# First deliverable

*Álvaro Martínez Arroyo*

*Daniel García Romero*

*par2303*

*Fall 2015-2016*

# 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).**

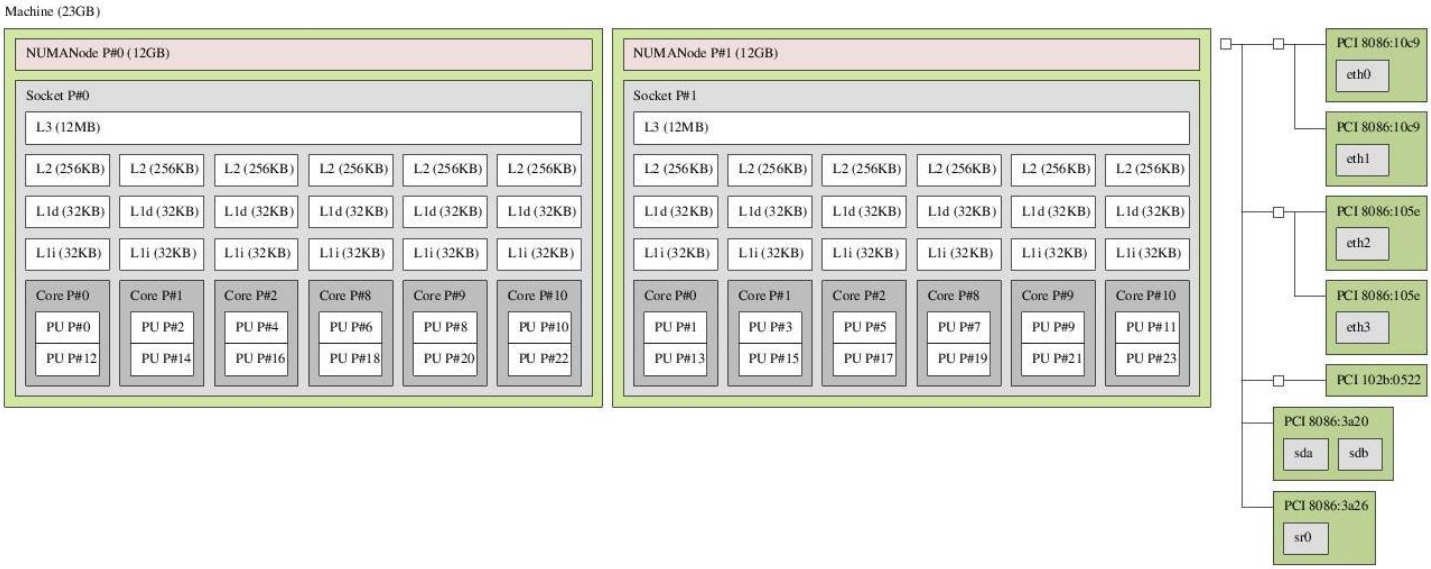



***Figura 1. Arquitectura de nuestra máquina***

Number of sockets: 2

Cores per socket: 6

Threads per core: 2

Cache: L1d cache: 32 KB

L1i cache: 32 KB

L2 cache: 256 KB

L3 cache: 12 MB

Main memory: 24 GB

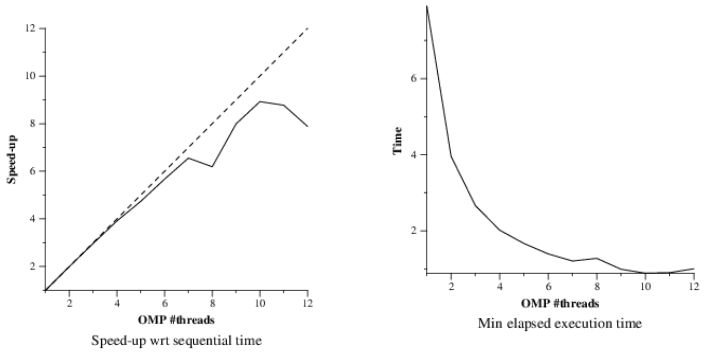
# Timing sequential and parallel executions

**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.**

```
#include <sys/time.h>  /* library header file */
…
double getusec_() {
        struct timeval time;
        gettimeofday(&time, NULL);  /* library function that need to be invoked */
        return ((double)time.tv_sec * (double)1e6 + (double)time.tv_usec);
        /* tv_sec y tv_usec son segundos y microsegundos, respectivamente */
}
#define START_COUNT_TIME stamp = getusec_();
#define STOP_COUNT_TIME(_m) stamp = getusec_() - stamp;\
                        stamp = stamp/1e6;\
                        printf ("%s%0.6f\n",(_m), stamp);
…

START_COUNT_TIME;
…  /* parte del código cuyo tiempo de ejecución queremos medir */
STOP_COUNT_TIME("");
```

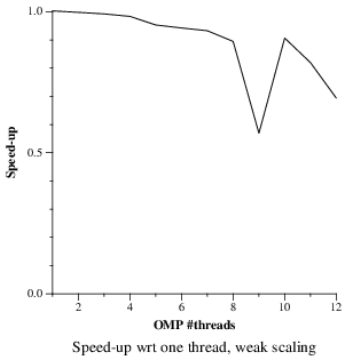
**3. Plot the speed-up obtained when varying the number of threads (strong scalability) and problem size (weak scalability) for pi omp.c. Reason about how the scalability of the program.**




Strong scalability

Al aumentar el número de threads, disminuimos el tiempo de ejecución y, en consecuencia, aumentamos el speed-up.

***Figura 2. Strong scalability***



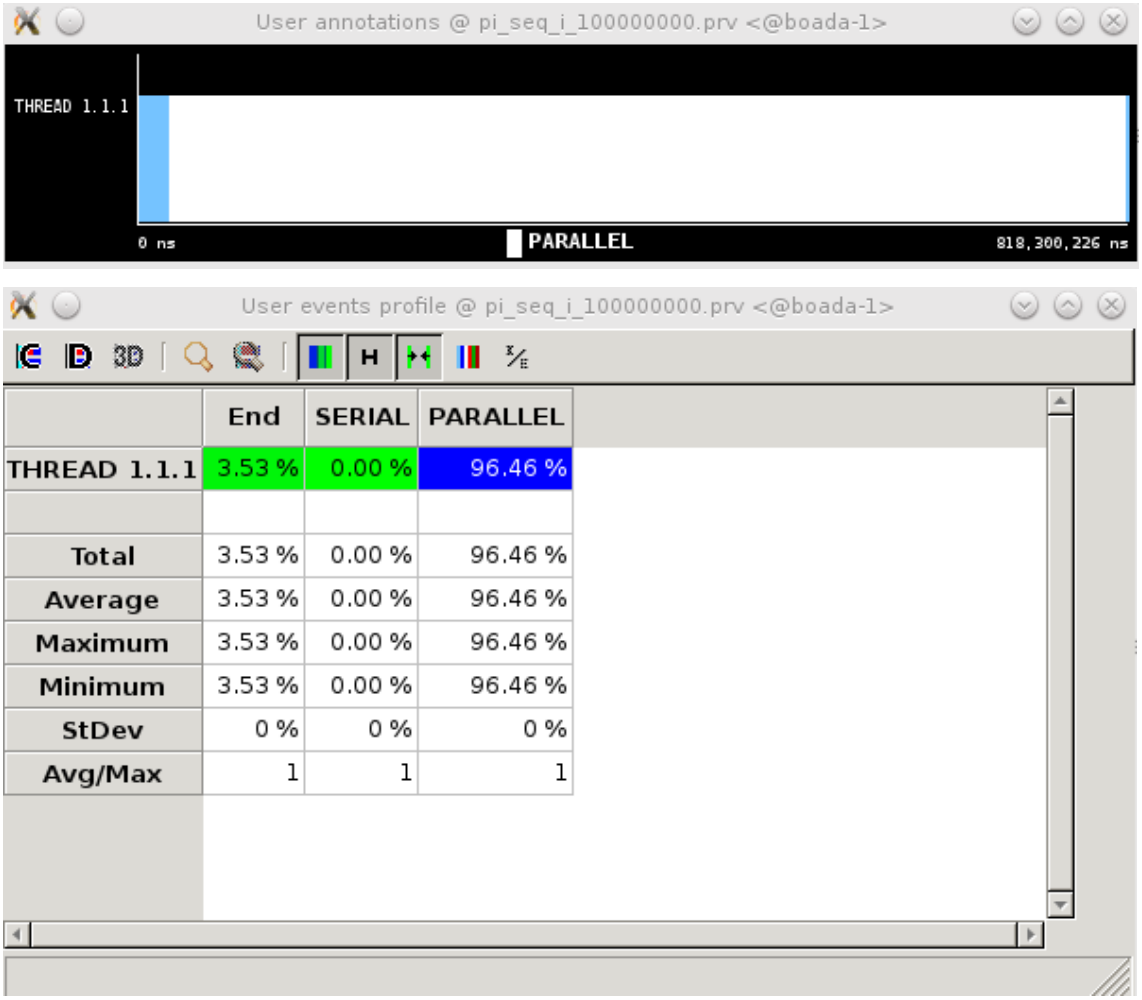

Weak scalability

Al aumentar el número de threads, aumenta el tamaño del problema, por lo que aumenta el tiempo de ejecución y, en consecuencia, disminuye el speed-up.

***Figura 3. Weak scalability***

# Tracing sequential and parallel executions

**4. From the instrumented version of pi seq.c, and using the appropriate Paraver configuration file, obtain the value of the *parallel fraction* $\Phi$ for this program when executed with 100.000.000 iterations.**

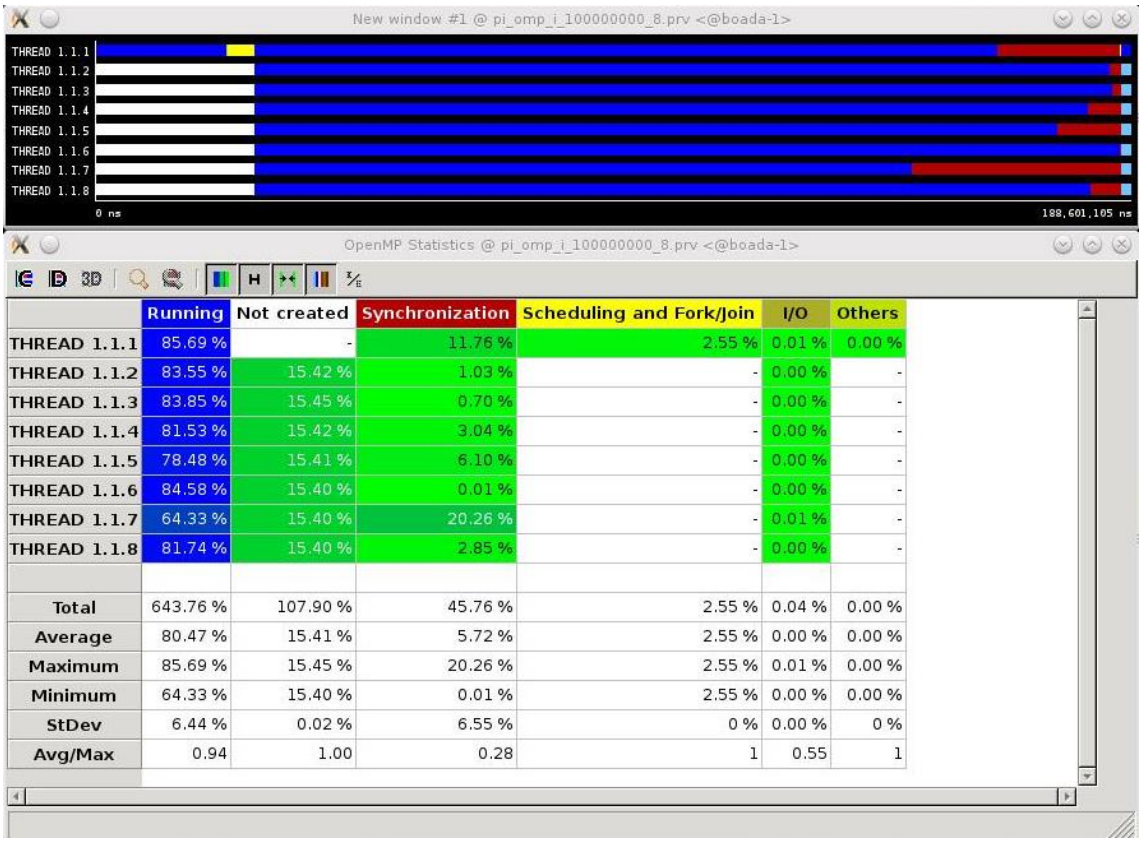



| | End | SERIAL | PARALLEL |
|---|---|---|---|
| THREAD 1.1.1 | 3.53 % | 0.00 % | 96.46 % |
| | | | |
| Total | 3.53 % | 0.00 % | 96.46 % |
| Average | 3.53 % | 0.00 % | 96.46 % |
| Maximum | 3.53 % | 0.00 % | 96.46 % |
| Minimum | 3.53 % | 0.00 % | 96.46 % |
| StDev | 0 % | 0 % | 0 % |
| Avg/Max | 1 | 1 | 1 |

***Figura 4. Traza generada para pi_seq.c en Paraver***

La fracción paralela $\Phi$ representa el 96.46% del total del tiempo de ejecución.

**5. 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 ONLY during the execution of the parallel region (not considering the sequential part before and after) when using 8 threads and for 100.000.000 iterations.**




| | Running | Not created | Synchronization | Scheduling and Fork/Join | I/O | Others |
|---|---|---|---|---|---|---|
| THREAD 1.1.1 | 85.69 % | - | 11.76 % | 2.55 % | 0.01 % | 0.00 % |
| THREAD 1.1.2 | 83.55 % | 15.42 % | 1.03 % | - | 0.00 % | - |
| THREAD 1.1.3 | 83.85 % | 15.45 % | 0.70 % | - | 0.00 % | - |
| THREAD 1.1.4 | 81.53 % | 15.42 % | 3.04 % | - | 0.00 % | - |
| THREAD 1.1.5 | 78.48 % | 15.41 % | 6.10 % | - | 0.00 % | - |
| THREAD 1.1.6 | 84.58 % | 15.40 % | 0.01 % | - | 0.00 % | - |
| THREAD 1.1.7 | 64.33 % | 15.40 % | 20.26 % | - | 0.01 % | - |
| THREAD 1.1.8 | 81.74 % | 15.40 % | 2.85 % | - | 0.00 % | - |
| Total | 643.76 % | 107.90 % | 45.76 % | 2.55 % | 0.04 % | 0.00 % |
| Average | 80.47 % | 15.41 % | 5.72 % | 2.55 % | 0.00 % | 0.00 % |
| Maximum | 85.69 % | 15.45 % | 20.26 % | 2.55 % | 0.01 % | 0.00 % |
| Minimum | 64.33 % | 15.40 % | 0.01 % | 2.55 % | 0.00 % | 0.00 % |
| StDev | 6.44 % | 0.02 % | 6.55 % | 0 % | 0.00 % | 0 % |
| Avg/Max | 0.94 | 1.00 | 0.28 | 1 | 0.55 | 1 |

***Figura 5. Traza generada para pi_omp.c en Paraver***

# Visualizing the task graph and data dependences

**6. 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;
        *acc=0.0;
        int i;
```

```
        for (i=0; i<N; i++) {
                tareador_start_task("exercise6");
                /* Now begins a code region to be a potential task */
                tareador_disable_object(acc);
                /* Now begins a code region with memory accesses to variable acc */
                prod = my_func(A[i], B[i]);
                *acc += prod;
                tareador_enable_object(acc);
                tareador_end_task("exercise6");
        }
}
```

**7. Capture the task dependence graph and execution timeline (for 8 processors) for that task decomposition.**

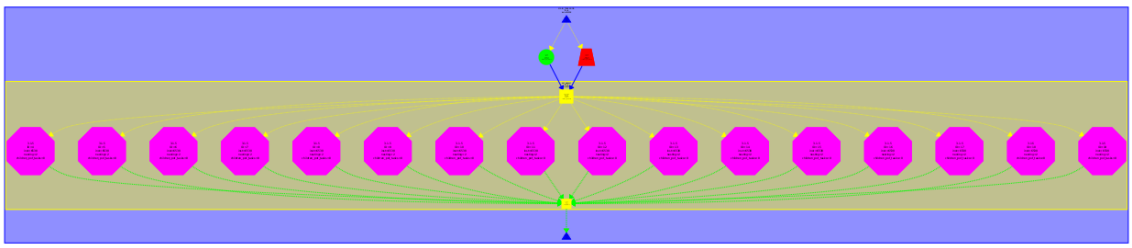


*Figura 6. Task dependence graph*


*Figura 7. Execution timeline*

# Analysis of task decompositions

**8. Complete the following table for the initial and different versions generated for 3dfft_seq.c.**

| Version | | $T_1$ | $T_\infty$ | Parallelism ($T_1$ / $T_\infty$) |
|---|---|---|---|---|
| **seq** | Manualmente | 593.772.000 ns | 593.758.000 ns | 1,000023579 |
| | Paraver | 593.772.001 ns | 593.705.001 ns | 1,000112851 |
| **v1** | Manualmente | 593.772.000 ns | 593.758.000 ns | 1,000023579 |
| | Paraver | 593.772.001 ns | 593.705.001 ns | 1,000112851 |
| **v2** | Manualmente | 593.772.000 ns | 315.523.000 ns | 1,881865981 |
| | Paraver | 593.772.001 ns | 315.470.001 ns | 1,882182138 |
| **v3** | Manualmente | 593.772.000 ns | 109.063.000 ns | 5,444302834 |
| | Paraver | 593.772.001 ns | 109.010.001 ns | 5,446949780 |
| **v4** | Manualmente | 593.772.000 ns | 60.148.000 ns | 9,871849438 |
| | Paraver | 593.772.001 ns | 60.095.001 ns | 9,880555639 |

*Figura 8. Analysis of task decompositions*

Para calcular $T_1$ y $T_\infty$ manualmente, primero hemos calculado los ns/instrucción, dividiendo las instrucciones totales (*Tareador*) entre el tiempo total en ns que tardan en ejecutarse (*View Simulation* de *Tareador*), e independientemente del número de procesadores, el resultado obtenido ha sido de, aproximadamente, 1000 ns/instrucción. Sabiendo esto, y mirando el número de instrucciones con *Tareador*, hemos podido calcular $T_1$ y $T_\infty$ ($T_1$ a partir de las instrucciones totales y $T_\infty$ a partir de las instrucciones del camino crítico).

Otra manera de obtener $T_1$ y $T_\infty$ es mediante *View Simulation* de *Tareador* con 1 procesador para $T_1$ y con Pmin para $T_\infty$, siendo Pmin el número de procesadores mínimos para conseguir $T_\infty$.

- seq

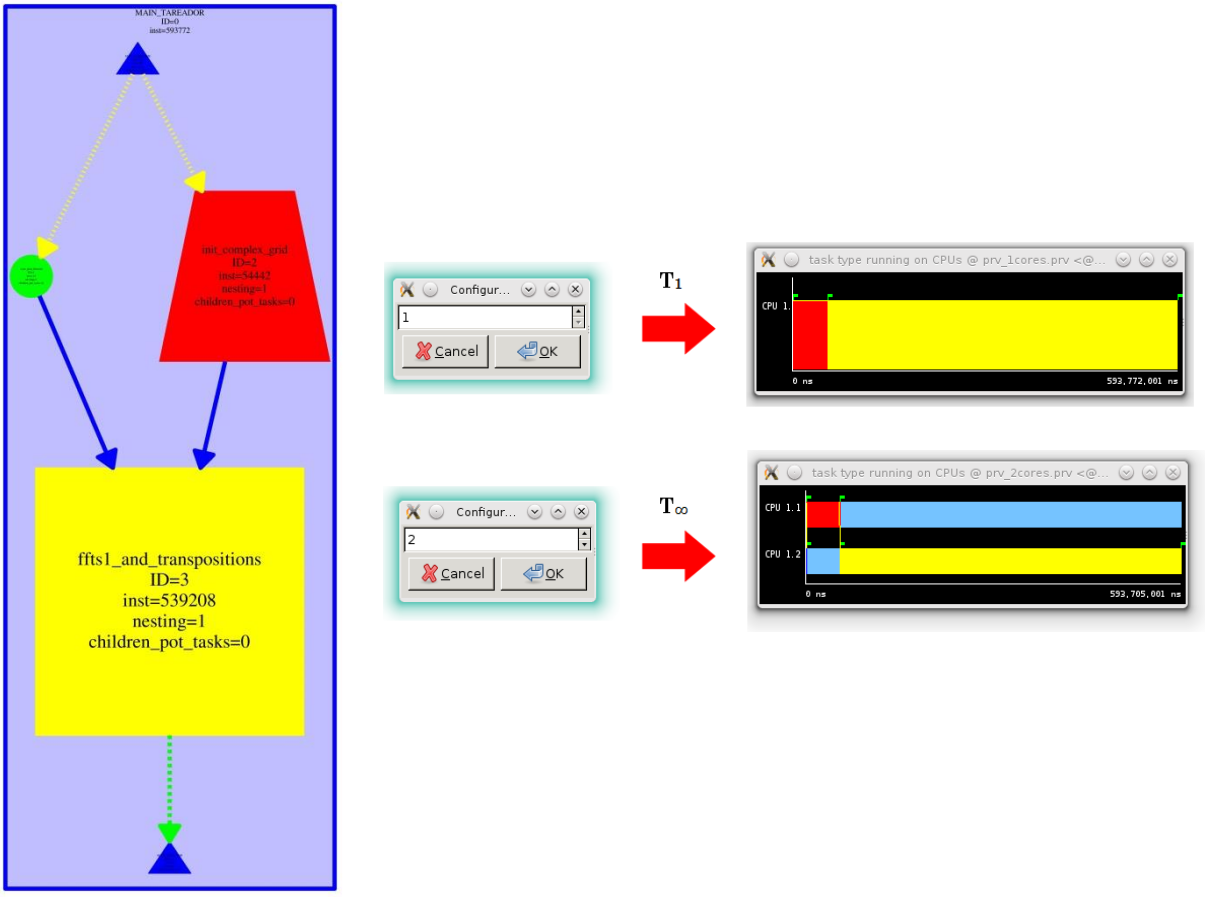



***Figura 9. 3dfft versión secuencial en Tareador y Paraver***

$T_1$ = 593.772 instr. x 1000 ns/instr. = 593.772.000 ns

$T_\infty$ = (108 + 54442 + 539208) instr. x 1000 ns/instr. = 593.758.000 ns

- v1

```
START_COUNT_TIME;

tareador_start_task("ffts1_and_transpositions");

tareador_start_task("task1");
ffts1_planes(pld, in_fftw);
tareador_end_task("task1");

tareador_start_task("task2");
transpose_xy_planes(tmp_fftw, in_fftw);
tareador_end_task("task2");

tareador_start_task("task3");
ffts1_planes(pld, tmp_fftw);
tareador_end_task("task3");

tareador_start_task("task4");
transpose_zx_planes(in_fftw, tmp_fftw);
tareador_end_task("task4");

tareador_start_task("task5");
ffts1_planes(pld, in_fftw);
tareador_end_task("task5");

tareador_start_task("task6");
transpose_zx_planes(tmp_fftw, in_fftw);
tareador_end_task("task6");

tareador_start_task("task7");
transpose_xy_planes(in_fftw, tmp_fftw);
tareador_end_task("task7");

tareador_end_task("ffts1_and_transpositions");

STOP_COUNT_TIME("Execution FFT3D");
```

***Figura 10. Código de 3dfft, versión 1***

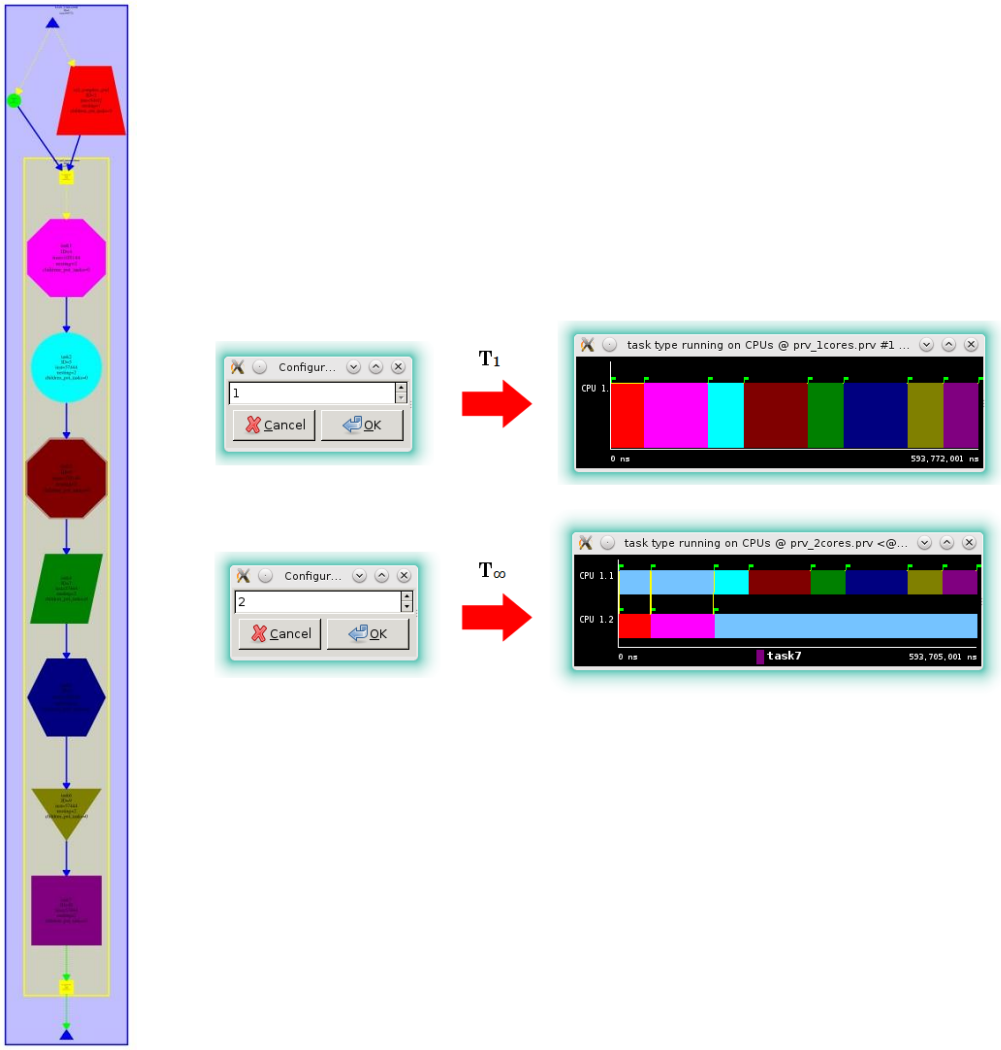



***Figura 11. 3dfft versión 1 en Tareador y Paraver***

$T_1$ = 593.772 instr. x 1000 ns/instr. = 593.772.000 ns

$T_\infty$ = (108 + 54442 + 103144 + 57444 + 103144 + 57444 + 103144 + 57444 + 57444) instr. x 1000 ns/instr. = 593.758.000 ns

- 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");
    }
}
```

***Figura 12. Código de 3dfft, versión 2***

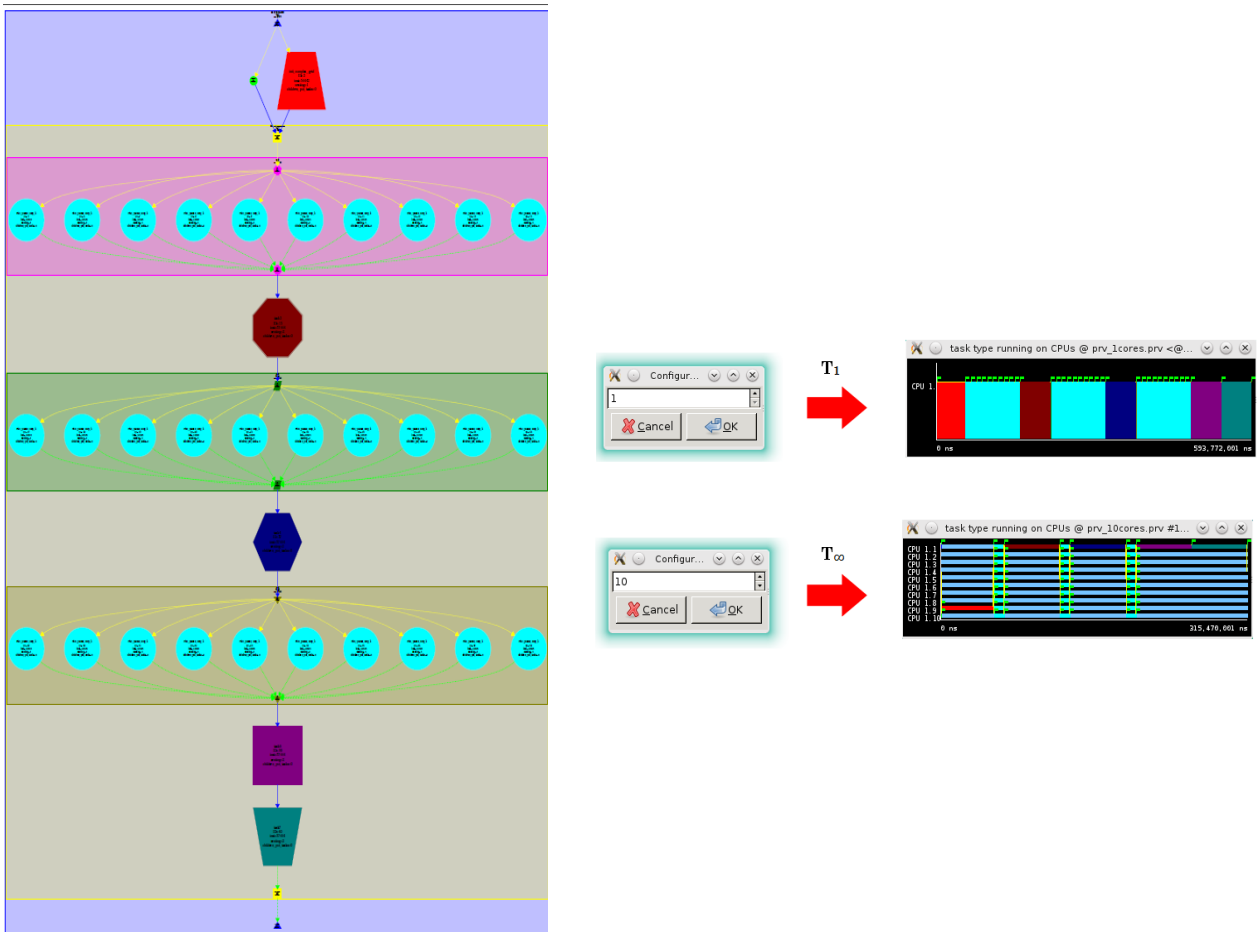



***Figura 13. 3dfft versión 2 en Tareador y Paraver***

$T_1$ = 593.772 instr. x 1000 ns/instr. = 593.772.000 ns

$T_\infty$ = (108 + 54442 + 94 + 10305 + 57444 + 94 + 10305 + 57444 + 94 + 10305 + 57444 + 57444) instr. x 1000 ns/instr. = 315.523.000 ns

- 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");
     }
}
```

***Figura 14. Código de 3dfft, versión 3***

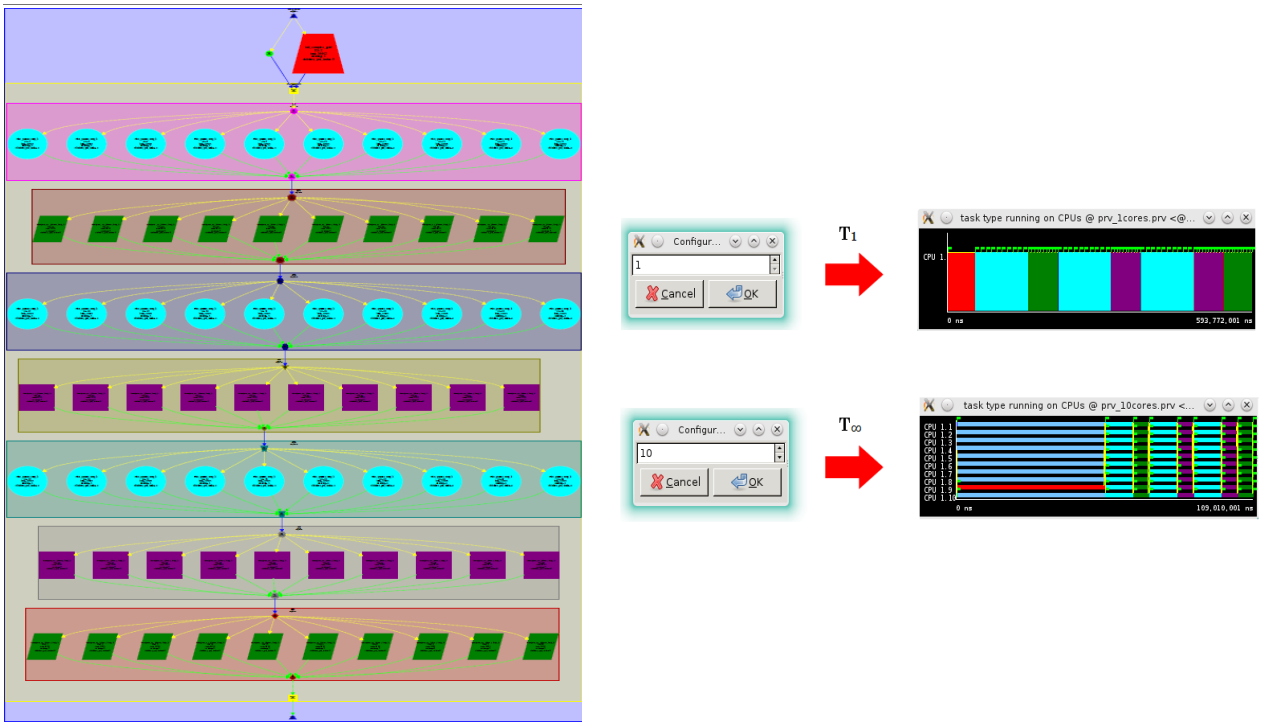



***Figura 15. 3dfft versión 3 en Tareador y Paraver***

$T_1$ = 593.772 instr. x 1000 ns/instr. = 593.772.000 ns

$T_\infty$ = (108 + 54442 + 94 + 10305 + 94 + 5735 + 94 + 10305 + 94 + 5735 + 94 + 10305 + 94 + 5735 + 94 + 5735) instr. x 1000 ns/instr. = 109.063.000 ns

- 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");
  }
}
```

***Figura 16. Código de 3dfft, versión 4***

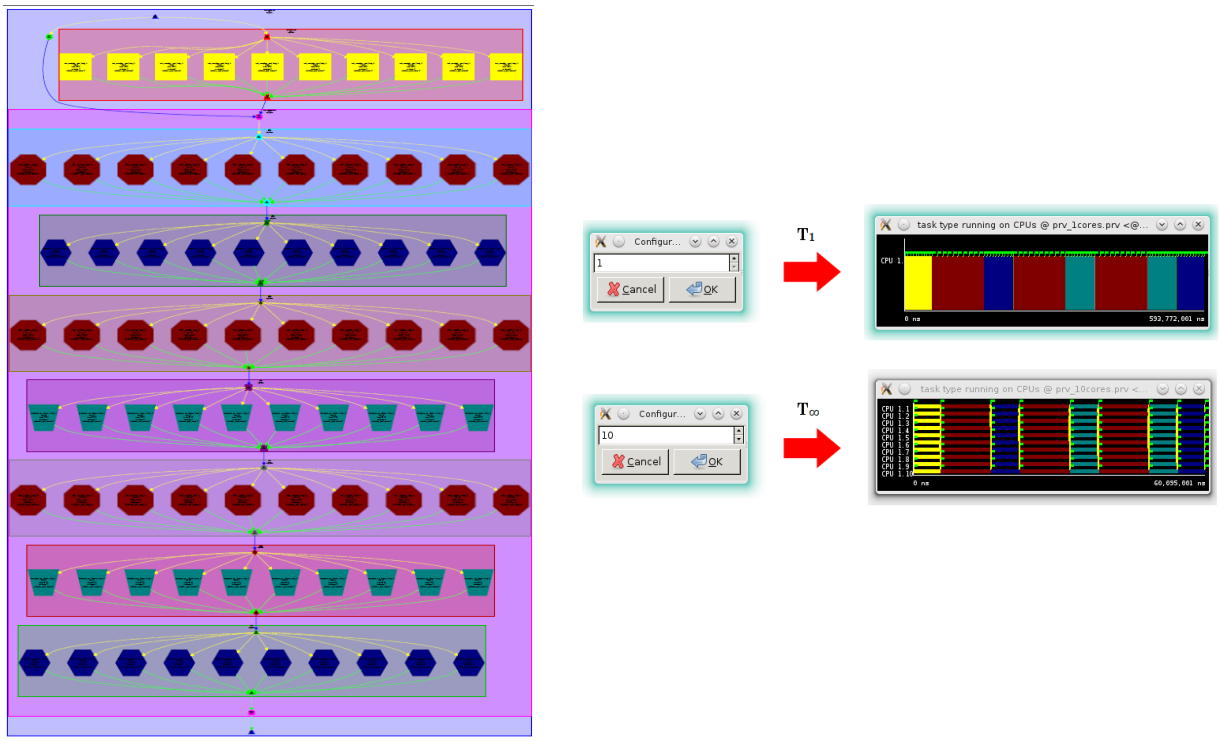



***Figura 17. 3dfft versión 4 en Tareador y Paraver***

$T_1$ = 593.772 instr. x 1000 ns/instr. = 593.772.000 ns

$T_\infty$ = (108 + 92 + 5435 + 94 x 7 + 10305 x 3 + 5735 x 4) instr. x 1000 ns/instr. = 60.148.000 ns

**10. 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).**

- 3dfft sequential execution (1 processor) → 593.772.001 ns
- 3dfft parallel execution with 2 processors → 297.255.001 ns

  (speedup = 593.772.001 / 297.255.001 = 1,99751728)
- 3dfft parallel execution with 4 processors → 178.080.001 ns

  (speedup = 593.772.001 / 178.080.001 = 3,334299178)
- 3dfft parallel execution with 8 processors → 118.790.001 ns

  (speedup = 593.772.001 / 118.790.001 = 4,998501566)
- 3dfft parallel execution with 16 processors → 60.012.001 ns

  (speedup = 593.772.001 / 60.012.001 = 9,894221008)
- 3dfft parallel execution with 32 processors → 60.012.001 ns

  (speedup = 593.772.001 / 60.012.001 = 9,894221008)

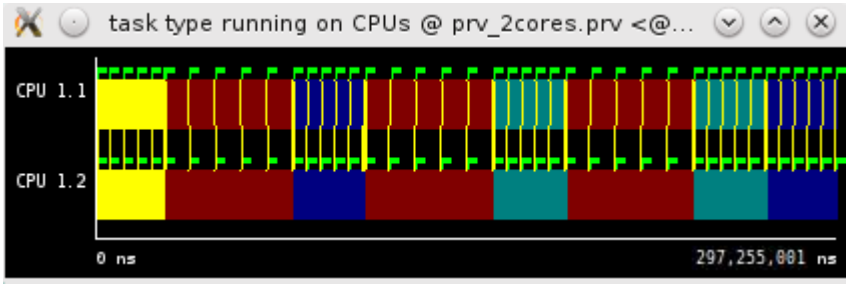



*Figura 18. 3dfft v4 parallel execution with 2 processors*

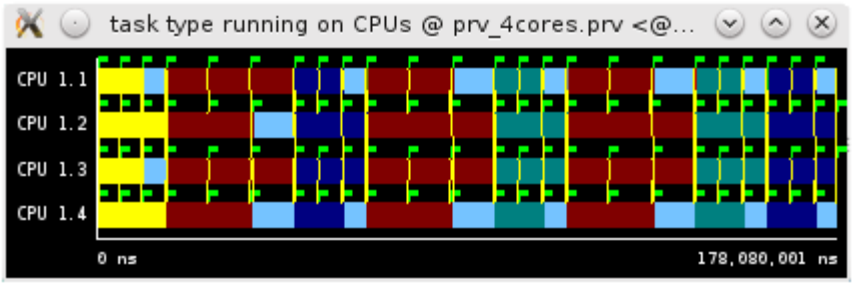



*Figura 19. 3dfft v4 parallel execution with 4 processors*

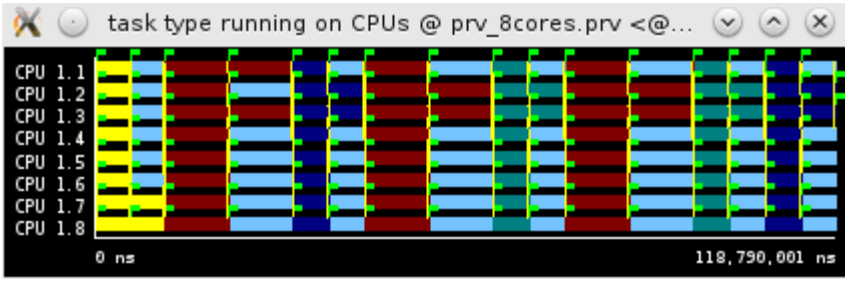



*Figura 20. 3dfft v4 parallel execution with 8 processors*

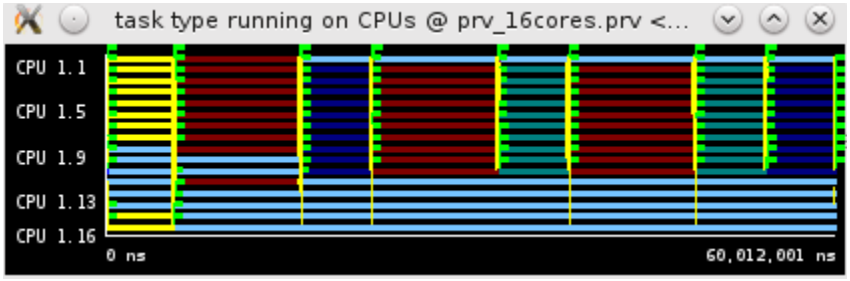



*Figura 21. 3dfft v4 parallel execution with 16 processors*

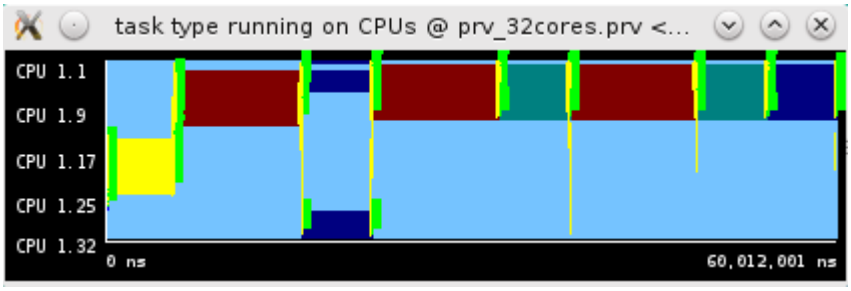



*Figura 22. 3dfft v4 parallel execution with 32 processors*

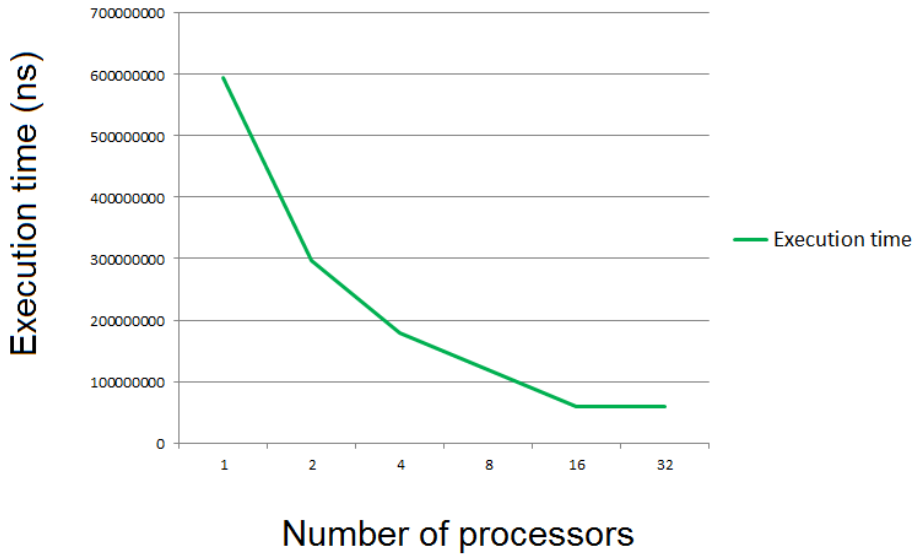



***Figura 23. Tiempo de ejecución en función del número de procesadores***

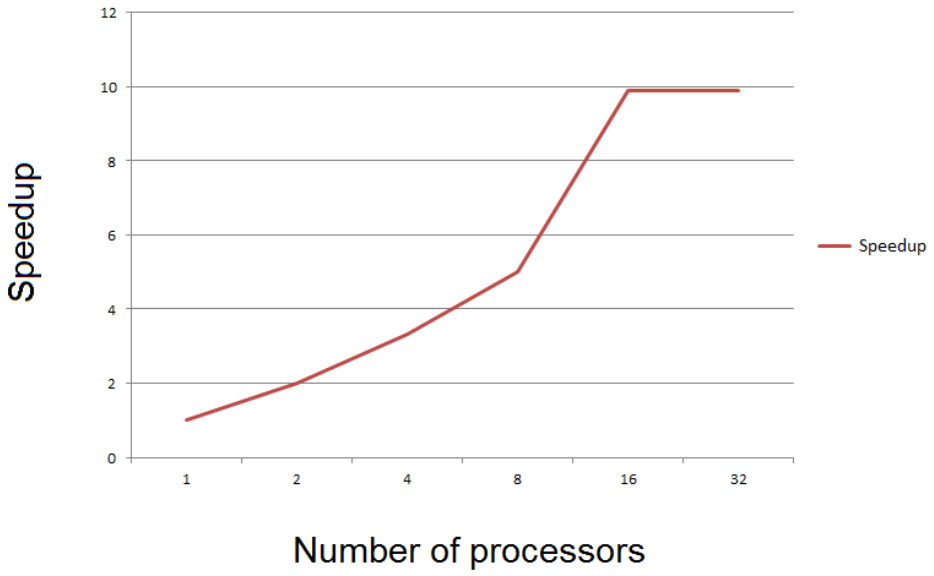



***Figura 24. Speedup en función del número de procesadores***