COMP90025 Parallel And Multicore Computing: Assignment 1A

Martin Raunkjær Andersen - login: marand
mrandersen@student.unimelb.edu.au
1011164
Matthijs van Delft - login: matthijsvandelft
mvandelft@student.unimelb.edu.au
1010922

August 24, 2018

This report documents the optimisation modifications on the sequential Floyd Warshall (FW) algorithm. Two main areas of optimisation is discussed: General optimisation of the C code and parallel optimisation using the OpenMP framework. In the end, the run-time and performance gain results are given for a number of cores.

1 General Optimisation

Two general optimisation techniques are used: *Loop unrolling* and *boundary check reduction*. Loop unrolling concerns increasing the number of operations that happen in a single iteration of a loop, to take advantage of increased pipe lining in processors and less loop iteration overhead[1]. Boundary check reduction avoids the implicit boundary checks that happens when one tries to access an index of an array, by reading the content of a pointer to the same memory address as that of the array index. Both techniques are illustrated in Listing 1, which contains most of the modified FW-algorithm produced.

```
1  for (int k = 1; k <= nodesCount; ++k) {
2      dik += k;
3      for (int i = 1; i <= nodesCount; ++i) {
4          dik += MAX;
5          dij += MAX;
6          for (int j = 1; j <= nodesCount;) {
7              if (nodesCount - j > 8) {
8                  ++dij;
9                  if ((*dik + distance[k][j])
       < *dij) *dij = *dik + distance[k][j];
10                 ++dij;
11                 if ((*dik + distance[k][j
       +1]) < *dij) *dij = *dik + distance[k][
       j+1];
12                 ...
13                 j += 8;
14             }
15             else {
16                 ++dij;
17                 if ((*dik + distance[k][j])
       < *dij) *dij = *dik + distance[k][j];
18                 ++j;
19             }
20         }
21         dij -= nodesCount;
22     }
23     dij -= nodesCount * MAX;
24     dik -= k + nodesCount * MAX;
25 }
```

Listing 1: Modified Floyd Warshall algorithm

Loop unrolling is showcased from line 7 to 19 in Listing 1. 8 operations, each of which would normally be in its own loop iteration, are put together in a single loop iteration. More than 8 operations could be placed in each iteration, but the grow rate of the performance gains decreases quickly at this point.

Boundary check reduction occurs whenever a value is accessed by pointer dereferencing, instead of by array index, such as at line 9 in Listing **??**. To make use of this technique, the pointers (*dik* and *dij*) must be kept up to date with the current iteration of the loops. This happens every time a pointer itself is modified, such as in line 2, 3, 4 and 8.

2 Parallel Optimisation

To parallelise the sequential FW-algorithm, the `parallel for` construct of OpenMP are used, which splits iterations of a for loop into chunks and assigns them to different cores. This can be seen in listing 2, which is the pragma associated with the nested for loop as seen in listing 1.

```
1 #pragma omp parallel for firstprivate(dik,
      dij) schedule(static, chunkSize)
      num_threads(maxNrThreads)
```

Listing 2: OpenMP parallel for construct

The clauses given to `parallel for` signifies the following: `firstprivate(dik, dij)` means that the *dik*, *dij* variables should retain the values they had outside the loop scope, in this case pointers to the start of the adjacency matrix. `schedule(static, chunkSize)` specifies that the loops should be split to the cores in size of *chunkSize* (in our case $chunkSize = nodesCount/numCores$). `num_threads(maxNrThreads)` specifies the maximum number of cores for the respective machine. The behaviour of the loop is identical in every iteration (no break, skips etc.) which makes it beneficial to static scheduling. Static scheduling produces

the least overhead[2] and since the execution of the chunks is identical it can be assumed that every core will finish at the same point in time. This results in the smallest waiting time for a possible barrier happening after the parallel for section.

3 Results

The execution times of the modified FW-algorithm on multiple numbers of cores is described in table 1 and visualised by the orange bars in figure 1. The result are based on a data set containing 2000 nodes and 50000 edges. This data set is generated with a script that randomly connect edges between nodes with arbitrary weights. The same data set executed with the sequential FW-algorithm, results in a execution time equal to 36.214 seconds, visualised in figure 1 by the blue bar. The sequential FW-algorithm does not utilise multiple cores, hence there is only one blue. The behaviour of the reduction in execution time can be described by x^{-n} where $0 < n < 1$.

Cores	Exe. time (s)	Cum. Gain	Step Gain
1	23.666	34.650	34.650
2	14.822	59.071	37.370
4	7.164	80.218	51.666
8	3.960	89.065	44.724
12	1.488	95.891	62.424

Table 1: Parallel execution time table

Described in the previous section, basic optimisation techniques are used which are not related to any parallel programming, hence there is a performance gain between the sequential FW-algorithm and the modified FW-algorithm executed on 1 core, see the blue and orange at 1 core in figure 1.

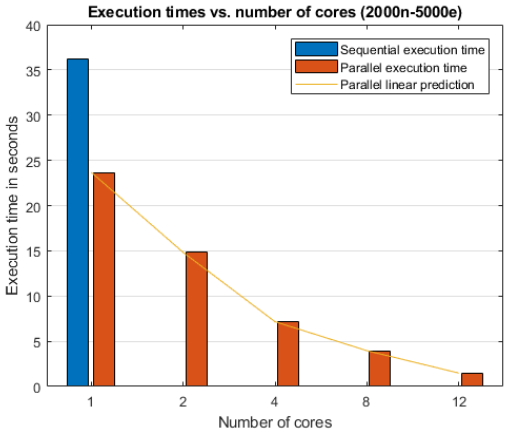

Figure 1: Execution time graph

Figure 2 described the performance gain in two ways. The cumulative gain is derived from the execution time on different number of cores compared to the sequential execution time (36.214 seconds). The step gain is derived from the comparison in execution time between the respective number of cores and the execution time of the previous run, e.g., the gain for 8 cores compared to 4 cores; 4 to 2 cores etc. The cumulative gain can be described as a function of $log_b(a)$ where $b \geq 1$ and a is the number of cores.

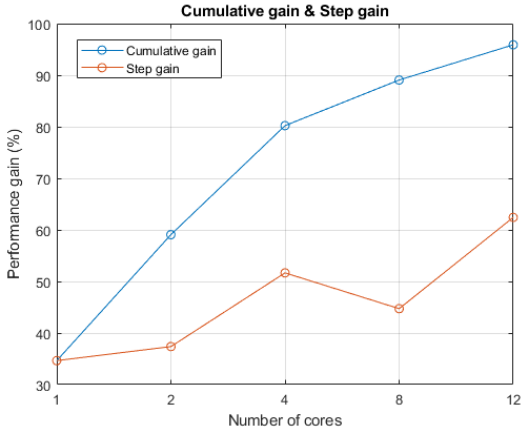

Figure 2: Performance gain graph

References

[1] *Writing efficient c and c code optimization - codeproject*, `https://www.codeproject.com/Articles/6154/Writing-Efficient-C-and-C-Code-Optimization`, (Accessed on 08/23/2018),

[2] D. College, *Scheduling of parallel loops*, `https://www.dartmouth.edu/~rc/classes/intro_openmp/schedule_loops.html`, (Accessed on 08/24/2018),