Third deliverable

Par4113: Jordi Bosch, Dean Zhu

April 2018

Contents

1 Introduction

The aim of these deliverable is to describe the perfect parallelism found when computing independent data. In our case the object to be studied is the mandelbrot set.

The mandelbrot set is formed by those complex numbers c which do not diverge under the recurrence relationship:

$$z_0 = 0, z_{n+1} = (z_n)^2 + c$$

It can be immediately seen that the verification whether each point belongs to our set is independent to each other as it only depends on the complex number chosen.

Points which belong to the set after k iterations are colored in white, other regions are colored according to the number of iterations before exiting the bounded region.

We are going to analyze how does the performance of the mandelbrot set calcuation behaves under several parallelization strategies and what causes some common pitfalls of these strategies.

Note that several metrics such as time elapsed are located in the corresponding directories.

2 Tareador

Before trying to apply parallelism to our code, we should first check what type of dependencies exists in order to be able to parallelize it correctly. We are going to consider two scenarios. First when we use a parallel strategy for the outer loop in our main function, hence parallelizing each row computation. And a second one where we parallelize each inner iteration.

2.1 Row Granularity

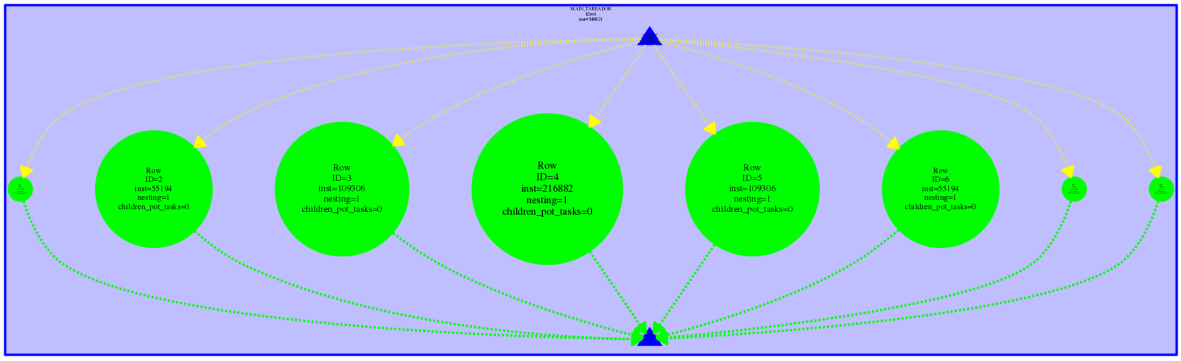


Figure 1: Dependency graph of mandel.c, 8x8 image with row granularity

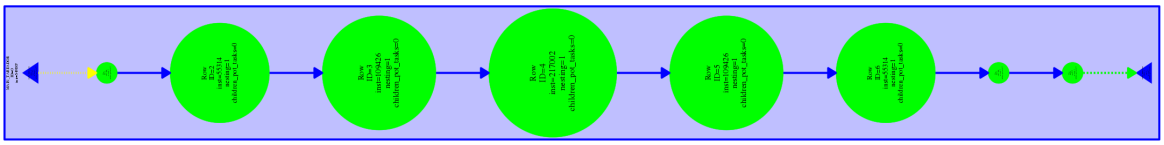


Figure 2: Dependency graph of graphic version of mandel.c, 8x8 image with row granularity, rotated 90 degrees

Firstly we can see that both task dependency graph contain 8 different sized tasks. The reason for the discrepancy in size is caused by the way we compute which points belong to the mandelbrot set. As we run a while loop when a point is still in the convergence radius it makes the rows which contain more points belonging to the set run more iterations. As we are dividing it by rows it is logical that those closer to the origin have more points in the set and hence run more iterations to compute as they do not diverge. The main difference we observe between the two graphs is the way the dependency edges are created. The graphical version of the program creates a sequential task dependency graph whereas the other version creates tasks which are totally independent from each other. After checking the code we can conclude that main culprit are these two lines:

```
XSetForeground (display, gc, color);
XDrawPoint (display, win, gc, col, row);
```

It appears that the way it renders the final image is pixel by pixel in left to right and top to bottom order and hence it makes all pixel computations dependant on all previous tasks. If we comment out any one of those out both lines we can see that it behaves just as the non-graphic version, and if we only comment out one of them nothing changes.

2.2 Point Granularity

Just as the row granularity we observe the same patterns when using point granularity. The observation on parallelism and secuential execution still stands and moreover we get a further proof that points which seem to belong to the mandelbrot set take more iterations. Those are the tasks which take longer time.

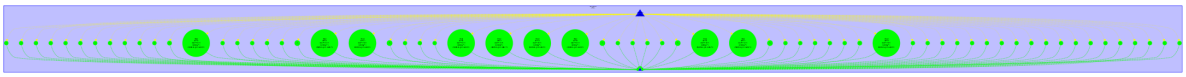


Figure 3: Dependency graph of mandel.c, 8x8 image with point granularity

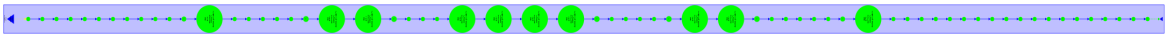


Figure 4: Dependency graph of graphic version of mandel.c, 8x8 image with point granularity, rotated 90 degrees

2.3 Parallelization strategy for Mandeld

To avoid the sequential execution we want to be able to parallelize the process of drawing on the display. If we use tareador to check the task dependency we can observe that the

global variable causing this problem is *X11_ COLOR_ fake* which is used by both lines of code above. Hence we have to come up with a strategy so this variable can be used by all threads at once. Therefore one possible strategy is to create a critical region so there is no datasharing problem. If we use `tareador_disable_object(&X11_COLOR_fake);` and create the corresponding dependency graph with tareador we can see it is now parallel.

3 OpenMP Parallelization

We are now going to try and parallelize the program with the above observations. There are various ways we can try to make the program parallel.

3.1 Task directive

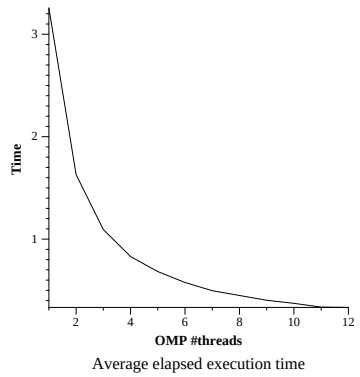
First we are going to use the `task` directive to parallelize the program.

The directives to be added in to have row granularity and point granularity are respectively:

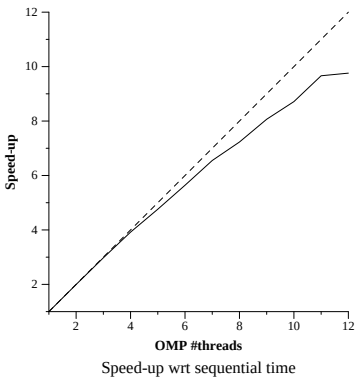
```
#pragma omp parallel
#pragma omp single
    for (row = 0; row < height; ++row) {
#pragma omp task firstprivate(row, col)
        for (col = 0; col < width; ++col) {
...

#pragma omp parallel
#pragma omp single
    for (row = 0; row < height; ++row) {
        for (col = 0; col < width; ++col) {
#pragma omp task firstprivate(row, col)
            {
...
```

We can see that the row based granularity has a strong parallelism, as the speedup grows linearly with the increase of threads. This seems to line up with the hypothesis that the Mandelbrot Set computation is perfectly parallelizable. However we do see an unexpected result when using the point granularity, It seems that due to the number of tasks created, the overhead created to synchronize and fork is too large for it to be worth parallelizing.

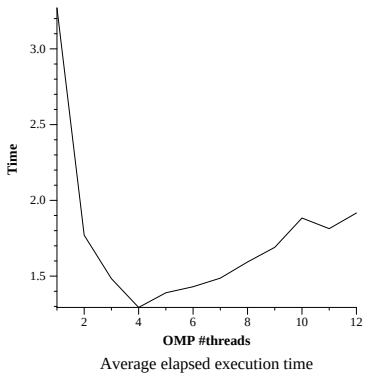


Average elapsed execution time

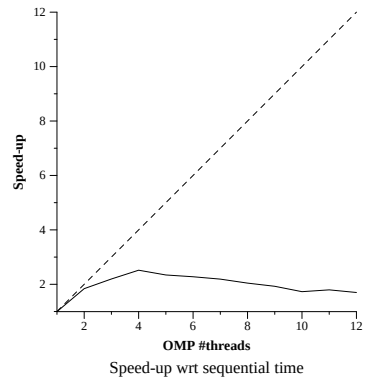


Speed-up wrt sequential time

Figure 5: Task directive with row granularity



Average elapsed execution time



Speed-up wrt sequential time

Figure 6: Task directive with point granularity

3.2 Taskloop directive

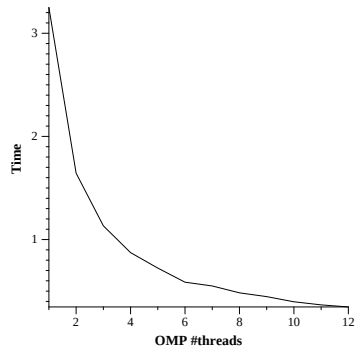
Another possible strategy is using the `taskloop` directive.

In this cases our final code for row and point granularity should look like this:

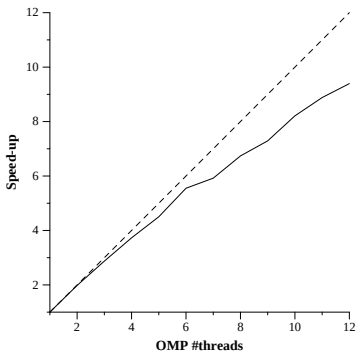
```
#pragma omp parallel
#pragma omp single
#pragma omp taskloop num_tasks(width/8) private(col)
    for (row = 0; row < height; ++row) {
...

#pragma omp parallel
```

```
#pragma omp single
    for (row = 0; row < height; ++row) {
#pragma omp taskloop num_tasks(width/8) firstprivate(row)
        for (col = 0; col < width; ++col) {
...
```

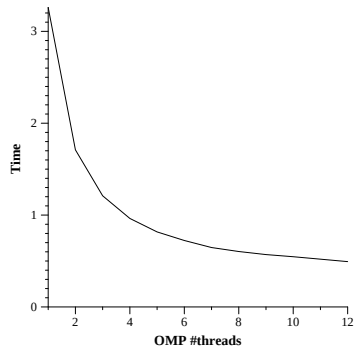


Average elapsed execution time

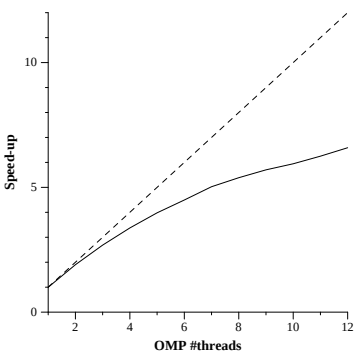


Speed-up wrt sequential time

Figure 7: Taskloop directive with row granularity



Average elapsed execution time



Speed-up wrt sequential time

Figure 8: Taskloop directive with row granularity

Our first decision in this optimization was to choose the grainsize or the number of tasks. We opted for creating width/64 number of tasks. This meant that the computation space was divided in enough chunks so we could have enough load balancing but not too much to avoid excessive overhead. Similar to the last case we can see that using the taskloop with a row granularity possesses the characteristics of strong parallelism. Even though there should be a

better grainsize to further optimize the execution, the same holds true for the point parallelism now as the grainsize is larger.

3.3 For directive

When using the `for` directive we have a few choices to make. First of all, it is necessary to choose a scheduling strategy, in addition it is also recommendable to choose a chunk size, also due to the fact that the points are independent to one and other we can and should add a nowait clause to properly run a parallel process.

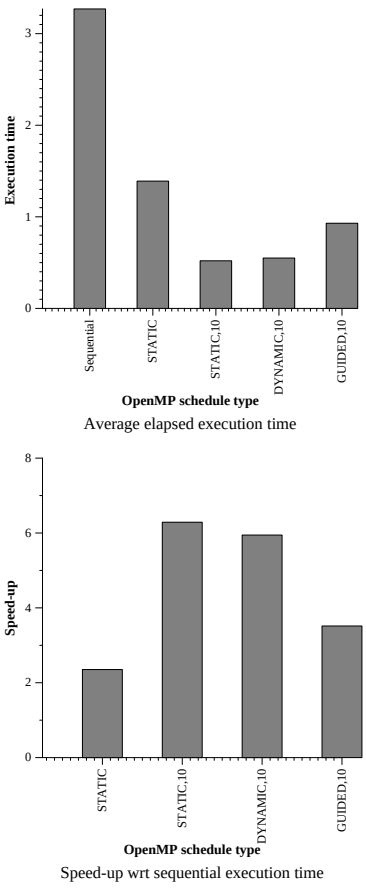


Figure 9: Comparison of different schedulings with row granularity

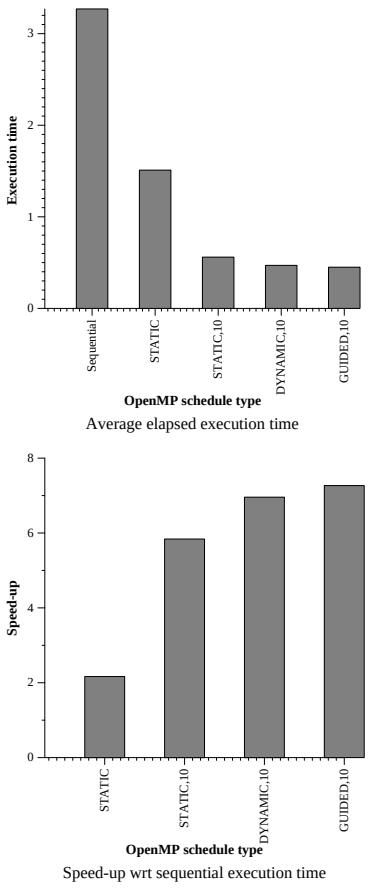


Figure 10: Comparison of different schedulings with point granularity

We compare the speedup and time taken for 5 different strategies. We see that when we consider the row granularity, the slowest one is indeed the sequential execution. What might

be surprising at first sight is the time taken for the static directive without specifying chunk size.

If we analyze the code, we observe that the computational cost of the program is mostly on the while loop inside the main function. We can see that the time taken for each point can range from 1 to the maximum number of iterations. That means that although the program is perfectly parallelizable not all the points take the same time, that is, this is an unbalanced problem.

Returning to our first point, now it seems obvious that the reason for the elapsed time in the static scheduling is caused by load unbalance, as by default it uses chunks of size $\frac{ProblemSize}{Numthreads}$.

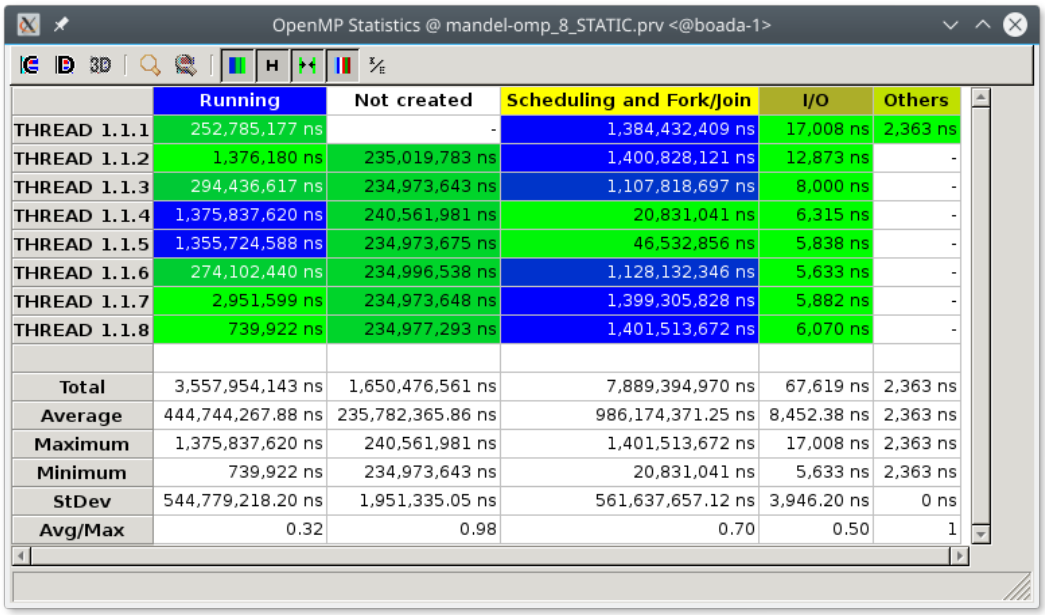
The other three strategies seem to have similar runtimes, with static and dynamic being slightly more effecient than guided. The most probable reason for guided to be slower is that it uses quite large tasks at the beginning, so it is then possible that we have a task unbalance caused by the first few tasks which causes a slight overhead in contrast to the other two faster parallelizations.

Now, in our second case when using the `for` directive with point granularity, we can see that the problem with load balancing still exists but it is fixed for guided, in fact it is faster than the other two. The problem which we mentioned above ceases to exist as we have a quadratic amount of iterations due to being a finer granularity.

We can assert our hypothesis by checking the statistics provided with `Paraver`

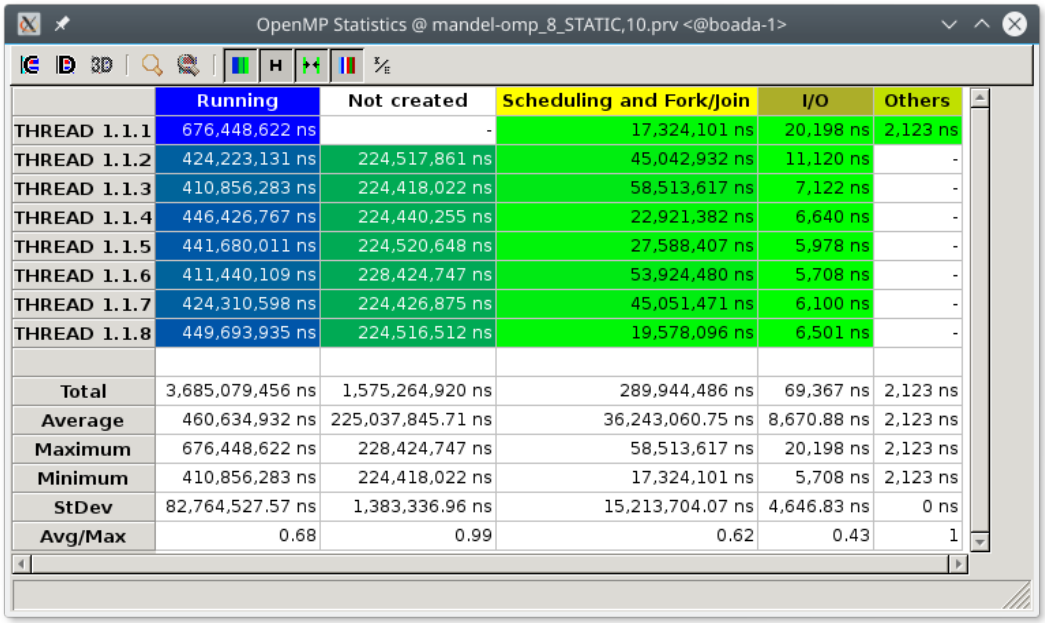
	static	static,10	dynamic,10	guided,10
Running average time per thread (ns)	444744267	460634932	483308567	463425925
Execution unbalance	0.32325	0.68096	0.64334	0.38661
SchedForkJoin (ns)	986174371	36243060	25199996	509464058

Indeed, we see that there is a big discrepancy in the execution unbalance. While the average running time is similar, we can see that the ratio between average time and maximum time is doubled when using static and guided instead of static with smaller chunks or dynamic. This also causes large joining overhead as many threads have to wait before being able to finish. As guided divides tasks better at later iteration, then it is also logical that its synchronization average is lower than the one of static which causes their difference in elapsed time.



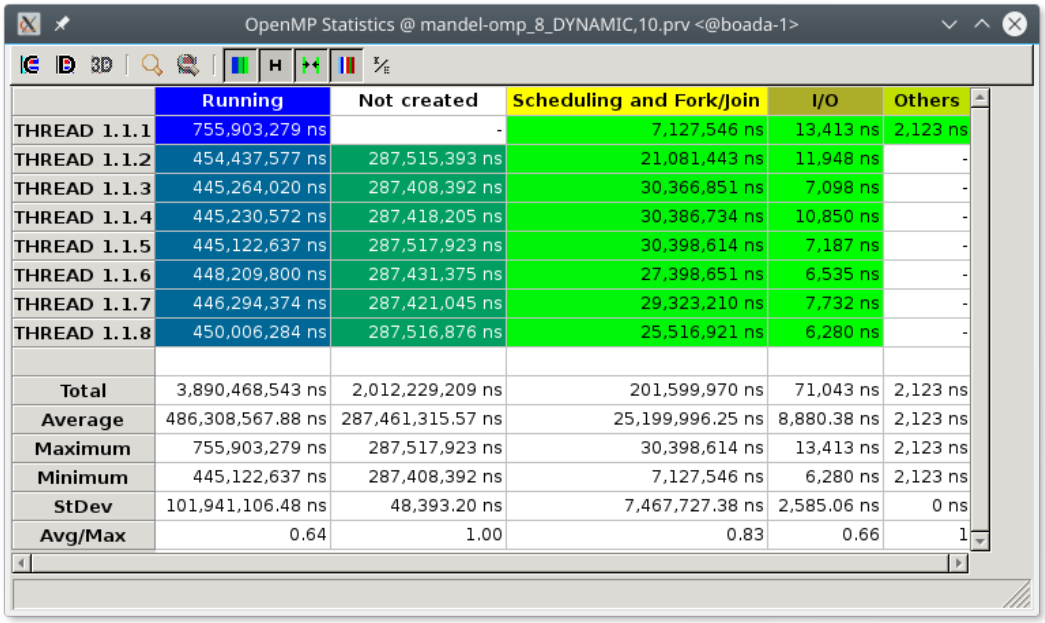
	Running	Not created	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	252,785,177 ns	-	1,384,432,409 ns	17,008 ns	2,363 ns
THREAD 1.1.2	1,376,180 ns	235,019,783 ns	1,400,828,121 ns	12,873 ns	-
THREAD 1.1.3	294,436,617 ns	234,973,643 ns	1,107,818,697 ns	8,000 ns	-
THREAD 1.1.4	1,375,837,620 ns	240,561,981 ns	20,831,041 ns	6,315 ns	-
THREAD 1.1.5	1,355,724,588 ns	234,973,675 ns	46,532,856 ns	5,838 ns	-
THREAD 1.1.6	274,102,440 ns	234,996,538 ns	1,128,132,346 ns	5,633 ns	-
THREAD 1.1.7	2,951,599 ns	234,973,648 ns	1,399,305,828 ns	5,882 ns	-
THREAD 1.1.8	739,922 ns	234,977,293 ns	1,401,513,672 ns	6,070 ns	-
Total	3,557,954,143 ns	1,650,476,561 ns	7,889,394,970 ns	67,619 ns	2,363 ns
Average	444,744,267.88 ns	235,782,365.86 ns	986,174,371.25 ns	8,452.38 ns	2,363 ns
Maximum	1,375,837,620 ns	240,561,981 ns	1,401,513,672 ns	17,008 ns	2,363 ns
Minimum	739,922 ns	234,973,643 ns	20,831,041 ns	5,633 ns	2,363 ns
StDev	544,779,218.20 ns	1,951,335.05 ns	561,637,657.12 ns	3,946.20 ns	0 ns
Avg/Max	0.32	0.98	0.70	0.50	1

Figure 11: Paraver statistics for static scheduling



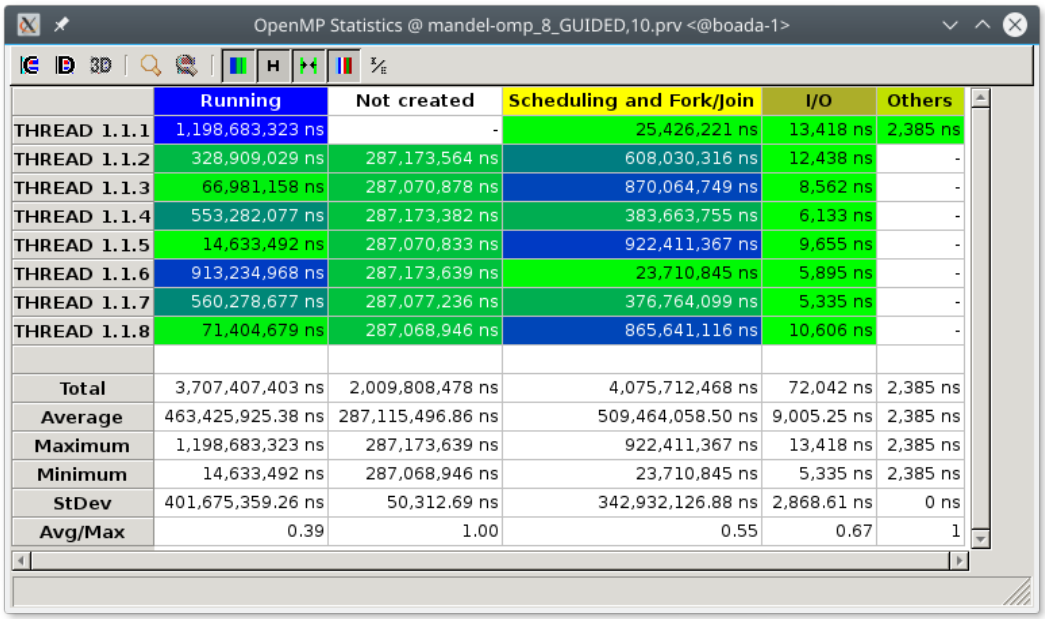
	Running	Not created	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	676,448,622 ns	-	17,324,101 ns	20,198 ns	2,123 ns
THREAD 1.1.2	424,223,131 ns	224,517,861 ns	45,042,932 ns	11,120 ns	-
THREAD 1.1.3	410,856,283 ns	224,418,022 ns	58,513,617 ns	7,122 ns	-
THREAD 1.1.4	446,426,767 ns	224,440,255 ns	22,921,382 ns	6,640 ns	-
THREAD 1.1.5	441,680,011 ns	224,520,648 ns	27,588,407 ns	5,978 ns	-
THREAD 1.1.6	411,440,109 ns	228,424,747 ns	53,924,480 ns	5,708 ns	-
THREAD 1.1.7	424,310,598 ns	224,426,875 ns	45,051,471 ns	6,100 ns	-
THREAD 1.1.8	449,693,935 ns	224,516,512 ns	19,578,096 ns	6,501 ns	-
Total	3,685,079,456 ns	1,575,264,920 ns	289,944,486 ns	69,367 ns	2,123 ns
Average	460,634,932 ns	225,037,845.71 ns	36,243,060.75 ns	8,670.88 ns	2,123 ns
Maximum	676,448,622 ns	228,424,747 ns	58,513,617 ns	20,198 ns	2,123 ns
Minimum	410,856,283 ns	224,418,022 ns	17,324,101 ns	5,708 ns	2,123 ns
StDev	82,764,527.57 ns	1,383,336.96 ns	15,213,704.07 ns	4,646.83 ns	0 ns
Avg/Max	0.68	0.99	0.62	0.43	1

Figure 12: Paraver statistics for static,10 scheduling



	Running	Not created	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	755,903,279 ns	-	7,127,546 ns	13,413 ns	2,123 ns
THREAD 1.1.2	454,437,577 ns	287,515,393 ns	21,081,443 ns	11,948 ns	-
THREAD 1.1.3	445,264,020 ns	287,408,392 ns	30,366,851 ns	7,098 ns	-
THREAD 1.1.4	445,230,572 ns	287,418,205 ns	30,386,734 ns	10,850 ns	-
THREAD 1.1.5	445,122,637 ns	287,517,923 ns	30,398,614 ns	7,187 ns	-
THREAD 1.1.6	448,209,800 ns	287,431,375 ns	27,398,651 ns	6,535 ns	-
THREAD 1.1.7	446,294,374 ns	287,421,045 ns	29,323,210 ns	7,732 ns	-
THREAD 1.1.8	450,006,284 ns	287,516,876 ns	25,516,921 ns	6,280 ns	-
Total	3,890,468,543 ns	2,012,229,209 ns	201,599,970 ns	71,043 ns	2,123 ns
Average	486,308,567.88 ns	287,461,315.57 ns	25,199,996.25 ns	8,880.38 ns	2,123 ns
Maximum	755,903,279 ns	287,517,923 ns	30,398,614 ns	13,413 ns	2,123 ns
Minimum	445,122,637 ns	287,408,392 ns	7,127,546 ns	6,280 ns	2,123 ns
StDev	101,941,106.48 ns	48,393.20 ns	7,467,727.38 ns	2,585.06 ns	0 ns
Avg/Max	0.64	1.00	0.83	0.66	1

Figure 13: Paraver statistics for dynamic,10 scheduling

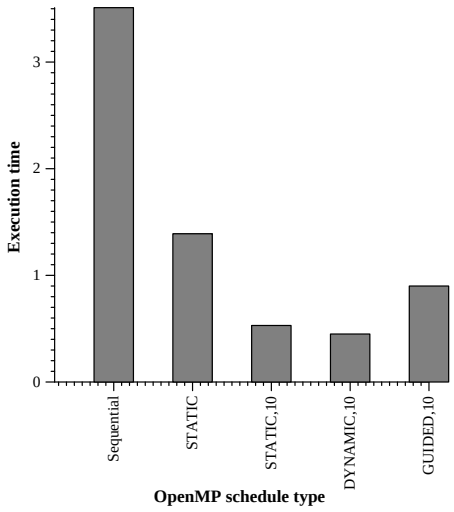


	Running	Not created	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	1,198,683,323 ns	-	25,426,221 ns	13,418 ns	2,385 ns
THREAD 1.1.2	328,909,029 ns	287,173,564 ns	608,030,316 ns	12,438 ns	-
THREAD 1.1.3	66,981,158 ns	287,070,878 ns	870,064,749 ns	8,562 ns	-
THREAD 1.1.4	553,282,077 ns	287,173,382 ns	383,663,755 ns	6,133 ns	-
THREAD 1.1.5	14,633,492 ns	287,070,833 ns	922,411,367 ns	9,655 ns	-
THREAD 1.1.6	913,234,968 ns	287,173,639 ns	23,710,845 ns	5,895 ns	-
THREAD 1.1.7	560,278,677 ns	287,077,236 ns	376,764,099 ns	5,335 ns	-
THREAD 1.1.8	71,404,679 ns	287,068,946 ns	865,641,116 ns	10,606 ns	-
Total	3,707,407,403 ns	2,009,808,478 ns	4,075,712,468 ns	72,042 ns	2,385 ns
Average	463,425,925.38 ns	287,115,496.86 ns	509,464,058.50 ns	9,005.25 ns	2,385 ns
Maximum	1,198,683,323 ns	287,173,639 ns	922,411,367 ns	13,418 ns	2,385 ns
Minimum	14,633,492 ns	287,068,946 ns	23,710,845 ns	5,335 ns	2,385 ns
StDev	401,675,359.26 ns	50,312.69 ns	342,932,126.88 ns	2,868.61 ns	0 ns
Avg/Max	0.39	1.00	0.55	0.67	1

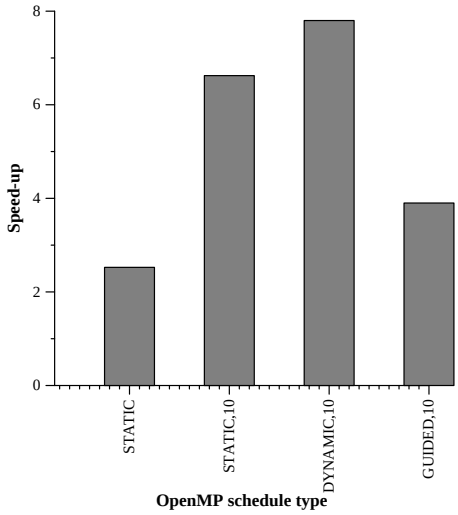
Figure 14: Paraver statistics for guided,10 scheduling

3.4 For directive with Collapse

When we use the `collapse(2)` directive, we are assuming that the two for loops are correctly nested and we are running through a $N * M$ computational space and we will let each thread compute a rectangular space. Hence if we try and compute this we will see a similar behaviour to the `for` directive with row granularity as the division is not as fine as point granularity.



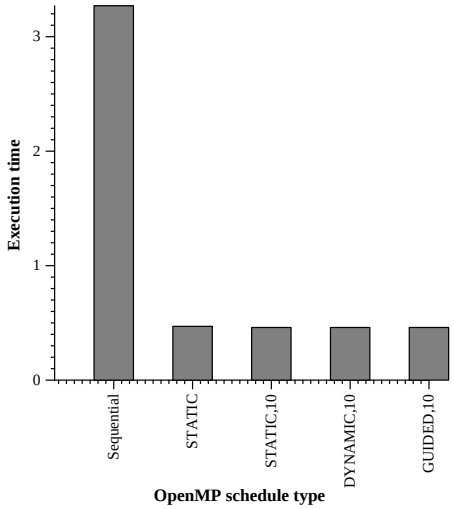
Average elapsed execution time



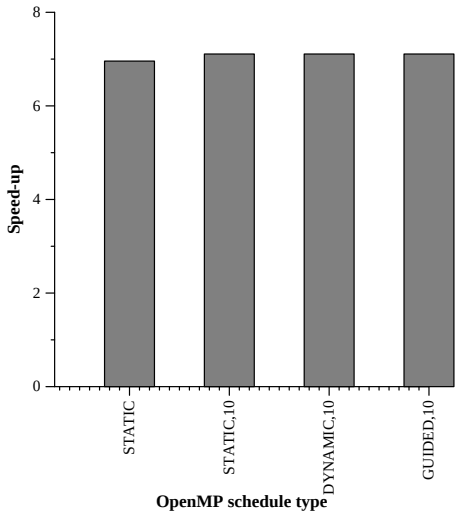
Speed-up wrt sequential execution time

3.5 For directive with task creation

In this new strategy our aim is to create tasks parallelly to reduce its overhead. And indeed if we check the speedup of this strategy it increases all strategies to almost a 7x speedup. That is, compared to the `for` directive we don't have the problem of load unbalancing as we are only using it to create tasks. And also we do not have the problem of excessive overhead obtained from creating $N * N$ tasks as we have them evenly divided.



Average elapsed execution time



Speed-up wrt sequential execution time

4 Conclusion

After analyzing the process of calculation of the mandelbrot set we have discovered several characteristics.
Its calculation is an embarrassingly parallel problem while also being unbalanced. To properly parallelize this problem we have two main steps. First we should divide the workload to several threads and later on compute all the work on each thread.
This means that while the problem is perfectly parallelizable not every strategy tackles the problem correctly and there are other factors to take into consideration, in our case the biggest culprits are load unbalance and fork/synchronization oberhead. Due to the reasons we mentioned we encounter several problems. If we divide into tasks thoughtlessly we might cause a load unbalance (static scheduling). But if we want to avoid load unbalance and use really fine partitions of the problem we might run into a gigantic overhead caused by the creation of tasks (task directive with point granularity, and in a lesser extent taskloop directive with point granularity). All other strategies seem to deal more or less well with this problem, which is to be expected as it is perfectly parallelizable, as a result we see that most strategies have a strong scalability.