab cluster	74.05s
15	Lab cluster	79.05s

Figure 2: Performance characteristics of the lab cluster versus the single i7 machine

Many other test configurations are omitted for brevity. Runs of *perf stat* indicate that the lab cluster configurations ran at 1.31 instructions per cycle whereas the pure i7 configuration ran at 1.43 instructions per cycle, which is a large difference considering then number of cycles executed. Many other characteristics like the branch prediction rate favour the i7 only configuration. However, the main difference is likely to be the memory access and communication time due to the memory locality in the i7 machine versus the distributed memory layout of the cluster.

2.2.2 Number of cores tests

The 8 core usage on the i7 machine was experimentally tested and proven to be the optimal number - which led me to come up with the final configuration.

Number of cores used	Time Taken (Parallel, s)
1	33.38
2	17.49
4	9.40
6	11.36
8	9.16
10	13.78

Figure 3: Performance characteristics of the i7 machine with reference to processors used

Hence the 8-core setup was chosen to minimize overall execution time.