Third deliverable

Álvaro Martínez Arroyo

Daniel García Romero

par2303

Fall 2015-2016

INDEX

5. Deliverable

5.1 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? Include the task graphs that are generated in both cases for -w 8.

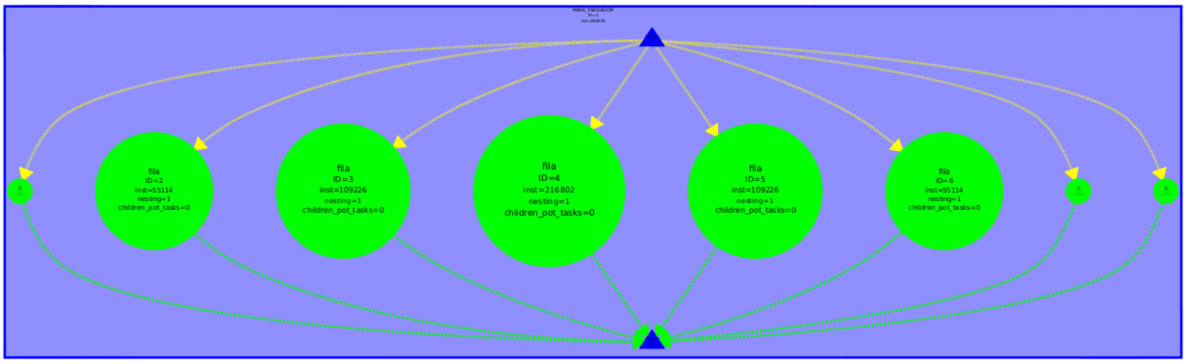


Figure 1. Row task graph

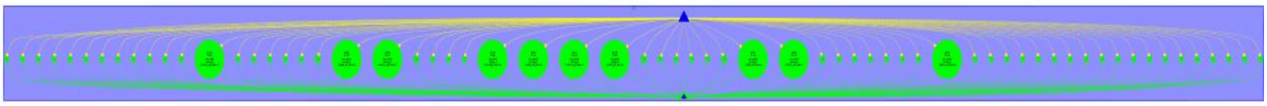


Figure 2. Point task graph

The two most important common characteristics of the task graphs generated are:

→ In neither task graph we see dependences between tasks.

→ In both task graph, the tasks don't have the same granularity (the tasks in the middle have more work than the tasks in the extremes).

2. Which section of the code is causing the serialization of all tasks in mandeld-tareador? How have you protected this section of code in the parallel OpenMP code?

```
#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);
            }
```

Figure 3. Section of code that cause the serialization of tasks

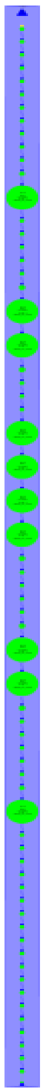


Figure 4. Row task graph with dependences   *Figure 5. Point task graph with dependences*

If we don't comment the section of code of Figure 3 in mandel-tareador.c, we will see in *Tareador* the Figure 4 or the Figure 5, depending on the version of the program that we are executing (Row or Point).

In order to find out what is generating these dependences, we right-click any dependence and select *Dataview* and *Edge*. Then we can see that the global variable that causes the serialization of tasks is X11_COLOR_fake, which is located in the C library Xlib.h.

Inside this Xlib.h, X11_COLOR_fake is used by the XDrawPoint and XSetForeground functions, as we can see in the following code lines:

```
extern int _tareador_fake_XDrawPoint(
    Display*         display,
    Drawable         d,
    GC               gc,
    int              x,
    int              y
) {
  //printf("Fake it: XDrawPoint (color)\n");
  __runtimeTareador_Read(display, 1);
  __runtimeTareador_Write(display, 1);
  __runtimeTareador_Read(&X11_COLOR_fake, 1);
  __runtimeTareador_Write(&X11_COLOR_fake, 1);
  //__runtimeTareador_Write(&(FAKE_DANGEROUS_WINDOW[x*__size+y]), 1);

  return XDrawPoint(display, d, gc, x, y);
}

extern int _tareador_fake_XSetForeground(
    Display*     display,
    GC       gc,
    unsigned long    foreground
) {
  //printf("Fake it: XSetForeground (color)\n");
  __runtimeTareador_Read(display, 1);
  __runtimeTareador_Write(display, 1);
  __runtimeTareador_Read(&foreground, sizeof(unsigned long));
```

```
  __runtimeTareador_Read(&X11_COLOR_fake, 1);
  __runtimeTareador_Write(&X11_COLOR_fake, 1);

  return XSetForeground(display, gc, foreground);
}
```

If we want to protect this section of code in the parallel OpenMP code, we should use the *#pragma omp critical* directive to define a region of mutual exclusion where only one thread can be working at the same time.

5.2 OpenMP task-based parallelization

1. Include the relevant portion of the codes that implement the task-based parallelization for the *Row* and *Point* decompositions (for the non-graphical and graphical options), commenting whatever necessary.

Common directives for Row and Point

The *#pragma omp parallel* creates the parallel region, the threads and one task for each thread.

The *#pragma single* indicates that the following code will be executed by only one thread. At the end of *single*, the other threads see that they have to wait and decide to do pending tasks.

The *#pragma omp critical* defines a region of mutual exclusion where only one thread can be working at the same time.

Row version

#pragma omp task firstprivate(row) private(col)

We want threads have a local copy of the variables *row* and *col*.
The *row* variable has been declared *firstprivate* because its old value is needed, whereas *col* has been declared *private* because after it is initialized to 0.

<u>*Point version*</u>

#pragma omp task firstprivate(row,col)

We want threads have a local copy of the variables *row* and *col*.
The *row* and *col* variables have been declared *firstprivate* because their old values are needed.

2. 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.

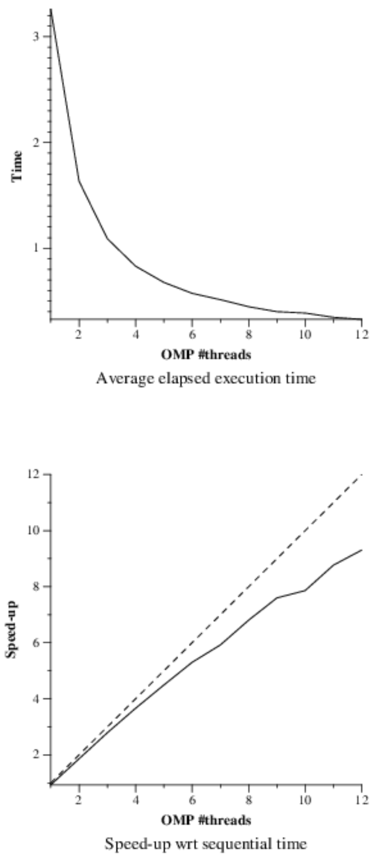


Figure 6. Plots of Row version

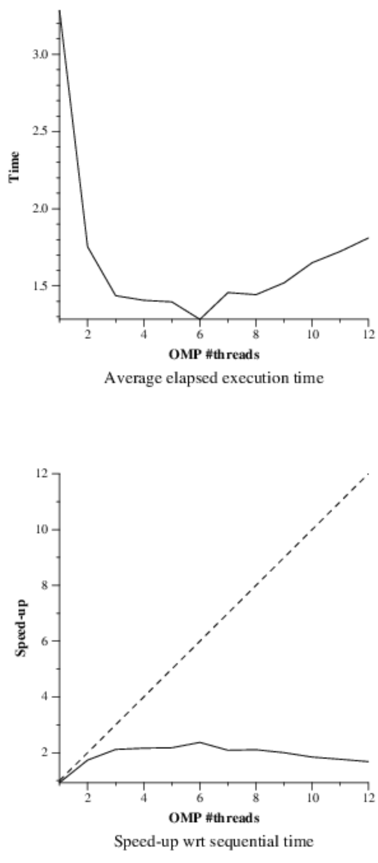


Figure 7. Plots of Point version

The speed-up of *Point* version (Figure 7) is lower than the *Row* version (Figure 6). It means that *Point* version takes more time to be executed, and it happens because *Point* version creates one task for each *for col*, whereas *Row* version creates one task for each *for row*. Therefore, *Point* version creates more tasks than *Row* version, and each task adds overhead to the total execution time.

5.3 OpenMP for-based parallelization

1. Include the relevant portion of the codes that implement the for-based parallelization for the *Row* and *Point* decompositions (for the non-graphical and graphical options), commenting whatever necessary.

Row version

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

We privatize the *row* and *col* variables so that each thread has a local copy of these variables, which are initialized to 0 because the *row* and *col* values are initialized later to this value.

Point version

```
#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) {
      …
```

We privatize the *row* and *col* variables for the same reason but now we added a *nowait* so that the threads don't wait at the end of *for col*.

2. For 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.

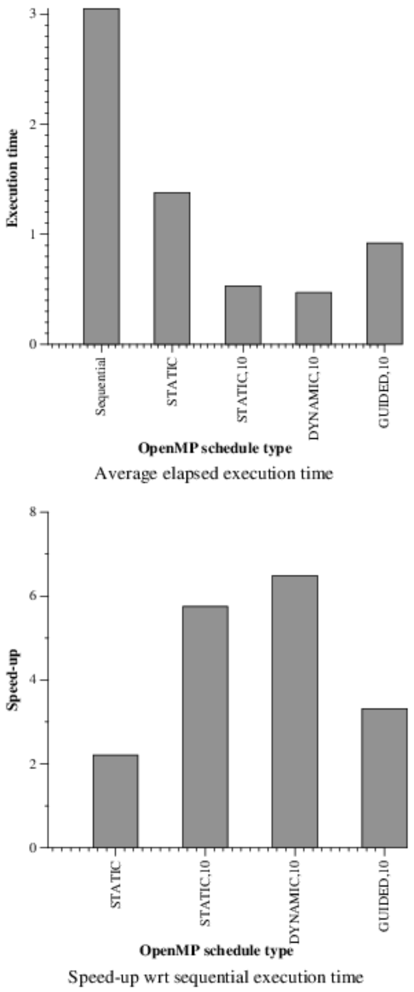


Figure 8. Plots of Row version

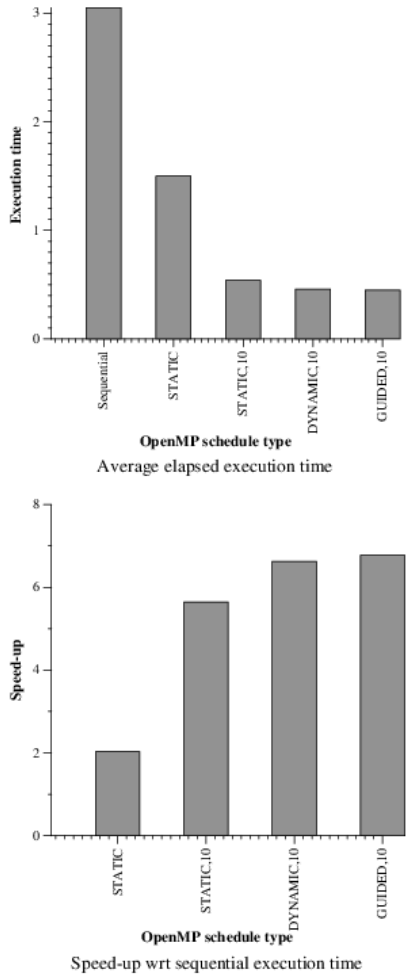


Figure 9. Plots of Point version

In order to obtain these plots, first of all we define the parallelization strategy in the mandel-omp.c. After that, we execute *make mandel-omp* and *./submit-schedule-omp.sh*. This script executes the binary generated with 8 threads using the 4 scheduling strategies and generates the plots with the execution time and speed-up with respect to sequential. If we want to view these plots, we should execute *gs mandel-omp-10000-8-schedule-omp.ps*.

The Mandelbrot code that we can find in mandel-serial.c is an example of non-balanced problem since the amount of work done varies from pixel to pixel.

Taking into account this and the fact that *static* schedule is recommendable to be used when the work-load inside a parallel region is fixed, it seems logical that using *static* schedule we obtain the second worst execution time (the sequential strategy obtains the worst time since no parallelization is used).

The *static,10* schedule takes less time to be executed than *static* because increasing the chunk we reduce the synchronization overhead between threads and this increases the speedup.

The *dynamic* schedule has a better performance than *guided* because of the same reason (using *guided*, the chunk size decreases exponentially with each successive work-assignment to a minimum size specified in the parameter chunk, so the synchronization overhead between threads increases and this reduces the speedup).

3. 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	451,213,619.12 ns	499,377,465.50 ns	498,824,455.12 ns	445,300,905 ns
Execution unbalance (average time divided by maximum time)	0.30	0.91	0.95	0.48
SchedForkJoin (average time per thread or time if only one does)	187,860,594.12 ns	2,512,282.88 ns	1,066,592.25 ns	108,163,126.25 ns

Figure 10. Table with the information extracted from the Paraver traces

In the Figure 10, we can see that the overhead generated by *static,10* and *dynamic,10* are higher than the others schedules. In addition, it seems logical that *static,10* and *dynamic,10* have a good balance because threads have similar amount of work, whereas *static* and *guided,10* are the schedule with more unbalanced execution.

OpenMP Statistics @ mandel-omp_8_STATIC.prv #1 <@boada-1>

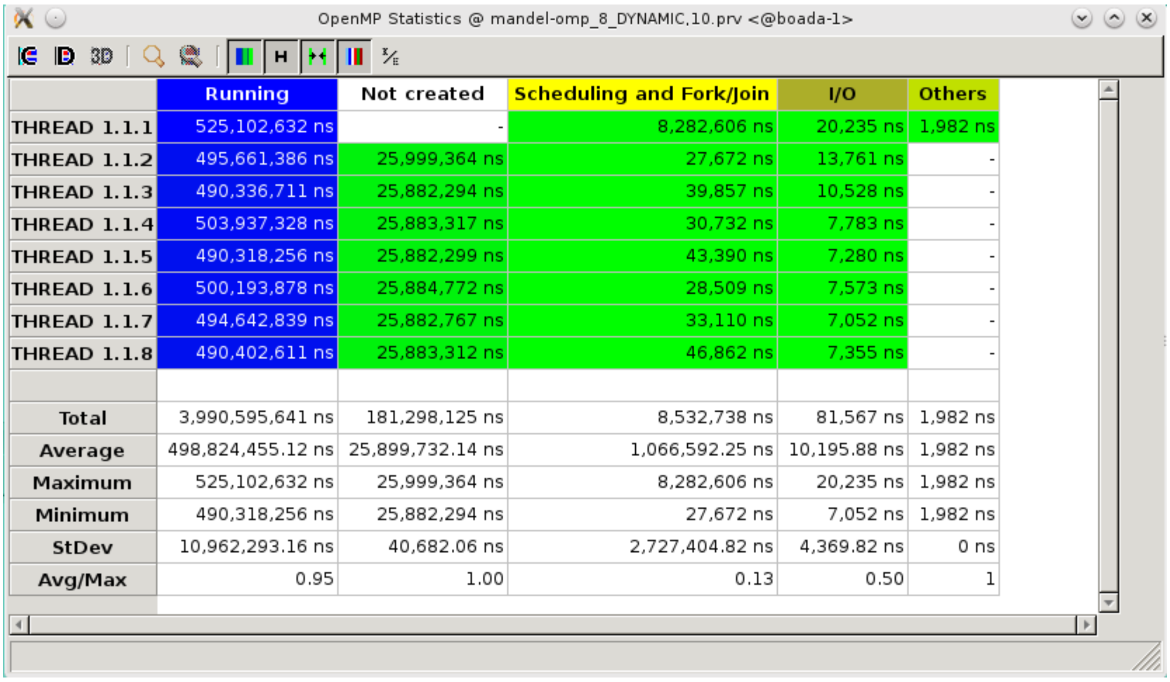
	Running	Not created	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	31,783,828 ns	-	1,502,749,366 ns	19,986 ns	2,313 ns
THREAD 1.1.2	3,111,705 ns	31,744,540 ns	11,454 ns	12,536 ns	-
THREAD 1.1.3	356,979,635 ns	31,732,367 ns	13,632 ns	9,793 ns	-
THREAD 1.1.4	1,499,677,706 ns	31,736,207 ns	39,098 ns	7,263 ns	-
THREAD 1.1.5	1,425,302,997 ns	31,732,369 ns	31,907 ns	8,120 ns	-
THREAD 1.1.6	288,895,075 ns	31,732,177 ns	23,198 ns	7,423 ns	-
THREAD 1.1.7	2,251,412 ns	31,794,287 ns	8,155 ns	7,651 ns	-
THREAD 1.1.8	1,706,595 ns	31,821,313 ns	7,943 ns	7,545 ns	-
Total	3,609,708,953 ns	222,293,260 ns	1,502,884,753 ns	80,317 ns	2,313 ns
Average	451,213,619.12 ns	31,756,180 ns	187,860,594.12 ns	10,039.62 ns	2,313 ns
Maximum	1,499,677,706 ns	31,821,313 ns	1,502,749,366 ns	19,986 ns	2,313 ns
Minimum	1,706,595 ns	31,732,177 ns	7,943 ns	7,263 ns	2,313 ns
StDev	598,299,121.93 ns	33,677.10 ns	496,981,241.84 ns	4,113.90 ns	0 ns
Avg/Max	0.30	1.00	0.13	0.50	1

Figure 11. Paraver trace for mandel-omp with 8 threads and schedule static

OpenMP Statistics @ mandel-omp_8_STATIC,10.prv <@boada-1>

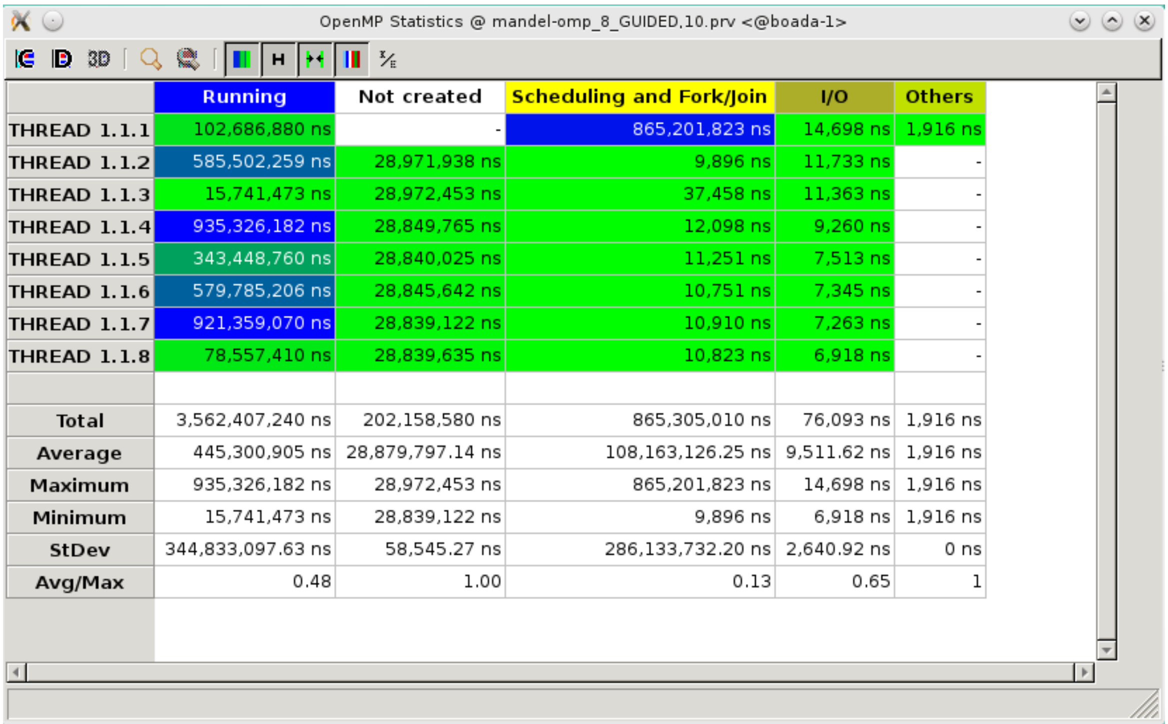
	Running	Not created	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	547,099,095 ns	-	19,792,220 ns	31,521 ns	2,038 ns
THREAD 1.1.2	485,953,593 ns	32,873,337 ns	57,016 ns	9,900 ns	-
THREAD 1.1.3	447,006,494 ns	32,870,634 ns	43,943 ns	9,078 ns	-
THREAD 1.1.4	530,727,927 ns	32,876,680 ns	47,039 ns	8,478 ns	-
THREAD 1.1.5	522,181,711 ns	32,970,826 ns	41,090 ns	9,285 ns	-
THREAD 1.1.6	481,201,640 ns	32,969,744 ns	38,641 ns	9,750 ns	-
THREAD 1.1.7	476,524,383 ns	32,876,670 ns	39,502 ns	9,298 ns	-
THREAD 1.1.8	504,324,881 ns	32,971,549 ns	38,812 ns	9,163 ns	-
Total	3,995,019,724 ns	230,409,440 ns	20,098,263 ns	96,473 ns	2,038 ns
Average	499,377,465.50 ns	32,915,634.29 ns	2,512,282.88 ns	12,059.12 ns	2,038 ns
Maximum	547,099,095 ns	32,971,549 ns	19,792,220 ns	31,521 ns	2,038 ns
Minimum	447,006,494 ns	32,870,634 ns	38,641 ns	8,478 ns	2,038 ns
StDev	30,782,013.65 ns	47,734.65 ns	6,531,204.84 ns	7,366.96 ns	0 ns
Avg/Max	0.91	1.00	0.13	0.38	1

Figure 12. Paraver trace for mandel-omp with 8 threads and schedule static,10



	Running	Not created	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	525,102,632 ns	-	8,282,606 ns	20,235 ns	1,982 ns
THREAD 1.1.2	495,661,386 ns	25,999,364 ns	27,672 ns	13,761 ns	-
THREAD 1.1.3	490,336,711 ns	25,882,294 ns	39,857 ns	10,528 ns	-
THREAD 1.1.4	503,937,328 ns	25,883,317 ns	30,732 ns	7,783 ns	-
THREAD 1.1.5	490,318,256 ns	25,882,299 ns	43,390 ns	7,280 ns	-
THREAD 1.1.6	500,193,878 ns	25,884,772 ns	28,509 ns	7,573 ns	-
THREAD 1.1.7	494,642,839 ns	25,882,767 ns	33,110 ns	7,052 ns	-
THREAD 1.1.8	490,402,611 ns	25,883,312 ns	46,862 ns	7,355 ns	-
Total	3,990,595,641 ns	181,298,125 ns	8,532,738 ns	81,567 ns	1,982 ns
Average	498,824,455.12 ns	25,899,732.14 ns	1,066,592.25 ns	10,195.88 ns	1,982 ns
Maximum	525,102,632 ns	25,999,364 ns	8,282,606 ns	20,235 ns	1,982 ns
Minimum	490,318,256 ns	25,882,294 ns	27,672 ns	7,052 ns	1,982 ns
StDev	10,962,293.16 ns	40,682.06 ns	2,727,404.82 ns	4,369.82 ns	0 ns
Avg/Max	0.95	1.00	0.13	0.50	1

Figure 13. Paraver trace for mandel-omp with 8 threads and schedule dynamic,10



	Running	Not created	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	102,686,880 ns	-	865,201,823 ns	14,698 ns	1,916 ns
THREAD 1.1.2	585,502,259 ns	28,971,938 ns	9,896 ns	11,733 ns	-
THREAD 1.1.3	15,741,473 ns	28,972,453 ns	37,458 ns	11,363 ns	-
THREAD 1.1.4	935,326,182 ns	28,849,765 ns	12,098 ns	9,260 ns	-
THREAD 1.1.5	343,448,760 ns	28,840,025 ns	11,251 ns	7,513 ns	-
THREAD 1.1.6	579,785,206 ns	28,845,642 ns	10,751 ns	7,345 ns	-
THREAD 1.1.7	921,359,070 ns	28,839,122 ns	10,910 ns	7,263 ns	-
THREAD 1.1.8	78,557,410 ns	28,839,635 ns	10,823 ns	6,918 ns	-
Total	3,562,407,240 ns	202,158,580 ns	865,305,010 ns	76,093 ns	1,916 ns
Average	445,300,905 ns	28,879,797.14 ns	108,163,126.25 ns	9,511.62 ns	1,916 ns
Maximum	935,326,182 ns	28,972,453 ns	865,201,823 ns	14,698 ns	1,916 ns
Minimum	15,741,473 ns	28,839,122 ns	9,896 ns	6,918 ns	1,916 ns
StDev	344,833,097.63 ns	58,545.27 ns	286,133,732.20 ns	2,640.92 ns	0 ns
Avg/Max	0.48	1.00	0.13	0.65	1

Figure 14. Paraver trace for mandel-omp with 8 threads and schedule guided,10