Antonio J Cabrera

Paul Gazel-Anthoine

PAR1401

Lab 2

Brief tutorial on OpenMP programming model

Spring 2018-19

Index

OpenMP questionnaire

A) Parallel regions

1.hello.c

1 - 24 times, which is the number of threads working in parallel.

2 - Executing this in the command line:

```
OMP_NUM_THREADS=4 ./1.hello
```

This will set the environment variable to 4, so the default

2.hello.c

1 - It is not what we expect from this program. The variable 'id' should not be global, but private. What is actually happening is that each thread overwrites the value of the variable and by the time each thread prints it, the value is meaningless.

2 - As said before, the id that gets printed for every thread is the latest modification of the variable up to that point by any thread. This means it will vary on each execution.

3.how_many.c

1 - 20 lines;

- The 8 first lines correspond to the 8 threads executing the printf in parallel.
- Inside the loop, the number of threads are set first to 2 and 3, each printing 2 and 3 lines respectively.
- The next 4 lines correspond to the printf after the loop, which explicitly indicates 4 threads have to execute it.
- The last 3 lines come from the omp_num_threads variable being last set to 3 inside the loop, thus using 3 threads to execute this printf.

2 - It returns the number of threads allocated to the execution of that region. Watching at this code, for example, you will receive 1 when invoked from outside of the parallel regions and the number of threads set to work on a task when inside a parallel region.

3.data_sharing.c

1 - The value of variable x after executing each parallel region:

	Shared	Private	Firstprivate	Reduction
X =	It varies *	5	5	125

Table 1: data_sharing values of variable x

Shared: the value of x is varies on execution. This is expected, because all threads are accessing the same variable when reading and writing it, instead of working with a private copy. This way, the result cannot be trusted.

Private: outside the parallel region, the variable is not modified and thus returns the value it had before entering the parallel region.

Firstprivate: similarly to the previous case, each thread has its own version of the variable and, outside the parallel region, the variable remains untouched. The difference is that inside the parallel region, the variable already has the value it had before this point.

Reduction: for each thread, the variable x is a private version with value 0. After assigning the identifier of each thread to x, it does a reduction, adding up all the values that private versions of x has to the value that x had before entering the parallel region (5). In this case, having 16 threads, it sums (0+1+...+ 15) + 5, which is 125.

B) Loop parallelism

1.schedule.c

1 - Iterations assigned to each thread depending on schedule kind.

- Static (default)
  When the execution reaches this part of the code, the scheduler assigns to each thread a number of consecutive iterations equal to N/#threads, where N is the total number of iterations.

- Static (chunk = 2)
  Similarly to the previous case, the scheduler assigns to each thread a specific number of iterations to execute. The main difference is that each thread will get M chunks of two consecutive iterations, M being N/(2*#threads).

- Dynamic (chunk = 2)
  In this case, while there are iterations to execute, the scheduler will assign chunks of two consecutive iterations to threads. This happens each time a thread finishes its previously assigned work.

- Guided (chunk = 2)
  Similarly to the dynamic case, the scheduler will be assigning consecutive iterations to threads, but it has a built-in optimization: over time, it reduces the number of iterations it assigns to threads. It starts at N/#threads and uses the specified chunk as the minimum iterations it will assign. This is done this way because some threads may take more time for certain iterations and it is not efficient to have a balanced distribution of iterations if the total work per thread is not actually balanced.

2.nowait.c

1 - The printf sequence could be any permutation, because even though two threads will be assigned to loop1 (one iteration each), the program will not wait and two threads will be assigned to loop2 and theoretically run in parallel. However, tasks are so small that it is more likely that the first loop will finish before the second loop.

2 - The main difference is that the first loop will be finished before initiating the second loop. This way, the two threads assigned to the first loop will be free to perform the two iterations of the second loop.

3 - In this case, the first loop has two threads assigned, but when the scheduler of the second loop has to distribute work, it assigns an iteration to the two first threads, which are busy in loop1, causing the program to be effectively waiting for loop1. It makes the nowait clause

3.collapse.c

1 - It is the default distribution of iterations/#threads, where iterations is N*N. The main difference in this case is that the two loops are collapsed into one and, instead of having N iterations (first loop), there are N*N iterations, increasing the maximum number of threads to which the scheduler will assign work.

2 - It is not correct, because the j variable is shared between the threads. To correct this, we should add private(j) as a clause.

C) Synchronization

1.datarace.c

1 - It is never correct, because the x variable is shared between the 8 threads and memory *reads* and writes of x are not synchronized.

2 - Two solutions would be:

```
#pragma omp atomic
x++;

#pragma omp critical
x++;
```

2.barrier.c

1 - Yes, for some of the code, we can predict the output. First, the printf instructions will appear unordered. After that, lower thread id's will print first and enter the barrier. When the thread with the highest id enters the barrier, all threads will print the last instruction in no particular order.

3.ordered.c

1 - The *Outside* messages can arrive in any order, but the *Inside* messages will respect the order that would be kept in a sequential execution of the loop.
2 - By using a chunk of size 2 in the scheduler: schedule(dynamic,2)

D) Tasks

1.single.c

1 - Because the workload is not exactly distributed among threads but all threads are executing the entire loop in parallel. The difference is that inside the loop, the *single* clause forces that only one thread will perform the task, even though all threads were supposed to do it. This way, the 4 threads get to the print instruction and we see a burst of different outputs, then they advance to the 1 second sleep and continue the loop, where we see several bursts until there are no more iterations to perform.

2.fibtasks.c

1 - Because the parallel region has not been defined.

2 - Code:

```
#pragma omp parallel
#pragma omp single
while( p != NULL ) {
  #pragma omp task firstprivate(p)
  {
      printf(...);
      processwork(p);
  }
  p = p->next;
}
```

***The rest of the code remains the same**

3.synchtasks.c

1 -

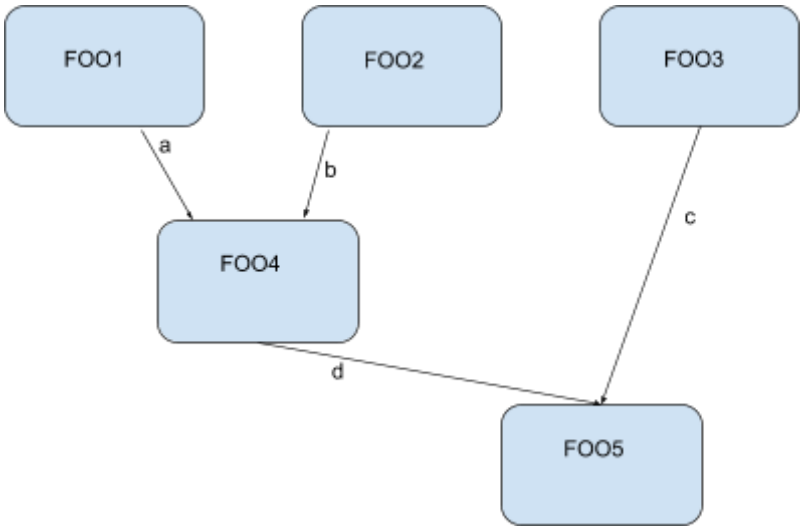

Figure 1: drawing of *synchtasks* dependence graph

2 - Code:

```
#pragma omp parallel
#pragma omp single
{
printf("Creating task foo1\n");
#pragma omp task
foo1();

printf("Creating task foo2\n");
#pragma omp task
foo2();

printf("Creating task foo3\n");
#pragma omp task
foo3();

#pragma omp taskwait
printf("Creating task foo4\n");
#pragma omp task
foo4();

#pragma omp taskwait
printf("Creating task foo5\n");
#pragma omp task
foo5();
}
```

4.taskloop.c

1 -

- *Grainsize* establishes the minimum granularity for each thread. In this case, setting the grain size to 5, it assigns at least 5 iterations to each working thread. We see that only two threads get assignments of 6 iterations.

- Clause *num_tasks(n)* will select the minimum between the number of iterations and n and create that number of tasks. In this case, theoretically, it creates 5 tasks that get distributed among the 4 threads available. The first thread available is the one that picks up the remaining task. Actually, this execution is so fast that sometimes the task management overhead takes longer than the thread computation and we see the same thread getting most of the assignments. This does not happen if we, for example, add ‘sleep(1)’ inside the loop.

2 - When the taskloop with *grainsize* = 5 establishes that two threads are doing all the tasks, the *nogroup* clause allows for the rest of them to go on executing the next instructions in the parallel region. This way, the second loop is computed at the same time than the first, using the rest of the threads available respecting the directives of the second loop.

Observing overheads

A) Results and conclusions

1. Thread creation and termination

# threads	Overhead (ms)	Overhead per thread
2	2.1685	1.0842
3	1.7074	0.5691
4	1.5487	0.3872
5	1.6943	0.3389
6	2.0289	0.3382
7	2.0882	0.2983
8	2.2361	0.2795
9	2.3136	0.2571
10	2.3397	0.234
11	2.3359	0.2124
12	2.5026	0.2086
13	2.6704	0.2054
14	2.632	0.188
15	2.5867	0.1724
16	2.6717	0.167
17	3.044	0.1791
18	3.0905	0.1717
19	3.0591	0.161
20	3.1105	0.1555
21	3.0919	0.1472
22	3.135	0.1425
23	3.1091	0.1352
24	3.1242	0.1302

Table 2: pi_omp_parallel overhead and thread data (1 iteration)

As seen in figures 2 and 3, the slope is steeper at the beginning and it flattens overtime. This means that the overhead of creating and terminating threads increases at a higher rate for a small amount of threads and almost does not when the number of threads is high.

We see that the order of magnitude for the overhead for each individual thread in the parallel region is tenths of milliseconds.

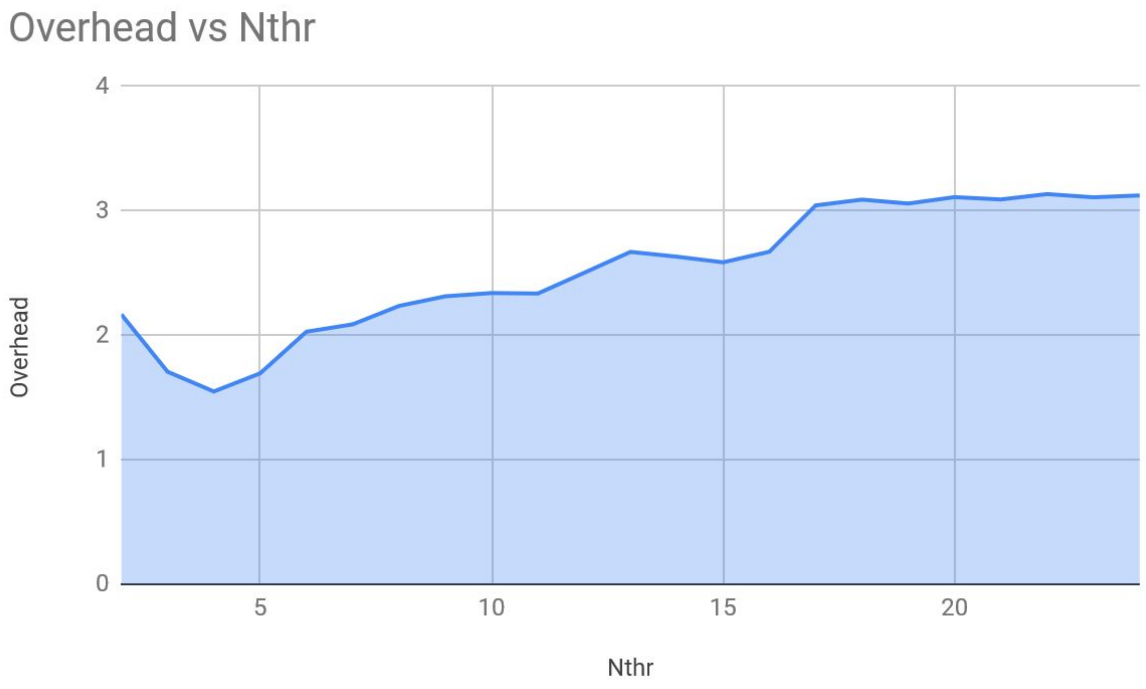

Figure 2: pi_omp_parallel, creation/termination of threads overhead, per #threads (1 iteration)

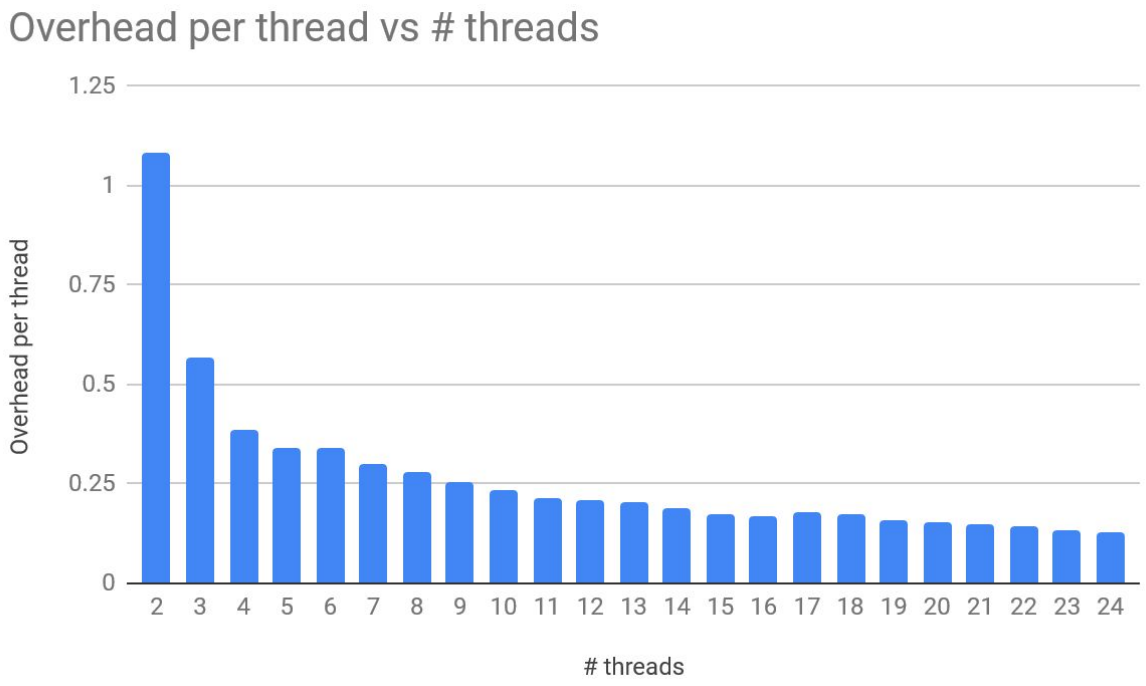

Figure 3: pi_omp_parallel, creation/termination of threads overhead per thread, per #threads

2. Task creation and synchronization

In this case, we see in figures 4 and 5 that the overhead of creating/synchronising tasks is linear to the number of tasks.

The order of magnitude for the overhead of creating/synchronising each individual task is around a tenth of millisecond.

# Tasks	Overhead (ms)	Overhead per task
2	0.1991	0.0995
4	0.5248	0.1312
6	0.7789	0.1298
8	1.031	0.1289
10	1.2827	0.1283
12	1.5342	0.1278
14	1.7855	0.1275
16	2.0373	0.1273
18	2.2916	0.1273
20	2.5384	0.1269
22	2.7895	0.1268
24	3.0412	0.1267
26	3.2912	0.1266
28	3.5426	0.1265
30	3.7923	0.1264
32	4.0467	0.1265
34	4.2858	0.1261
36	4.5524	0.1265
38	4.801	0.1263
40	5.0434	0.1261
42	5.3012	0.1262
44	5.5338	0.1258
46	5.8045	0.1262
48	6.0554	0.1262
50	6.3098	0.1262
52	6.5569	0.1261
54	6.8111	0.1261
56	7.0621	0.1261
58	7.3144	0.1261
60	7.5596	0.126
62	7.8193	0.1261
64	8.0682	0.1261

Table 3: pi_omp_tasks overhead and task data (10 iterations, 1 thread)

Figure 4: pi_omp_tasks, creation/termination of tasks overhead per task, per #tasks

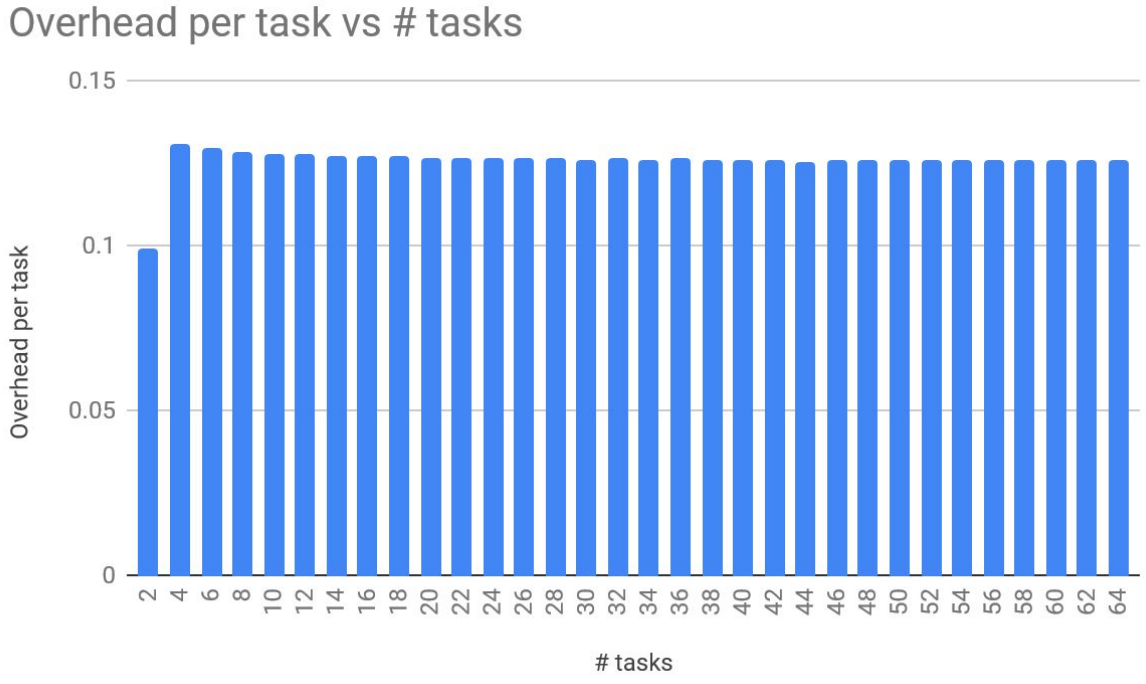

Figure 5: pi_omp_tasks, creation/termination of tasks overhead per task, per #tasks

All the upcoming paraver traces have been executed with 100 000 iterations and 8 threads.

Critical

From the trace obtained by the execution of pi_omp_critical (figures 6, 7 and 8) we can see that the use of the clause critical in this situation is really inefficient due to the fact that every thread is trying to update the variable “sum”. Each threads spends around 25% of the execution in “Synchronisation” mode waiting for the variable “sum” to be unlocked against only between 1 and 2% in “Running” mode.

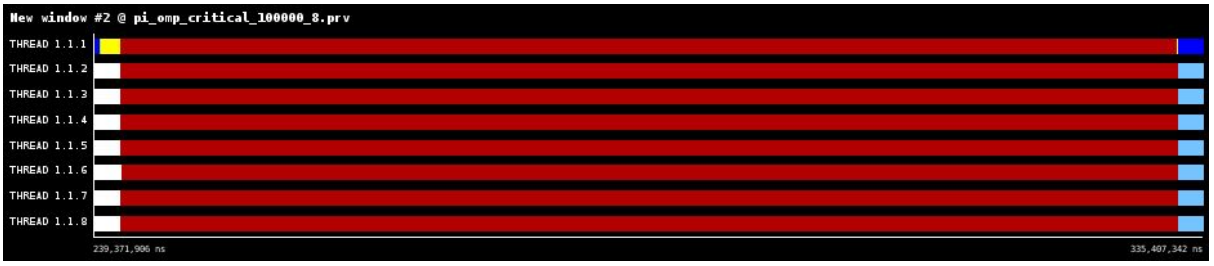

Figure 6: Paraver obtained from the execution of pi_omp_critical

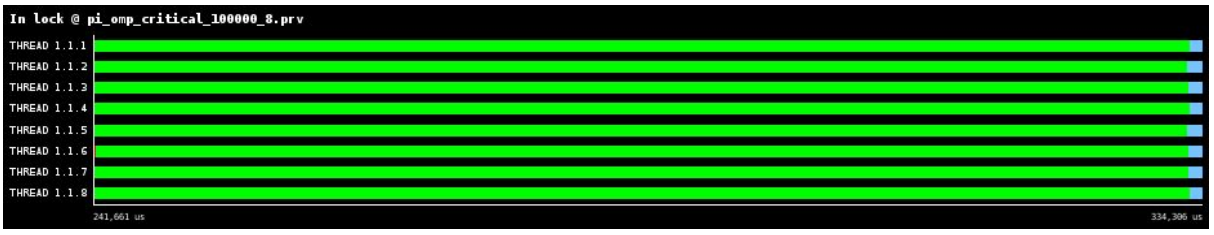

Figure 7:Paraver obtained from the execution of pi_omp_critical with the configuration “in_critical”

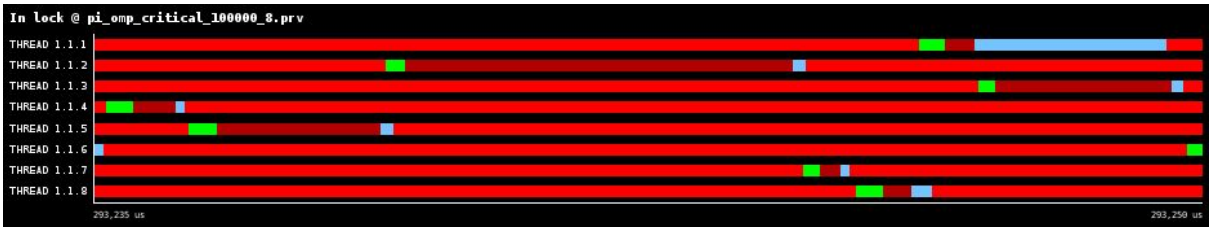

Figure 8: Zoom of the paraver obtained from the execution of pi_omp_critical with the configuration “in_critical”

	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	78.09	0	20.48	0.38	1.06	0
THREAD 1.1.2	1.86	72.52	24.46	0	1.16	0
THREAD 1.1.3	2.4	72.48	23.9	0	1.22	0
THREAD 1.1.4	1.78	72.52	24.54	0	1.17	0
THREAD 1.1.5	1.75	72.53	24.56	0	1.15	0
THREAD 1.1.6	1.81	72.55	24.48	0	1.15	0
THREAD 1.1.7	1.77	72.53	24.54	0	1.15	0
THREAD 1.1.8	2.92	72.53	23.39	0	1.15	0
Total	92.39	507.67	190.35	0.38	9.22	0
Average	11.55	72.52	23.79	0.38	1.15	0
Maximum	78.09	72.55	24.56	0.38	1.22	0
Minimum	1.75	72.48	20.48	0.38	1.06	0
Stdev	25.15	0.02	1.31	0	0.04	0
Avg/Max	0.15	1	0.97	1	0.95	1

Table 4: 2D state profile obtained from the execution of pi_omp_critical (in %)

Atomic

As we can see in table 5 and figure 9, for this example atomic works way better, we obtain a really small percentage of “Synchronization” compared to “Running” mode. But the “Running” mode is a little bit longer than using Critical on average.

	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	99.5	0	0.02	0.47	0.01	0
THREAD 1.1.2	2.25	97.63	0.11	0	0	0
THREAD 1.1.3	2.22	97.63	0.15	0	0	0
THREAD 1.1.4	1.18	97.64	1.19	0	0	0
THREAD 1.1.5	2.25	97.64	0.11	0	0	0
THREAD 1.1.6	2.21	97.66	0.12	0	0	0
THREAD 1.1.7	2.36	97.64	0	0	0	0
THREAD 1.1.8	2.25	97.64	0.11	0	0	0
Total	114.21	683.48	1.81	0.47	0.03	0
Average	14.28	97.64	0.23	0.47	0	0
Maximum	99.5	97.66	1.19	0.47	0.01	0
Minimum	1.18	97.63	0	0.47	0	0
Stdev	32.21	0.01	0.37	0	0	0
Avg/Max	0.14	1	0.19	1	0.62	1

Table 5: state profile obtained from the execution of pi_omp_atomic (in %)

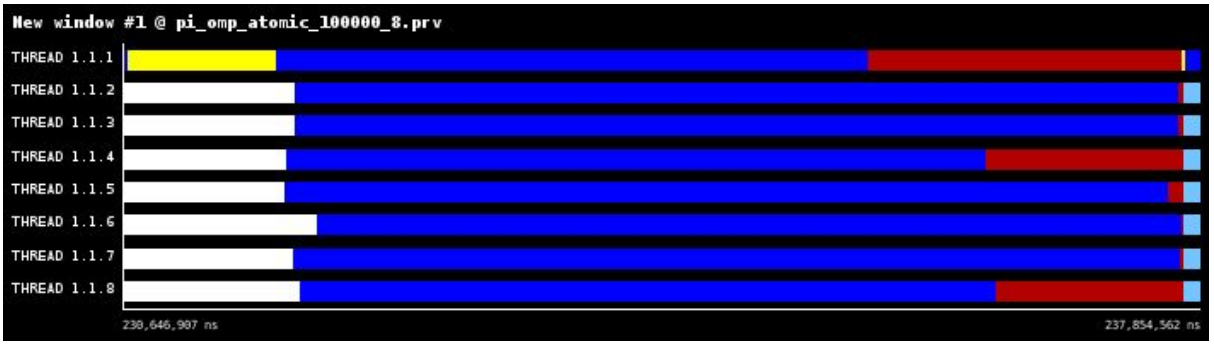

Figure 9: Zoom on the parallel section of the paraver obtained from the execution of pi_omp_atomic

Reduction

For the pi_omp_reduction trace (figure 10) we see that the parallel part is considerably reduced we go from an average of 1-2% of the global run time in the use of the clauses atomic and critical to an average of 0.1%. As “running” mode, the percentage of time in “Synchronization” mode has reduce, but no as considerably as for the running time. We can clearly conclude that the use of reduction is, in this case way better than the use of the clauses atomic or critical.

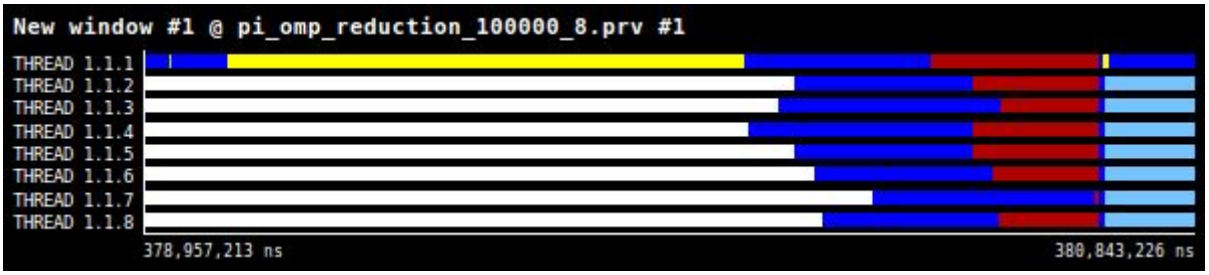

Figure 10: Zoom on the parallel section of the paraver obtained from the execution of pi_omp_reduction

	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	99.72	0	0.07	0.21	0	0
THREAD 1.1.2	0.09	99.85	0.06	0	0	0
THREAD 1.1.3	0.11	99.84	0.05	0	0	0
THREAD 1.1.4	0.11	99.83	0.06	0	0	0
THREAD 1.1.5	0.09	99.85	0.06	0	0	0
THREAD 1.1.6	0.09	99.86	0.05	0	0	0
THREAD 1.1.7	0.11	99.89	0	0	0	0
THREAD 1.1.8	0.09	99.87	0.05	0	0	0
Total	100.39	698.99	0.39	0.21	0.02	0

Average	12.55	99.86	0.05	0.21	0	0
Maximum	99.72	99.89	0.07	0.21	0	0
Minimum	0.09	99.83	0	0.21	0	0
Stdev	32.95	0.02	0.02	0	0	0
Avg/Max	0.13	1	0.71	1	0.7	1

Table 6: state profile obtained from the execution of pi_omp_reduction (in %)

Sumlocal

The trace obtained with the execution of pi_omp_sumlocal, seen in figure 11, allows us to see that as reduction, sumlocal seems to manage better the parallel execution of this program than the use of the clauses atomic and critical. But it looks a little less efficient than reduction in this scenario.

	Running	**Not created**	**Synchronization**	**Scheduling and Fork/Join**	**I/O**	**Others**
THREAD 1.1.1	99.56	0	0.12	0.31	0.01	0
THREAD 1.1.2	0.16	99.76	0.07	0	0	0
THREAD 1.1.3	0.12	99.79	0.09	0	0	0
THREAD 1.1.4	0.12	99.75	0.13	0	0	0
THREAD 1.1.5	0.16	99.75	0.08	0	0	0
THREAD 1.1.6	0.16	99.83	0.01	0	0	0
THREAD 1.1.7	0.16	99.77	0.07	0	0	0
THREAD 1.1.8	0.12	99.86	0.02	0	0	0
Total	100.57	698.51	0.58	0.31	0.03	0
Average	12.57	99.79	0.07	0.31	0	0
Maximum	99.56	99.86	0.13	0.31	0.01	0
Minimum	0.12	99.75	0.01	0.31	0	0
Stdev	32.88	0.04	0.04	0	0	0
Avg/Max	0.13	1	0.58	1	0.46	1

Table 7: state profile obtained from the execution of pi_omp_sumlocal (in %)

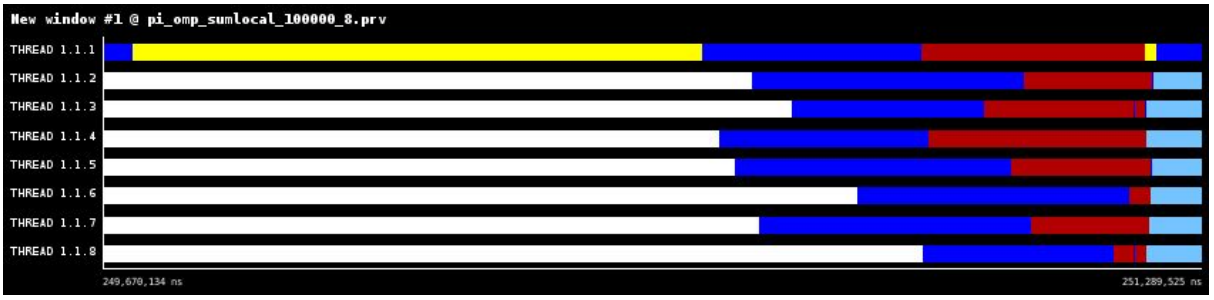

Figure 11: Zoom on the parallel section of the paraver obtained from the execution of pi_omp_sumlocal