Parallel and Distributed Computing

MEEC - MEIC

Project Report - OpenMP

Autores:

João Vinagre (93095) marquesvinagre@tecnico.ulisboa.pt
André Santos (96152) andresantos1.alf@gmail.com
Tomás Marques (99338) tomas.c.marques@tecnico.ulisboa.pt

Group 4

2023/2024 – 2º Semestre, P3

1 Introduction - Game of Life 3D

For the Parallel and Distributed Computing project, we were tasked with implementing a 3D simulation of Conway's Game of Life, using OpenMP and MPI and applying concepts and methods learned in the theoretical classes. This report will focus on the OpenMP implementation of the program, along with a brief explanation of the choices made during the linear implementation to facilitate parallelization.

2 Linear Implementation

In this section, we will focus on the linear implementation of the program. To compute each generation, it is required to analyse each cell of the map by counting its neighbours and rank the species. The function `next_state()` is responsible for performing the task of computing the cell's next value. The function `simulate()`, iterates over the generations by calling `next_state()` for every cell sequentially. To get the best out of the cache, the map is accessed in memory order.

2.1 Results

To assert the correctness and performance of the program, the program was run with the example arguments given in the class page. The timing results obtained in the Lab 1, Computer 3 are shown in the table below1.

Table 1: Times of running serial program.

	# Generations	Cube Side	Density	Seed	Time
Example 1	1000	64	0.4	0	23.9
Example 2	200	128	0.5	1000	40.8
Example 3	10	512	0.4	0	165.4
Example 4	3	1024	0.4	100	421.2

3 OpenMP Adaptation

To take advantage of the multi-core architecture of our computers, the program was made parallel with the help of OpenMP. Since the serial version was initially made with parallelism in mind, not much was changed on the original implementation.

Initially, the primitive task was identified as the `next_state()` function. So, a small effort was made to reduce the number of `if` statements resulting in a performance improvement on each iteration.

The computation required for each call of the `next_state()` function is always the same, so the scheduling used was the default, which is static. This means the work is equally distributed among the threads. This scheduling, apart from being simple and predetermined, also takes advantage of data locality improving cache utilization.

To implement the division discussed above, a simple pragma directive was enough:

```
#pragma omp parallel for reduction(+:current_speties_count)

private(x_linearized, y_linearized)
```

The above line of code was added in the loop that runs over the generations since these are dependent on each other, Resulting in the splitting of an equal number of ”faces” among the threads.

The first pass through the grid, to count the numbers of each species in generation 0, was also made parallel with a `#pragma omp parallel for` with reduction of the population totals.

The remaining loops in the `simulate()` function were vectorized using a `#pragma omp simd`, resulting in a negligible improvement. No extra threads were created due to the smaller size of the loop not justifying it.

4 Performance Analysis

Running the OpenMP adaptation in the Lab 1 Computer 3, which has a Intel Core i5-7500 (four cores, each with a single thread and running at 3.40 GHz), we obtained the following times and speedup values:

Table 2: Times of running parallel program with one thread.

1 thread	# Generations	Cube Side	Density	Seed	Time	Speedup (%)
Example 1	1000	64	0.4	0	24.4	0.97
Example 2	200	128	0.5	1000	42.2	0.96
Example 3	10	512	0.4	0	169.9	0.97
Example 4	3	1024	0.4	100	425.0	0.99

The times observed in the OpenMP implementation running with only one thread are worse than the ones seen in the linear implementation above. This is due to the fact that, even when using one thread, there are overheads related to thread creation in OpenMP.

Table 3: Times of running the program with 2 threads and respective speedup.

2 threads	# Generations	Cube Side	Density	Seed	Time	Speedup (%)
Example 1	1000	64	0.4	0	12.3	1.94
Example 2	200	128	0.5	1000	21.2	1.92
Example 3	10	512	0.4	0	85.2	1.94
Example 4	3	1024	0.4	100	214.1	1.96

Table 4: Times of running parallel program with 3 threads and respective speedup.

3 threads	# Generations	Cube Side	Density	Seed	Time	Speedup (%)
Example 1	1000	64	0.4	0	8.6	2.77
Example 2	200	128	0.5	1000	14.6	2.77
Example 3	10	512	0.4	0	58.6	2.82
Example 4	3	1024	0.4	100	147.2	2.86

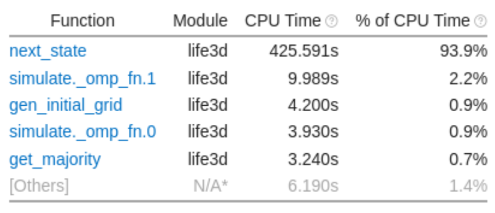
Table 5: Times of running parallel program with 4 threads and respective speedup.

4 threads	# Generations	Cube Side	Density	Seed	Time	Speedup (%)
Example 1	1000	64	0.4	0	6.4	3.81
Example 2	200	128	0.5	1000	10.9	3.74
Example 3	10	512	0.4	0	43.7	3.78
Example 4	3	1024	0.4	100	110.0	3.82

As it can be seen by the tables above, the values obtained for speedup are very close to the number of threads used. Getting any closer to the ideal speedup (number of threads) would be very difficult, as the implementation makes use of reductions which are executed sequentially. Since the CPU used does not have hyper-threading, only possessing one thread per physical core, there are also no overheads or resource sharing between threads, which means all run sequentially.

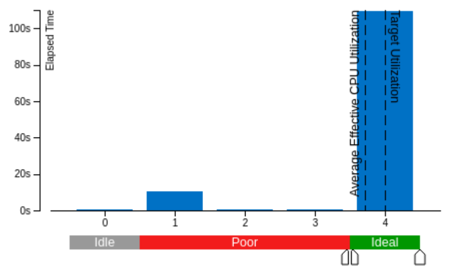
4.0.1 Intel VTune Profiler

Intel VTune Profile was then used to analyse if the implementation utilizes the available resources as much as possible. These analysis were also ran on the computers present in Lab1, which has the four-core CPU referred above. The figures below show values for the most complex example given (`$ life3d 3 1024 .4 100`)



Function	Module	CPU Time	% of CPU Time
next_state	life3d	425.591s	93.9%
simulate._omp_fn.1	life3d	9.989s	2.2%
gen_initial_grid	life3d	4.200s	0.9%
simulate._omp_fn.0	life3d	3.930s	0.9%
get_majority	life3d	3.240s	0.7%
[Others]	N/A*	6.190s	1.4%

(a) Hotspot Analysis



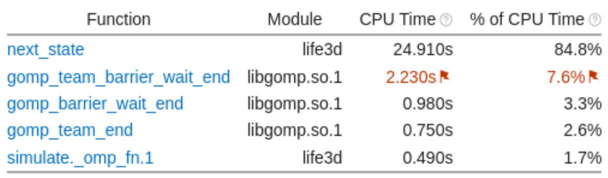
(b) Threading Analysis

Figure 1: Vtune Analysis for the Last Example

As it can be seen, most CPU time is spent on the `next_state` function. This is to be expected, as it is the most compute-intensive section of the code. Due to the nature of the parallelization, each thread is running it's own instance of `next_state`, resulting in the *CPU Time* observed in Vtune being around four times (the number of cores) the total execution time.

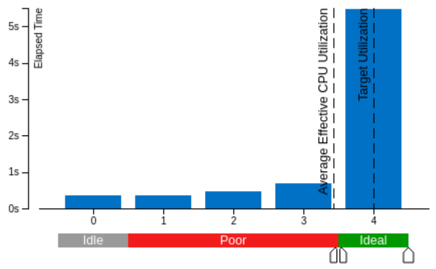
It is possible to infer, from the information present in the performed analysis, that our implementation achieves very high core utilization. It is also worth noting that the time spent in serial execution (only one core) is very close to the time used by the function `gen_initial_grid`, which is sequential. This means that in practical terms, the algorithm is running almost exclusively in parallel across all threads, for this problem size. The Effective CPU Utilization displayed by Vtune after the analysis for this case was of 93.0%.

Below are the same analysis for the first example given (`$ life3d 1000 64 0.4 0`).



Function	Module	CPU Time	% of CPU Time
next_state	life3d	24.910s	84.8%
gomp_team_barrier_wait_end	libgomp.so.1	2.230s	7.6%
gomp_barrier_wait_end	libgomp.so.1	0.980s	3.3%
gomp_team_end	libgomp.so.1	0.750s	2.6%
simulate._omp_fn.1	life3d	0.490s	1.7%

(a) Hotspot Analysis



(b) Threading Analysis

Figure 2: Vtune Analysis for the First Example

As expected, the `next_state` function utilizes most of the CPU time. There is also a lot of time spent in a OpenMP barrier, which is part of the reduction used in the implementation. When performing the threading analysis for the smaller sized grid, it is possible to see that thread utilization is not as good as the one seen above, even showing a significant amount of idle time. This is due to the fact that there are less elements to process; due to the nature of static scheduling, imbalances in distribution are more apparent. In this scenario, as there is less work to distribute among the threads, there is a greater chance that one thread performs considerably more work than another, which will be left without nothing to do, since there are no more tasks to distribute. The Effective CPU Utilization displayed by Vtune after the analysis for this case was of 85.9%.

5 Conclusion

Overall, the OpenMP implementation was very successful, possessing very high usage of the available resources and a speed-up very close to ideal. It is also worth noting that utilization was superior when benchmarking larger problems, which means that the implementation speedup scales with problem size.