LAB 3 DELIVERABLE

PAR 2108

Raül Montoya Pérez
Miguel Ángel Álvarez Vázquez

Introduction

In this laboratory sessions we practiced with different parallelization strategies on a matrix, with row decomposition and point decomposition.

In the first decomposition strategy, we are creating a task for each row of the matrix. On the one hand, we are creating bigger tasks so it can be a problem because we could not have enough parallelization due to the thread's work so the rest threads rest waiting to finish the row work. On the other hand, the overhead created generating tasks is smaller than the point decomposition because the number of tasks created is reduced.

The other decomposition strategy is creating a task for each element of the matrix. With this, the tasks are smaller that the previous strategy but it produces a lot of overhead due to the big number of tasks generated.

We also tested different Open-MP ways to parallelize a for loop with the pragma directives task, taskloop and for.

In addition, we can say that the execution time of the programs are different depending on the Mandelbot.

In order to evaluate both decomposition strategies with task and taskloop directives we will use the submit-strong-omp script which executes the program sequentially and with different values of number of threads from 1 to 12 threads, obtaining the elapsed time and the speed up for all the executions saving each one of the different variables to compare in a text document. It also creates two plots, one for the elapsed time and another for the speed up for the different number of threads.

Finally, to evaluate the strategies with for directive we used the submit-schedule-omp script.
This script executes the program with different schedule options which are STATIC, STATIC 10, DYNAMIC 10 and GUIDED 10. The script generates the plot of the executing time and the speed up of the sequential version and the four schedule options commented before in order to compare them properly.

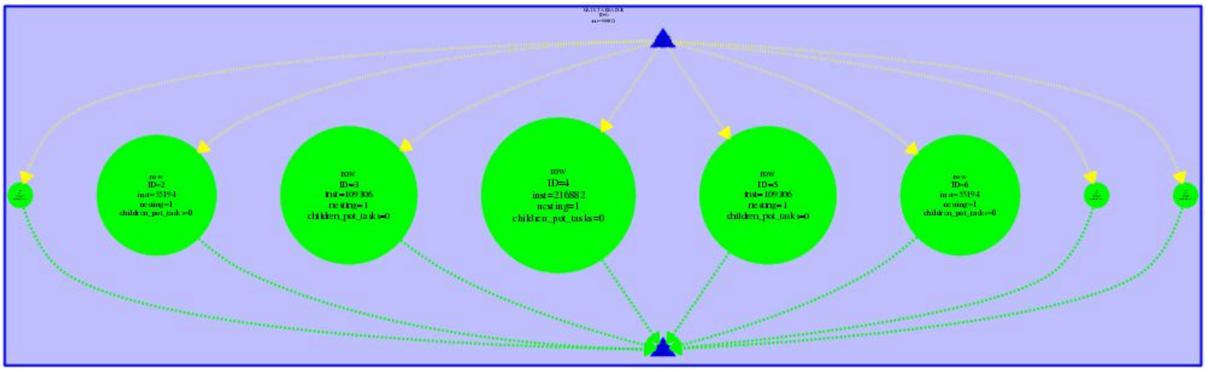
Conclusions

After doing the evaluations we have seen that the best way to execute that program in parallel with 8 threads is task or taskloop with row strategy. With the results obtained, we have seen that depending of the program a decomposition strategy could be better than the other, so we have to measure if creating more tasks in order to increment the parallelism will be worth evaluating the overhead produced creating those tasks.

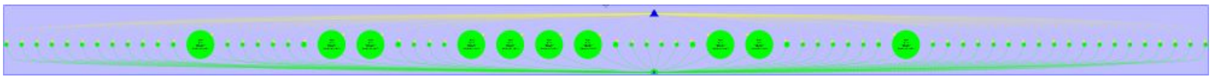
PART I - Task granularity analysis

1. Which are the two most important common characteristics of the task graphs generated for the two task granularities (Row and Point) for the non-graphical version of mandel-tareador? Obtain the task graphs that are generated in both cases for -w 8.

Task graph for Row granularity:



Task graph for Point granularity:



The two most important common characteristics of the task graphs generated for the two task granularities are that in both task graph the tasks don't have the same granularity, because the middle ones have more work than the others, and in neither of the graphs we see dependences between tasks.

Code used:

```
{
        /* Calculate points and save/display */
        for (row = 0; row < height; ++row) {
        tareador_start_task("row");
        for (col = 0; col < width; ++col) {
        //tareador_start_task("point");
        complex z, c;

        z.real = z.imag = 0;

        /* Scale display coordinates to actual region  */
        c.real = real_min + ((double) col * scale_real);
        c.imag = imag_min + ((double) (height-1-row) * scale_imag);
                                /* height-1-row so y axis displays
                                 * with larger values at top
                                 */
```

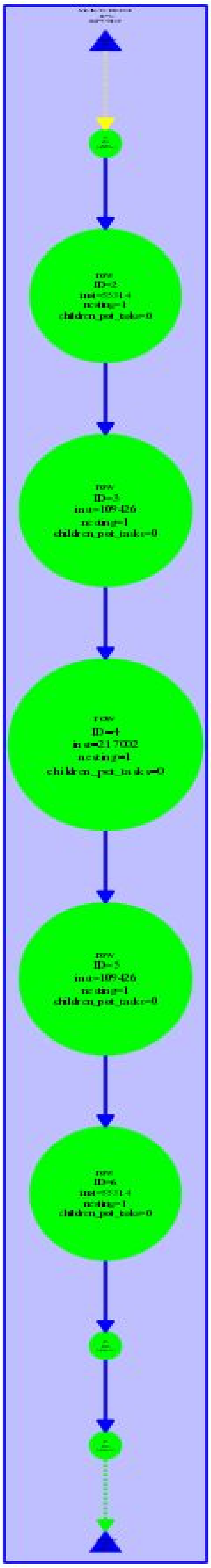
```
        /* Calculate z0, z1, .... until divergence or maximum iterations */
        int k = 0;
        double lengthsq, temp;
        do  {
                temp = z.real*z.real - z.imag*z.imag + c.real;
                z.imag = 2*z.real*z.imag + c.imag;
                z.real = temp;
                lengthsq = z.real*z.real + z.imag*z.imag;
                ++k;
        } while (lengthsq < (N*N) && k < maxiter);

#if _DISPLAY_
        /* Scale color and display point  */
        long color = (long) ((k-1) * scale_color) + min_color;
        if (setup_return == EXIT_SUCCESS) {
                XSetForeground (display, gc, color);
                XDrawPoint (display, win, gc, col, row);
        }
#else
        output[row][col]=k;

#endif
   //tareador_end_task("point");
        }
   tareador_end_task("row");
        }
}
```

2. Which section of the code is causing the serialization of all tasks in mandeld-tareador? How do you plan to protect this section of code in the parallel OpenMP code?

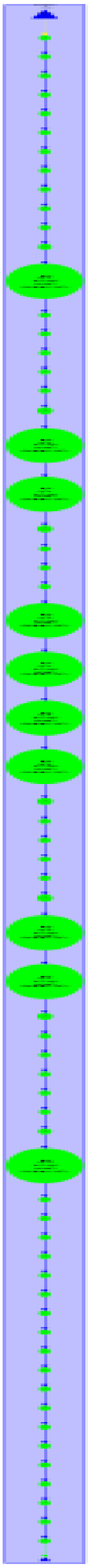
The part when all the pixels are printed one by one on the screen is causing the serialization of all tasks in the program. We will protect it with the directive “#pragma omp critical”.

Task graphs for both granularities in the graphical version are plotted in the next page.

Task graph for Row granularity:



Task graph for Point granularity:



PART II - OpenMP task–based parallelization

For Row code:

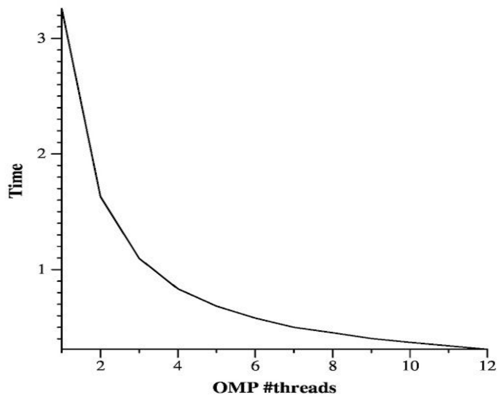
```
#pragma omp parallel
#pragma omp single
for (row = 0; row < height; ++row) {
   #pragma omp task firstprivate(row) private(col)
   for (col = 0; col < width; ++col) {
```

For Point code:

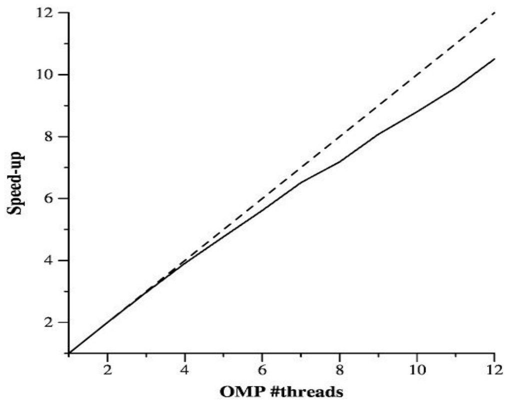
```
#pragma omp parallel
#pragma omp single
for (row = 0; row < height; ++row) {
   for (col = 0; col < width; ++col) {
      #pragma omp task firstprivate(row,col)
```

1. For the Row and Point decompositions of the non-graphical version, include the execution time and speed–up plots obtained in the strong scalability analysis (with -i 10000). Reason about the causes of good or bad performance in each case.

Plots of Row decomposition:

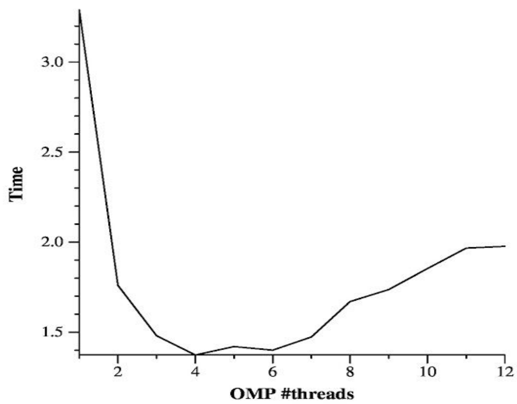


Average elapsed execution time

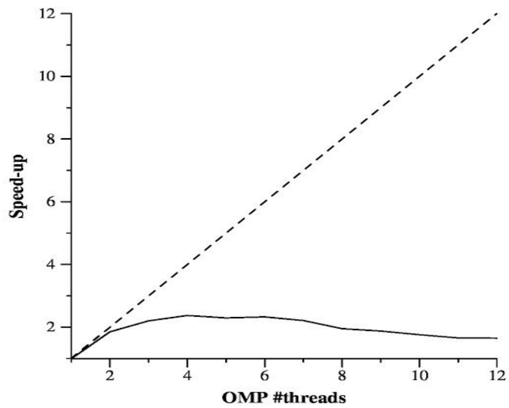


Speed-up wrt sequential time

Plots of Point decomposition:



Average elapsed execution time



Speed-up wrt sequential time

We can see that the speed-up of Point decomposition is lower than the Row decomposition and if more than 6 threads are executing the program, the Point version takes more time to execute that program so the Row decomposition works better. It happens because the Point version creates one task for each for "col", whereas the Row version creates one task for each for "row". For that reason, the Point decomposition generates more overhead than the Row decomposition.

PART III - OpenMP taskloop–based parallelization

For Row code:

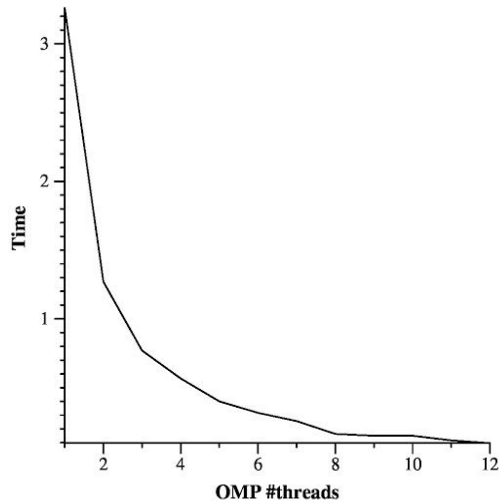
```
#pragma omp parallel
#pragma omp single
#pragma omp taskloop grainsize(8)
for (row = 0; row < height; ++row) {
    for (col = 0; col < width; ++col) {
```

For Point code:

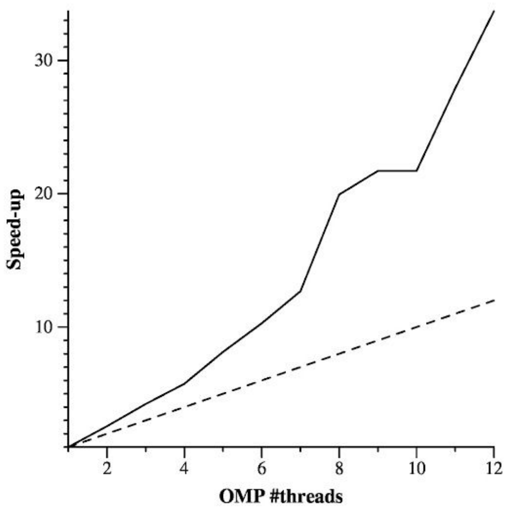
```
#pragma omp parallel
#pragma omp single
for (row = 0; row < height; ++row) {
    #pragma omp taskloop grainsize(8)
    for (col = 0; col < width; ++col) {
```

1. For the Row and Point decompositions of the non-graphical version, include the execution time and speed–up plots obtained in the strong scalability analysis (with -i 10000). Reason about the causes of good or bad performance in each case.

Plots of Row decomposition:

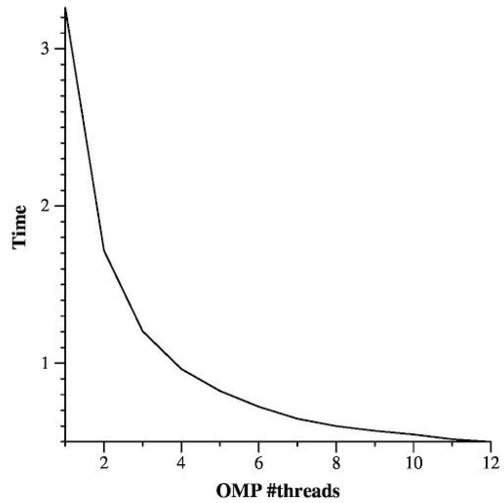


Average elapsed execution time

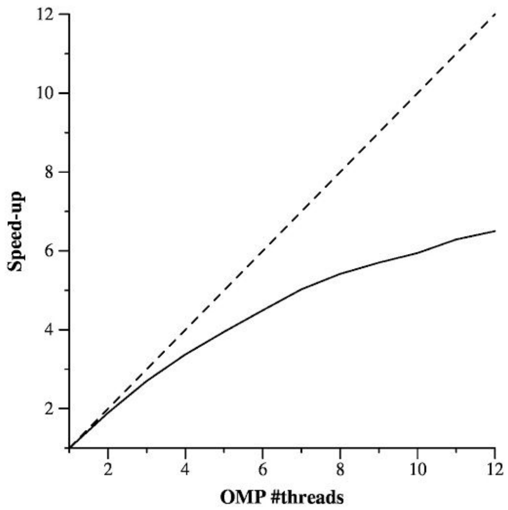


Speed-up wrt sequential time

Plots of Point decomposition:



Average elapsed execution time



Speed-up wrt sequential time

We can see that the speed-up of Point decomposition is lower than the Row decomposition and the execution time of both strategies is lower when the number of threads becomes bigger. Again we can see that the row decomposition works better that the point one for the same reason explained before in the previous exercise.

PART IV - OpenMP for–based parallelization

<u>For Row code:</u>

```
#pragma omp parallel for schedule(runtime) private(row,col)
for (row = 0; row < height; ++row) {
    for (col = 0; col < width; ++col) {
```

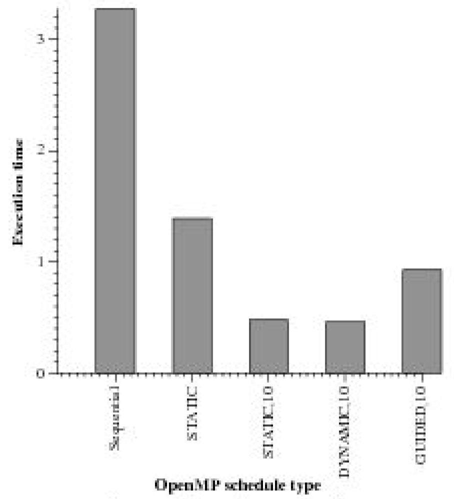
<u>For Point code:</u>

```
#pragma omp parallel private(row)
for (row = 0; row < height; ++row) {
    #pragma omp for schedule(runtime) private(col) nowait
    for (col = 0; col < width; ++col) {
```

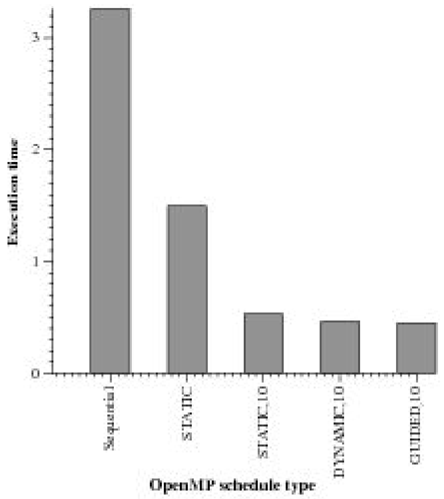
1. For the the Row and Point decompositions of the non-graphical version, include the execution time and speed–up plots that have been obtained for the 4 different loop schedules when using 8 threads (with -i 10000). Reason about the performance that is observed.

Plots of Row decomposition:

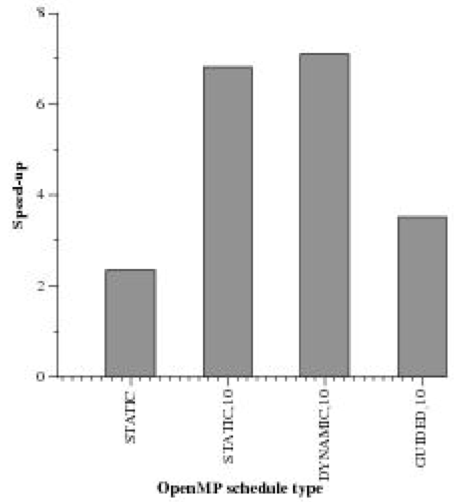
Plots of Point decomposition:



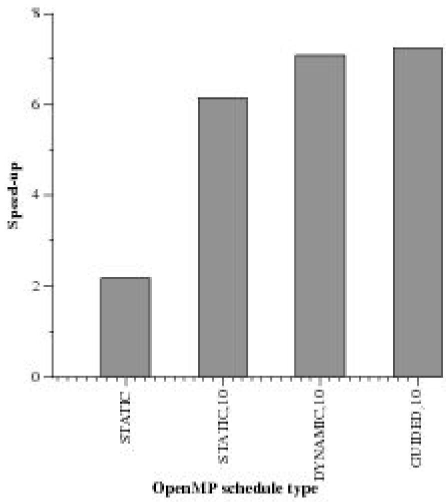
Average elapsed execution time



Average elapsed execution time



Speed-up wrt sequential execution time



Speed-up wrt sequential execution time

From the different graphics, you can see major differences, not only between sequential and any sort of parallelization, but even inside the different structures that can be used when que parallelize the code. In this concrete case, we can assume “Dynamic, 10” schedule is the best one, followed nearby by “Static,10”. This better performance could be explained thanks to the increased chunk that both “static,10” and “dynamic,10” brings us over “static” and “guided,10” respectively.

2. For the Row parallelization strategy, complete the following table with the information extracted from the Extrae instrumented executions (with 8 threads and -i 10000) and analysis with Paraver, reasoning about the results that are obtained. static static,10 dynamic,10 guided,10 Running average time per thread Execution unbalance (average time divided by maximum time) SchedForkJoin (average time per thread or time if only one does)

	static	**static, 10**	**dynamic, 10**	**guided, 10**
Running average time per thread (ns)	439.215.174'75	479.743.460'38	471.594.907	455.207.713'50
Execution unbalance (average time divided by maximum time)	0'32	0'64	0'67	0'50
SchedForkJoin (average time per thread or time if only one does)	976,507,156	55.832.398'50	39.524.737	505.620.130

OpenMP Statistics @ mandel-omp_8_STATIC.prv #2

	Running	Not created	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	229,324,353 ns	-	1,373,924,845 ns	13,956 ns	2,318 ns
THREAD 1.1.2	1,374,280 ns	214,070,379 ns	1,387,805,147 ns	15,666 ns	-
THREAD 1.1.3	286,092,458 ns	216,140,805 ns	1,101,021,704 ns	10,505 ns	-
THREAD 1.1.4	1,369,020,131 ns	213,972,812 ns	20,266,514 ns	6,015 ns	-
THREAD 1.1.5	1,352,621,146 ns	213,972,779 ns	36,666,195 ns	5,352 ns	-
THREAD 1.1.6	273,236,455 ns	214,072,284 ns	1,115,951,412 ns	5,321 ns	-
THREAD 1.1.7	1,300,773 ns	214,067,177 ns	1,387,892,502 ns	5,020 ns	-
THREAD 1.1.8	751,802 ns	213,979,591 ns	1,388,528,933 ns	5,146 ns	-
Total	3,513,721,398 ns	1,500,275,827 ns	7,812,057,252 ns	66,981 ns	2,318 ns
Average	439,215,174.75 ns	214,325,118.14 ns	976,507,156.50 ns	8,372.62 ns	2,318 ns
Maximum	1,369,020,131 ns	216,140,805 ns	1,388,528,933 ns	15,666 ns	2,318 ns
Minimum	751,802 ns	213,972,779 ns	20,266,514 ns	5,020 ns	2,318 ns
StDev	544,245,025.87 ns	742,555.53 ns	558,877,499.45 ns	4,101.23 ns	0 ns
Avg/Max	0.32	0.99	0.70	0.53	1

Schedule Static - 8 threads

OpenMP Statistics @ mandel-omp_8_STATIC,10.prv

	Running	Not created	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	745,122,107 ns	-	21,677,527 ns	21,880 ns	2,095 ns
THREAD 1.1.2	407,524,797 ns	264,223,395 ns	95,064,756 ns	10,661 ns	-
THREAD 1.1.3	411,889,017 ns	264,329,349 ns	90,598,801 ns	6,442 ns	-
THREAD 1.1.4	459,614,978 ns	264,329,996 ns	42,872,372 ns	6,263 ns	-
THREAD 1.1.5	481,220,552 ns	264,240,858 ns	21,356,334 ns	5,865 ns	-
THREAD 1.1.6	436,197,967 ns	264,232,340 ns	66,387,267 ns	6,035 ns	-
THREAD 1.1.7	432,992,619 ns	264,328,959 ns	69,495,878 ns	6,153 ns	-
THREAD 1.1.8	463,385,646 ns	264,222,394 ns	39,206,253 ns	9,316 ns	-
Total	3,837,947,683 ns	1,849,907,291 ns	446,659,188 ns	72,615 ns	2,095 ns
Average	479,743,460.38 ns	264,272,470.14 ns	55,832,398.50 ns	9,076.88 ns	2,095 ns
Maximum	745,122,107 ns	264,329,996 ns	95,064,756 ns	21,880 ns	2,095 ns
Minimum	407,524,797 ns	264,222,394 ns	21,356,334 ns	5,865 ns	2,095 ns
StDev	103,079,067.81 ns	49,657.64 ns	27,031,282.37 ns	5,116.95 ns	0 ns
Avg/Max	0.64	1.00	0.59	0.41	1

Static, 10 - 8 threads

OpenMP Statistics @ mandel-omp_8_DYNAMIC,10.prv

	Running	Not created	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	699,766,774 ns	-	15,773,760 ns	24,196 ns	2,208 ns
THREAD 1.1.2	435,609,911 ns	233,629,280 ns	46,317,187 ns	10,560 ns	-
THREAD 1.1.3	438,827,168 ns	233,685,970 ns	43,047,860 ns	5,940 ns	-
THREAD 1.1.4	433,700,438 ns	233,686,010 ns	48,170,742 ns	9,748 ns	-
THREAD 1.1.5	438,906,124 ns	233,605,097 ns	43,049,552 ns	6,165 ns	-
THREAD 1.1.6	437,926,018 ns	233,669,204 ns	43,965,386 ns	6,330 ns	-
THREAD 1.1.7	454,292,095 ns	233,605,114 ns	27,663,631 ns	6,098 ns	-
THREAD 1.1.8	433,730,728 ns	233,618,242 ns	48,209,778 ns	8,190 ns	-
Total	3,772,759,256 ns	1,635,498,917 ns	316,197,896 ns	77,227 ns	2,208 ns
Average	471,594,907 ns	233,642,702.43 ns	39,524,737 ns	9,653.38 ns	2,208 ns
Maximum	699,766,774 ns	233,686,010 ns	48,209,778 ns	24,196 ns	2,208 ns
Minimum	433,700,438 ns	233,605,097 ns	15,773,760 ns	5,940 ns	2,208 ns
StDev	86,459,723.50 ns	33,923.22 ns	10,872,393.60 ns	5,745.69 ns	0 ns
Avg/Max	0.67	1.00	0.82	0.40	1

Dynamic, 10 - 8 threads

OpenMP Statistics @ mandel-omp_8_GUIDED,10.prv

	Running	Not created	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	807,823,157 ns	-	354,945,107 ns	13,216 ns	2,348 ns
THREAD 1.1.2	71,525,579 ns	230,746,846 ns	860,498,925 ns	12,478 ns	-
THREAD 1.1.3	315,617,731 ns	230,830,927 ns	616,326,559 ns	8,611 ns	-
THREAD 1.1.4	549,819,072 ns	230,829,705 ns	382,129,539 ns	5,512 ns	-
THREAD 1.1.5	14,145,836 ns	230,755,482 ns	917,872,442 ns	10,068 ns	-
THREAD 1.1.6	71,875,459 ns	230,758,462 ns	860,143,372 ns	6,535 ns	-
THREAD 1.1.7	908,000,834 ns	230,827,615 ns	23,950,079 ns	5,300 ns	-
THREAD 1.1.8	902,854,040 ns	230,829,270 ns	29,095,017 ns	5,501 ns	-
Total	3,641,661,708 ns	1,615,578,307 ns	4,044,961,040 ns	67,221 ns	2,348 ns
Average	455,207,713.50 ns	230,796,901 ns	505,620,130 ns	8,402.62 ns	2,348 ns
Maximum	908,000,834 ns	230,830,927 ns	917,872,442 ns	13,216 ns	2,348 ns
Minimum	14,145,836 ns	230,746,846 ns	23,950,079 ns	5,300 ns	2,348 ns
StDev	361,744,789.95 ns	37,651.72 ns	341,113,742.10 ns	3,010.06 ns	0 ns
Avg/Max	0.50	1.00	0.55	0.64	1

Guided, 10 - 8 threads

PART V - Optional

Optional 1: How is the Mandelbrot space computed and what is the performance for the different schedules when using the collapse clause? Look at the following incomplete code:

#pragma omp for collapse(2) schedule(runtime)
for (row = 0; row < height; ++row) {
        for (col = 0; col < width; ++col) {

The space is computed like a vector with the collapse clause, both fors are united in a single one making its condition “i=0; i < height*width; ++i”.

Optional 2: How is the Mandelbrot space computed and what is the performance for the different schedules when combining both for and task? Look at the following incomplete code:

#pragma omp for schedule(runtime)
for (row = 0; row < height; ++row) {
        #pragma omp task
        for (col = 0; col < width; ++col) {

The space is computed like the row decomposition strategy, for each row a task is created.