LAB 1 DELIVERABLE

PAR 2108

Raül Montoya Pérez
Miguel Ángel Álvarez Vázquez

Node architecture and memory

1. Draw and briefly describe the architecture of the computer in which you are doing this lab session (number of sockets, cores per socket, threads per core, cache hierarchy size and sharing, and amount of main memory).

Number of sockets: 2
Cores per socket: 6
Threads per core: 2
Cache hierarchy size and sharing:

- All the cores in a socket share a L3 cache of 12 MB
- Each core has a L2 cache of 256 KB
- Each core has a L1 instruction cache of 32 KB and L1 data cache of 32KB.

Amount of main memory: 24GB (12 GB per socket)

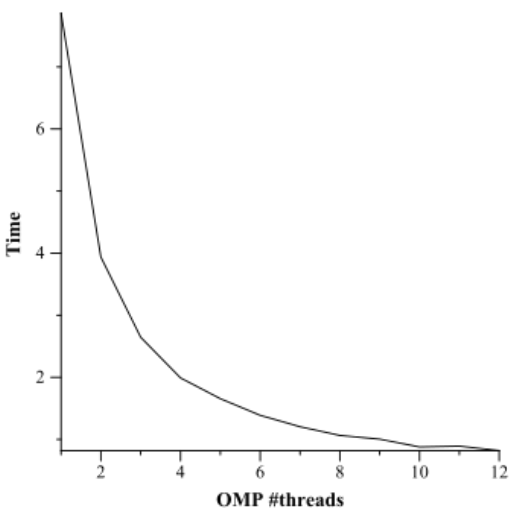
Timing sequential and parallel executions

2. Describe what do you need to add to your program to measure the elapsed execution time between a pair of points in the program, clearly indicating the library header file that needs to be included, the library functions that need to be invoked, the data structure and its fields.

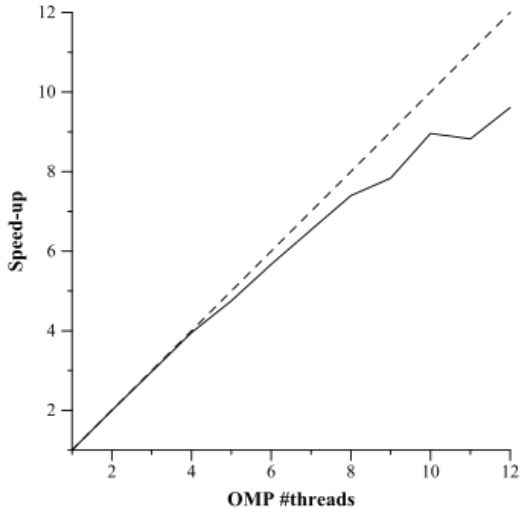
We have to include the header “#include <sys/time.h>” to use the functions gettimeofday() and put the commands START_COUNT_TIME and STOP_COUNT_TIME before and after the interval we want to measure. The fields that timeval has are tv_sec(long int) and rv_usec(long int). The first represents the seconds of the time, and the second represents the microseconds of the time.

3. Plot the speed–up obtained when varying the number of threads (strong scalability) and problem size (weak scalability) for pi omp.c. Reason about how the scalability of the program.

Strong scalability:



Min elapsed execution time



Speed-up wrt sequential time

We can see that when the number of threads increases, the execution time decreases and the speed-up increases. This is what happens in strong scalability.

Weak scalability:

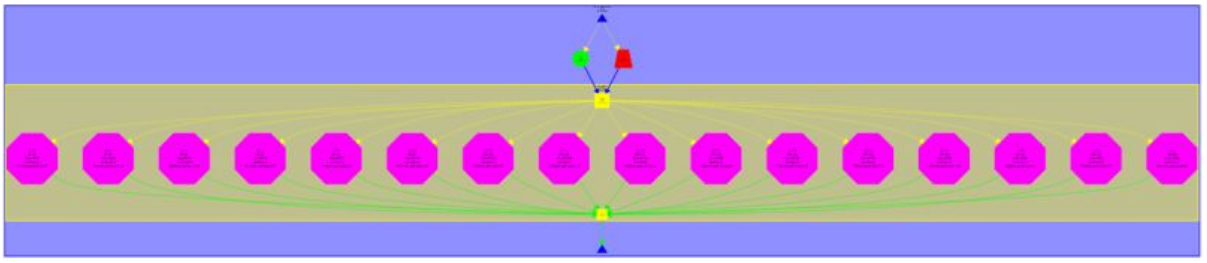
When we increase the size of the problem, number of threads and the execution time increase but the speed-up decreases. The best thing is that the speed-up become constant. In the figure seems to be constant, except one moment, when we change to 8 threads.

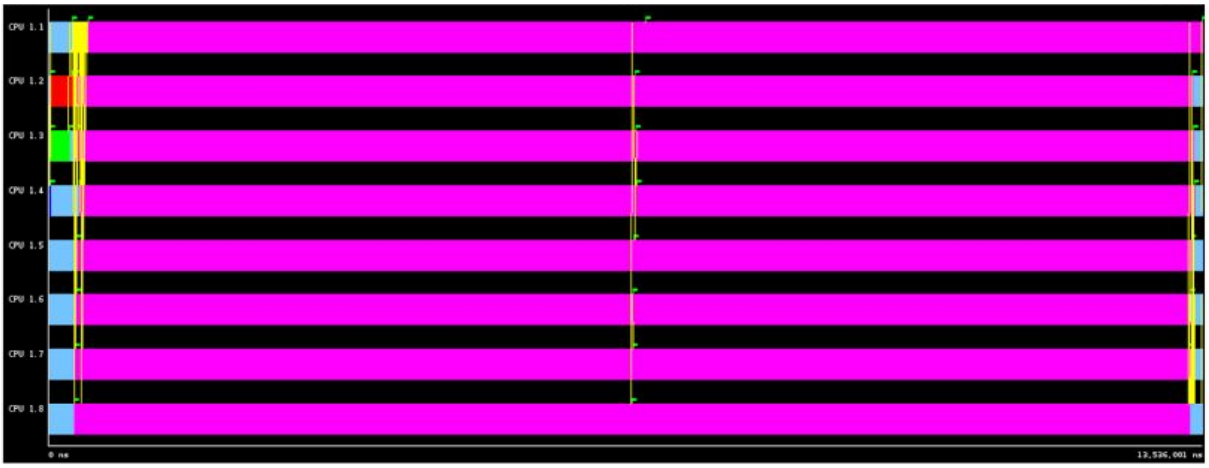
Visualizing the task graph and data dependences

4. Include the source code for function dot product in which you show the Tareador instrumentation that has been added to study the potential parallelism in the code. This instrumentation has to appropriately define tasks and filter the analysis of variable(s) that cause the dependence(s).

```
void dot_product (long N, double A[N], double B[N], double *acc){
   double prod;
   int i;
   *acc=0.0;
   for (i=0; i<N; i++) {
         tareador_start_task("dot_product"); /* We create a task for each iteration */
         prod = my_func(A[i], B[i]);
         tareador_disable_object(acc);       /* 'Delete' the dependence */
         *acc += prod;                       /* acc is the variable that causes dependence */
         tareador_enable_object(acc);
         tareador_end_task("dot_product");
   }
}
```

5. Capture the task dependence graph for that task decomposition and the execution timelines (for 8 processors) that allow you to understand the potential parallelism attainable. Briefly comment the relevant information that is reported by the tools.





With both screens we can see the power of parallelism, we can see independent tasks being executed at the same time, saving a lot of execution time so the program goes faster. As expected, the granularity of the parallel task is bigger than others because it's the key point of the program, also, other tasks on the graph could just be the task that unifies the results of each parallel execution.

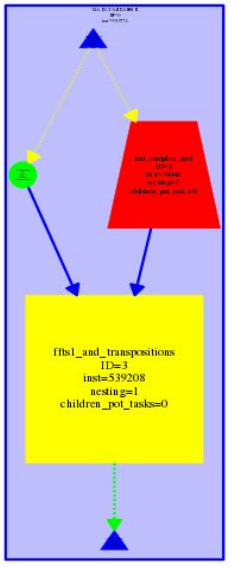
Analysis of task decompositions

6. Complete the following table for the initial and different versions generated for 3dfft seq.c, briefly commenting the evolution of the metrics with the different versions.

Version	T1 (1 Processor)	T∞ (∞ processors)*	Parallelism (T1/T∞)
seq	593772	593758	1,000023578629677
v1	593772	593758	1,000023578629677
v2	593772	315523	1,88186598124384
v3	593772	109063	5,444302834141735
v4	593772	60148	9,871849438052803

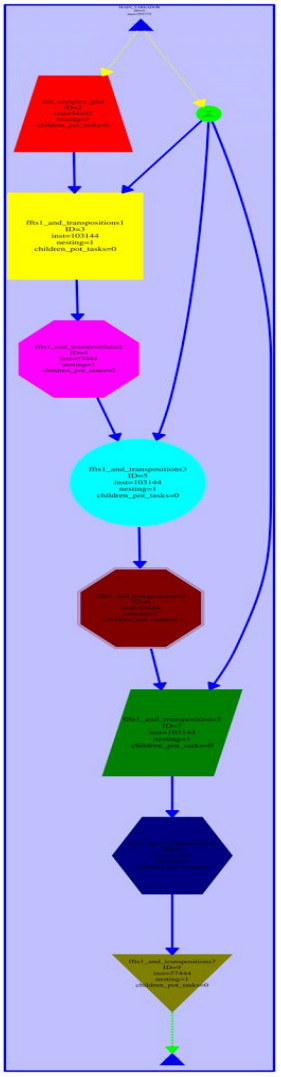
We suppose for this exercise, any task starts before any of it dependences finish, we summed the critical path following the graph, and not the time the simulation says.

Seq version:



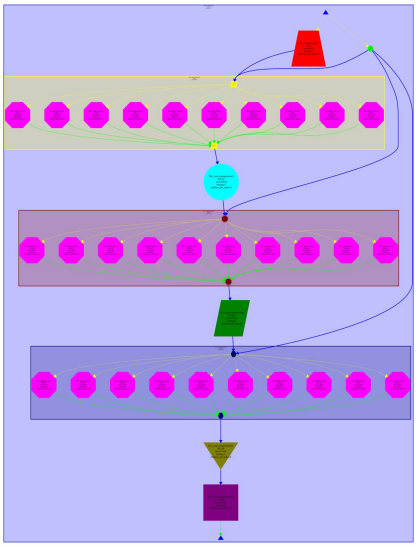
v1:

```
START_COUNT_TIME;
tareador_start_task("ffts1_and_transpositions1");
ffts1_planes(p1d, in_fftw);
tareador_end_task("ffts1_and_transpositions1");

tareador_start_task("ffts1_and_transpositions2");
transpose_xy_planes(tmp_fftw, in_fftw);
tareador_end_task("ffts1_and_transpositions2");

tareador_start_task("ffts1_and_transpositions3");
ffts1_planes(p1d, tmp_fftw);
tareador_end_task("ffts1_and_transpositions3");

tareador_start_task("ffts1_and_transpositions4");
transpose_zx_planes(in_fftw, tmp_fftw);
tareador_end_task("ffts1_and_transpositions4");

tareador_start_task("ffts1_and_transpositions5");
ffts1_planes(p1d, in_fftw);
tareador_end_task("ffts1_and_transpositions5");

tareador_start_task("ffts1_and_transpositions6");
transpose_zx_planes(tmp_fftw, in_fftw);
tareador_end_task("ffts1_and_transpositions6");

tareador_start_task("ffts1_and_transpositions7");
transpose_xy_planes(in_fftw, tmp_fftw);
tareador_end_task("ffts1_and_transpositions7");
```



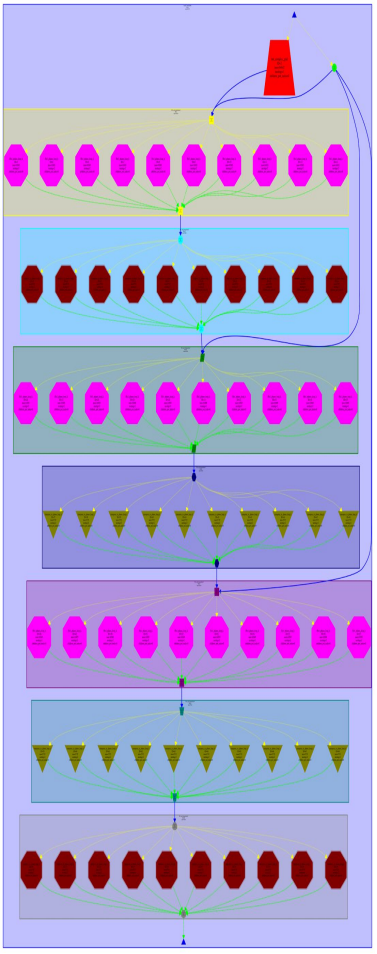
v2:

```
void ffts1_planes(fftwf_plan p1d, fftwf_complex
in_fftw[][N][N]){
    int k,j;
    for (k=0; k<N; k++)  {
        tareador_start_task("ffts1_planes_loop_k");
        for (j=0; j<N; j++)
        fftwf_execute_dft( p1d, (fftwf_complex*)in_fftw[k][j][0],
        (fftwf_complex *)in_fftw[k][j][0]);
        tareador_end_task("ffts1_planes_loop_k");
    }
}
```



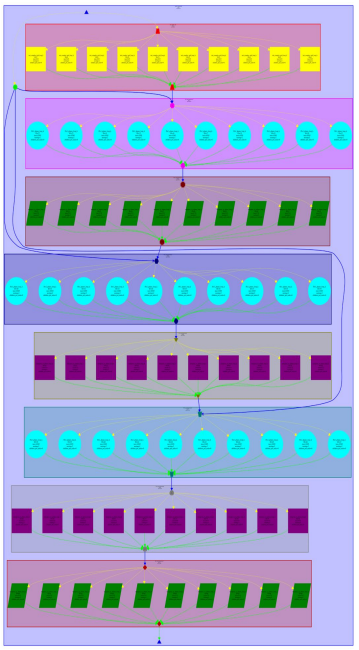
v3:

```
void transpose_xy_planes(fftwf_complex  tmp_fftw[][N][N],
fftwf_complex in_fftw[][N][N]){
    int k,j,i;
    for (k=0; k<N; k++) {
        tareador_start_task("transpose_xy_planes_loop_k");
        for (j=0; j<N; j++)
            for (i=0; i<N; i++){
                tmp_fftw[k][i][j][0] = in_fftw[k][j][i][0];
                tmp_fftw[k][i][j][1] = in_fftw[k][j][i][1];
            }
            tareador_end_task("transpose_xy_planes_loop_k");
    }
}

void transpose_zx_planes(fftwf_complex in_fftw[][N][N],
fftwf_complex tmp_fftw[][N][N]){
    int k, j, i;
    for (k=0; k<N; k++) {
        tareador_start_task("transpose_zx_planes_loop_k");
        for (j=0; j<N; j++)
            for (i=0; i<N; i++){
                in_fftw[i][j][k][0] = tmp_fftw[k][j][i][0];
                in_fftw[i][j][k][1] = tmp_fftw[k][j][i][1];
            }
        tareador_end_task("transpose_zx_planes_loop_k");
     }
}
```



v4:

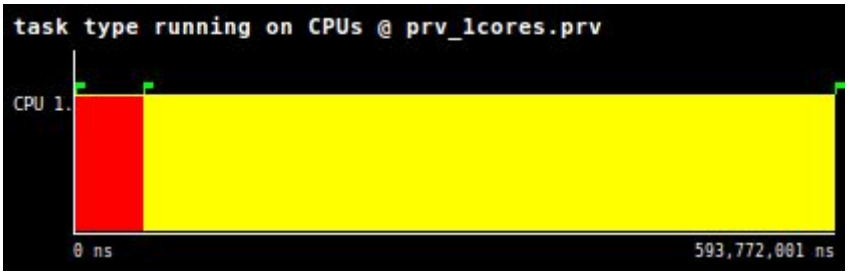
```
void init_complex_grid(fftwf_complex in_fftw[][N][N]){
   int k,j,i;
   for (k = 0; k < N; k++) {
      tareador_start_task("init_complex_grid_loop_k");
      for (j = 0; j < N; j++)
         for (i = 0; i < N; i++){
            in_fftw[k][j][i][0] = (float)(sin(M_PI*((float)i)/64.0)+
            sin(M_PI*((float)i)/32.0)+sin(M_PI*((float)i/16.0)));
            in_fftw[k][j][i][1] = 0;
#if TEST
            out_fftw[k][j][i][0]= in_fftw[k][j][i][0];
            out_fftw[k][j][i][1]= in_fftw[k][j][i][1];
#endif
         }
         tareador_end_task("init_complex_grid_loop_k");
   }
}
```



We see that the number of the instructions of the critical path in each version decreases so the parallelism increase because the total number of instructions in each version is the same. This happens because in each version we create more tasks that can work simultaneously.

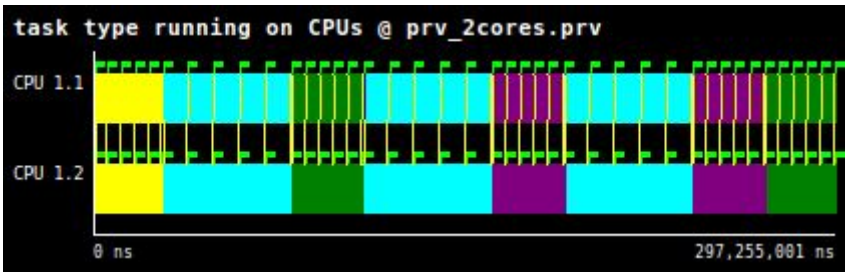
7. With the results from the parallel simulation with 2, 4, 8, 16 and 32 processors, draw the execution time and speedup plots for version v4 with respect to the sequential execution (that you can estimate from the simulation of the initial task decomposition that we provided in 3dfft seq.c, using just 1 processor). Briefly comment the scalability behaviour shown on these two plots.

Seq version 1 core



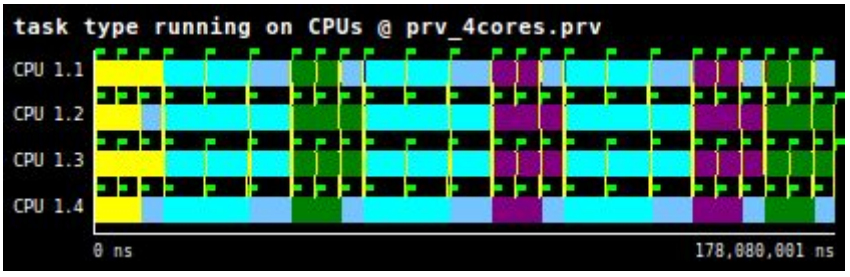
Texe = 593,772,001 ns
Speed-up = 1

v4 2 cores



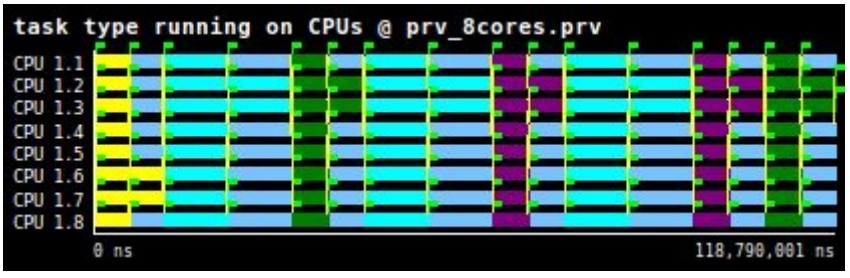
Texe = 297,255,001 ns
Speed-up = 593,772,001 ns / 297,255,001 ns = 1,9975

v4 4 cores



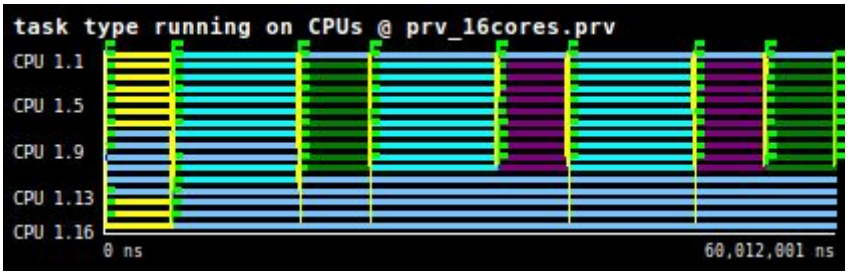
Texe = 178,080,001 ns
Speed-up = 593,772,001 ns / 178,080,001 ns = 3,3343

v4 8 cores



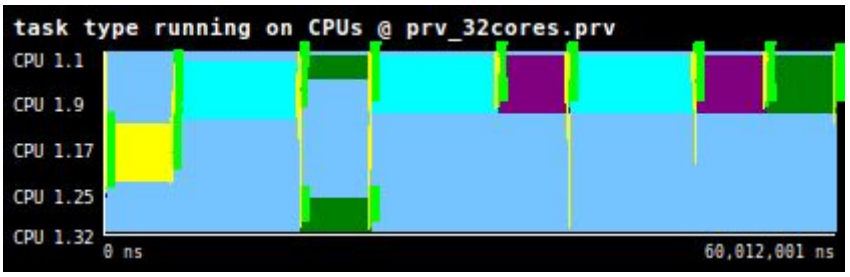
Texe = 118,790,001 ns
Speed-up = 593,772,001 ns / 118,790,001 ns = 4,9985

v4 16 cores



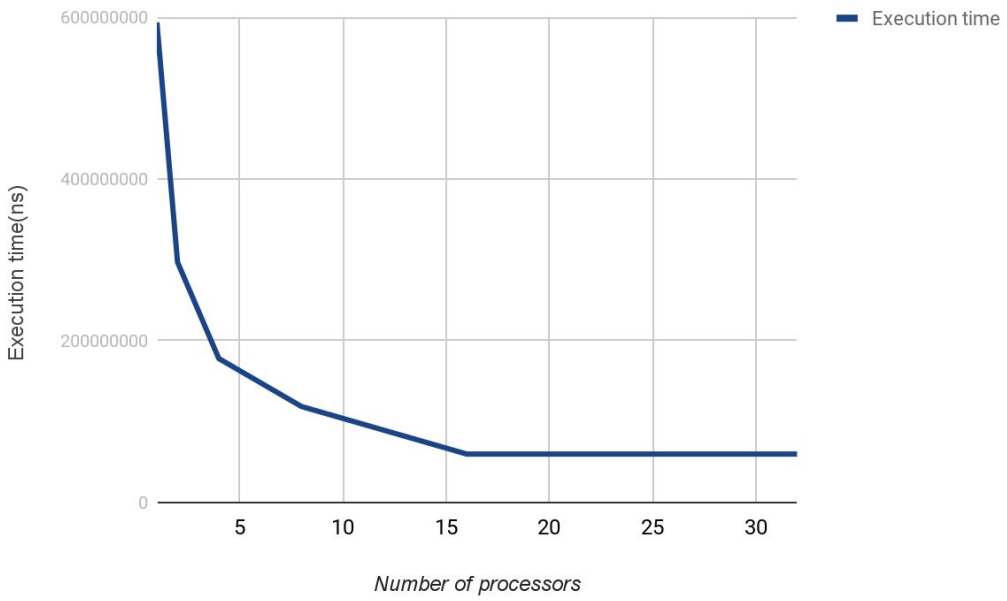
Texe = 60,012,001 ns
Speed-up = 593,772,001 ns / 60,012,001 ns = 9,8942

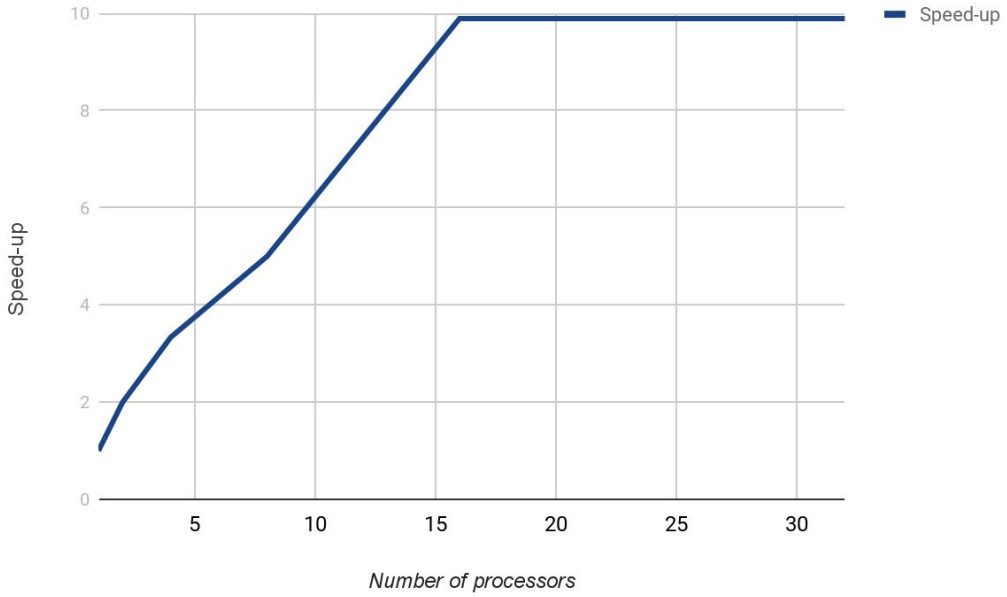
v4 32 cores



Texe = 60,012,001 ns

Speed-up = 593,772,001 ns / 60,012,001 ns = 9,8942

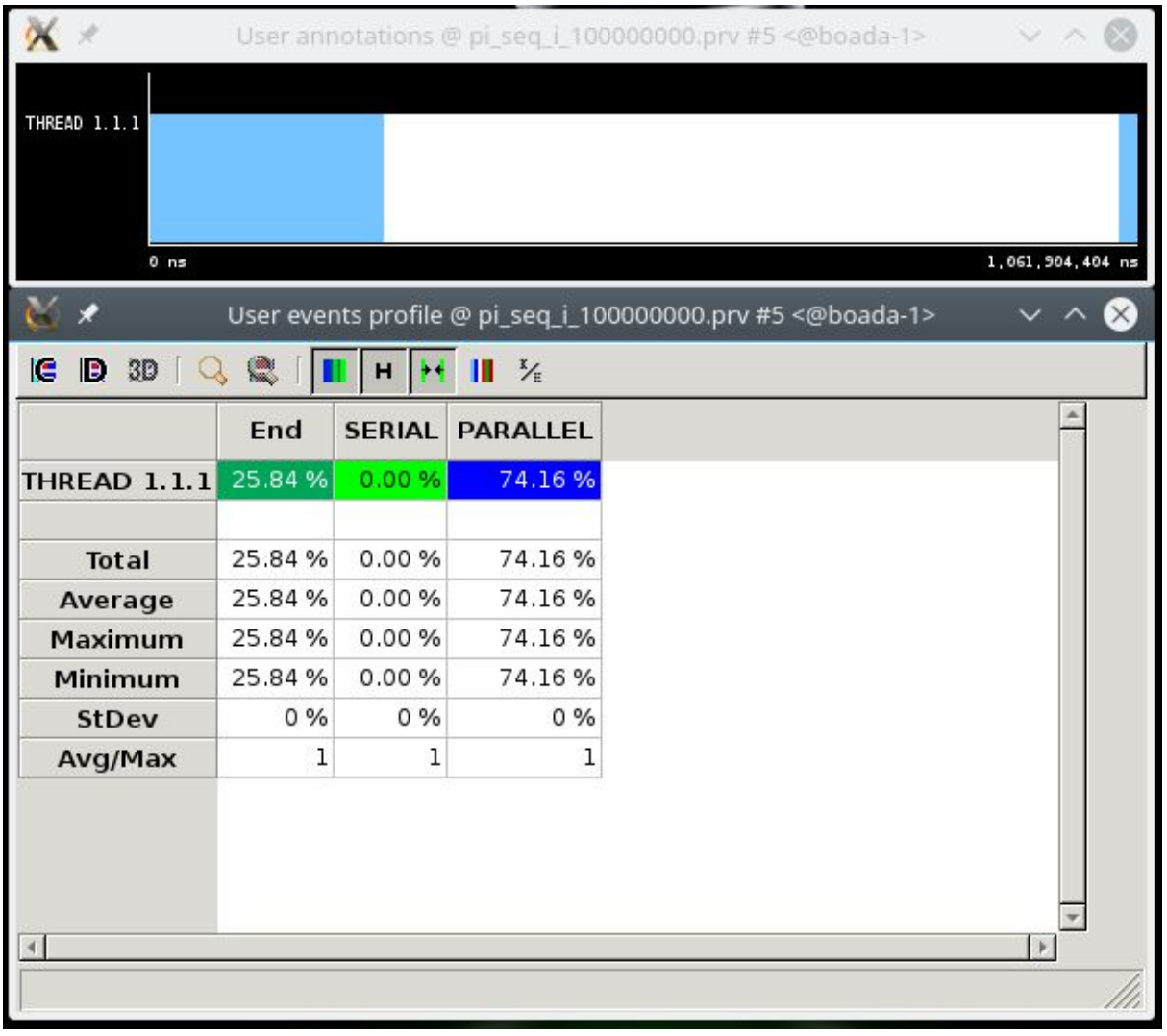




In the figures, we can see that if we increase the number of cores, the execution time decrease and the speed-up increase. That's because v4 executes the tasks in parallel while the sequential version does not. Also, we can see that if we have more than 16 cores the result is the same. That's because there are parts of the program that can't be divided in more tasks due to dependencies for example.

Tracing sequential and parallel executions

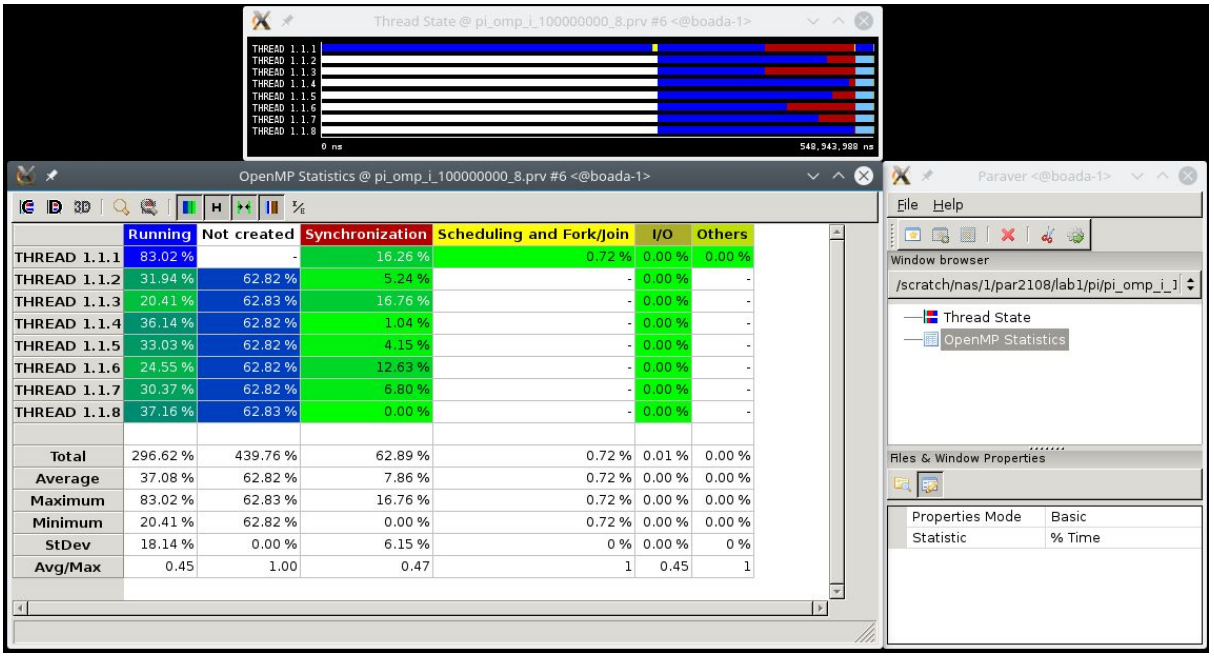
8. From the instrumented version of pi seq.c, and using the appropriate Paraver configuration file, obtain the value of the parallel fraction φ for this program when executed with 100.000.000 iterations, showing the steps you followed to obtain it. Clearly indicate which Paraver configuration file(s) did you use.



	End	SERIAL	PARALLEL
THREAD 1.1.1	25.84 %	0.00 %	74.16 %
Total	25.84 %	0.00 %	74.16 %
Average	25.84 %	0.00 %	74.16 %
Maximum	25.84 %	0.00 %	74.16 %
Minimum	25.84 %	0.00 %	74.16 %
StDev	0 %	0 %	0 %
Avg/Max	1	1	1

On the image generated thanks to configuration files "APP_userevents.cfg" and "APP_userevents_profile.cfg" we can see that the parallel fraction φ has a value of 74.16%.

9. From the instrumented version of pi omp.c, and using the appropriate Paraver configuration file, show a profile of the % of time spent in the different OpenMP states when using 8 threads and for 100.000.000 iterations. Clearly indicate which Paraver configuration file(s) did you use and your own conclusions from that profile.



	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	83.02 %	-	16.26 %	0.72 %	0.00 %	0.00 %
THREAD 1.1.2	31.94 %	62.82 %	5.24 %	-	0.00 %	-
THREAD 1.1.3	20.41 %	62.83 %	16.76 %	-	0.00 %	-
THREAD 1.1.4	36.14 %	62.82 %	1.04 %	-	0.00 %	-
THREAD 1.1.5	33.03 %	62.82 %	4.15 %	-	0.00 %	-
THREAD 1.1.6	24.55 %	62.82 %	12.63 %	-	0.00 %	-
THREAD 1.1.7	30.37 %	62.82 %	6.80 %	-	0.00 %	-
THREAD 1.1.8	37.16 %	62.83 %	0.00 %	-	0.00 %	-
Total	296.62 %	439.76 %	62.89 %	0.72 %	0.01 %	0.00 %
Average	37.08 %	62.82 %	7.86 %	0.72 %	0.00 %	0.00 %
Maximum	83.02 %	62.83 %	16.76 %	0.72 %	0.00 %	0.00 %
Minimum	20.41 %	62.82 %	0.00 %	0.72 %	0.00 %	0.00 %
StDev	18.14 %	0.00 %	6.15 %	0 %	0.00 %	0 %
Avg/Max	0.45	1.00	0.47	1	0.45	1

The configuration file used is “OMP_state_profile.cfg” with . With this profile, we can see every thread created by parallelism. In total percentage, we can see a number above 100%, this is because parallelism and its implied speedup against a sequential execution.