COMP90025 Parallel And Multicore Computing: Assignment 1B

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

September 15, 2018

This report documents the parallel optimisation modifications on the Mandelbrot set count[1]. Two main areas of optimisation is discussed: General optimisation of the C code and parallel optimisation using the OpenMPI framework. In the end, the run-time and performance gain results are given for a number of processors.

1 Vision

The vision for the implementation was to split up the calculation of the Mandelbrot set by the amount of nodes, using OpenMPI. This was to be done by identifying a process for each node, split the work up on these processes and let each process enter its individual sub problem of the Mandelbrot set, while using OpenMP to take advantage of all the threads on a given node. The advantage of this approach is the reduction in communication, since the threads on the same node share memory. Hence, these threads do not need OpenMPI to communicate but can utilize the shared memory in that node, which result in less barriers, hence generally a faster execution time.

2 Parallel Optimisation

During the implementation of the proposed vision in section 1 many inconveniences were encountered. One of the main difficulties was to determine which processes is part of which node. This information is crucial to determine the communication between the different nodes. The assumption was made that duplicate processes per node had to be filtered out manually.

Another concerns was whether the nodes would have an arbitrary number of cores, which would result in varying execution times on different nodes for subset of an equally sized "problem" (equal input arguments). This issue gets more significant whenever the size of the sub-problem grows. The nodes with higher performance will get further and further ahead of lesser nodes, hence, at a barrier, the greater node will have to wait until the others nodes have caught up.

Because of these difficulties a different approach was attempted, with no shared memory between processes assumed, thereby foregoing OpenMP usage. This approach has the advantage of being (almost) unaffected by the location of the processors. It comes at the cost of more communication, however. Work in each region is split up into chunks of one between the processors by utilizing the processor id, as can be seen in Listing 1. An alternative to this approach is to let the chunk-size be N/P, where N is the number of loop iterations in a region and P is the number of processes. The performance difference between these approaches is discussed in Section 4.

```
mandelbrotSetCount(realLower, ..., num,
 ..., idProcess, nrProcesses)
    FOR each (real + idProcess) < num
        real ← real + nrProcesses
        tmpReal ← realLower + (real +
idProcess) * realStep
        /// functionality inset(...)
```

Listing 1: Loop split approach

This method utilizes the fact that the process id are sequential numbers $0, 1, ..., P-1$ where P is the number of processes. The `real` iterator variable increases `nrProcesses` per iteration of the loop and the `tmpReal`, which `inset` is called with, is calculated based on the value of the process id plus the `real` iterator. The *(real + idProcess) ¡ num* termination condition ensures that too much work is not done.

At the end of a region calculation, as can be seen in Listing 2, a call to `MPI_Reduce` is made, which sums the `localCount` variable of each process into the `globalCount` variable of the master process. This is then printed to standard output as the result for that region.

```
MPI_Reduce(&localCount, &globalCount,1,
    MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
```

Listing 2: `MPI_Reduce` call to sum results

3 General Optimisation

A few general optimisation techniques are used. First, the function *inset(...)* is iteratively called

from the function *mandelbrotSetCount(...)*, and the number of function calls is determined based on the number of Mandelbrot set regions to be calculated times the *num* input argument for each individual region. To reduce the overhead of the numerous function calls, the functionality of *inset(...)* is integrated in *mandelbrotSetCount(...)*.

Secondly, instead of declaring a variable in an inner loop, initialization is done as close to the outer loop as possible. This way the variable does not have to be constantly allocated in memory, instead it will exist and be reset to its initialization value whenever necessary.

4 Results

The following results are the product of program execution on the physical partition of the High Performance Computing system *Spartan*[2], with the input in Appendix 6.1. Due to running time differences of up to 500 milliseconds, each result in Table 1, is the average of 10 runs. Possible reasons for this difference could be the quality and closeness to each other of allocated processors.

CPUs	Chunksize N/P	Chunksize 1	Sequential
1	26.009	25.998	26.118
5	10.307	6.454	
8	7.463	5.224	
16	4.130	3.536	
32	3.476	3.447	

Table 1: Average execution time given different numbers of allocated processors, measured in seconds.

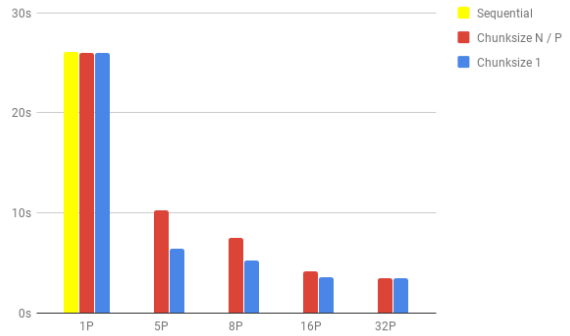

Figure 1: Execution time graph, with allocated processors on the x-axis and execution time in seconds on the y-axis.

As can be seen in Table 1 and Figure 1, the chunk size of the work split between processors has a great impact on the execution time with fewer processors allocated. This is because some sub-regions in a given region require drastically more computation than others to determine whether their points are part of the Mandelbrot set. by splitting up the work into smaller chunks, the computation-heavy sub-regions are distributed more evenly among the processors. Since the processors has to wait for each other to finish after each region, the higher utilization factor of each processor gained by the more even split of work translates to drastically less waiting time overall. The speedup at 5 processors allocated with chunksize 1 is $26.118/6.454 = 4.047$ and the efficiency is $26.118/6.454 * 5 = 0.809$, which is far from optimal but acceptable results.

With larger amounts of processors, however, the efficiency plummets for both chunksizes, with the speedup plateauing at about $26.118/3.536 = 7.386$ for 16 processors. The efficiency of 16 processors is $26.118/3.536 * 16 = 0.462$, while the efficiency of 32 processors is $26.118/3.447 * 32 = 0.237$. The bottleneck at this point is probably the communication costs, which are $P * R$, where P is the number of processors and R is the number of regions in the input, along with the fact that the processors has to wait for each other after each region, wasting utilization time.

5 Discussion

A dynamic load balancing approach would have eliminated the need for barriers, and thereby increased processor utilization, at the cost of increased communication, as the work would have to be split up using OpenMPI instead of calculated locally, as done by us. It would have been interesting to compare the results from such an approach with ours.

6 Appendix

6.1 Test Arguments

```
-2.0 1.0 -1.0 1.0 200 10000 -1 1.0 0.0 1.0
300 10000 -1 1.0 -1.0 0.0 500 10000 -1 0.0
-2.0 -1.0 100 10000 -1 0.0 0.0 2.0 100
10000 0 2.0 0.0 2.0 100 10000 1 2.0 -1.0
1.0 200 10000 -2.0 2.0 -2.0 2.0 200 10000
-1.0 2.0 -1.0 1.0 200 10000 0.0 2.0 0.0
1.0 300 10000 0.0 2.0 -2.0 1.0 200 10000
-1 0.0 -1.0 1.0 500 10000 -1 1.0 -1.0 2.0
700 10000
```

References

[1] E. Weisstein, *Mandelbrot set – from wolfram mathworld*, `http://mathworld.wolfram.`

com / MandelbrotSet . html, (Accessed on 09/13/2018),

[2] U. of Melbourne, *Spartan documentation*, https://dashboard.hpc.unimelb.edu.au/, (Accessed on 09/13/2018),