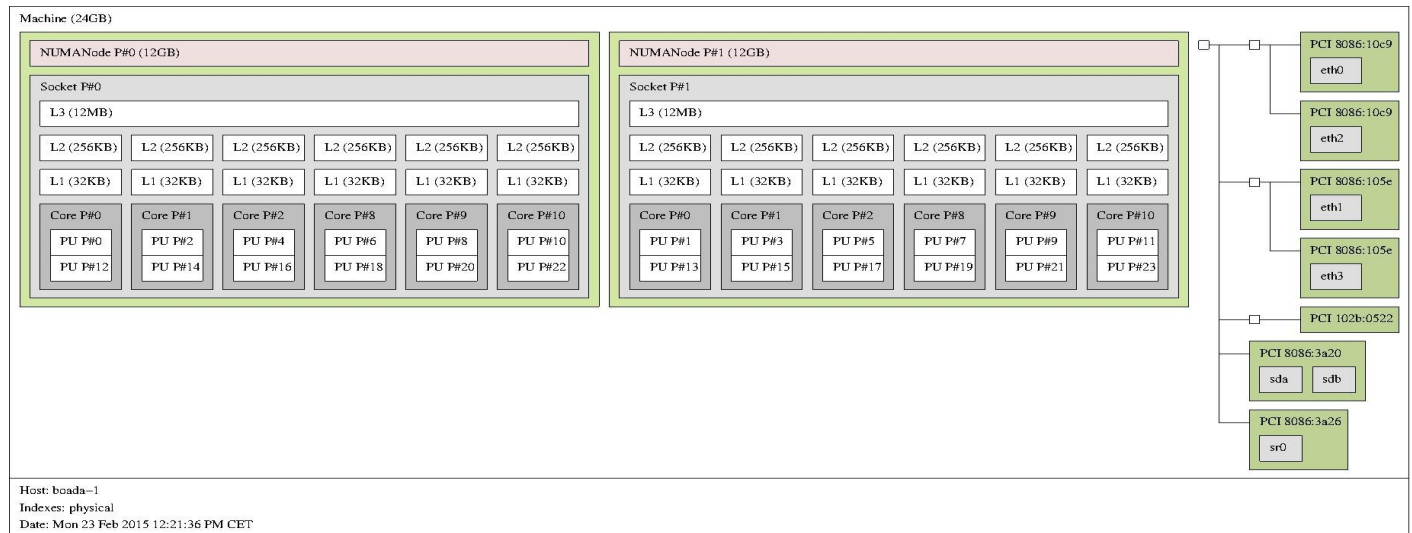


Robert Almar Graupera
Erick Aramayo Monrroy
par1205

First Deliverable

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).



Tenemos una memoria principal de 24 GB, la arquitectura esta dividida en 2 sockets, en cada uno tenemos 1 NumaNode de 12 GB y 6 cores con 2 threads cada uno, cada core cuenta con un primer y segundo nivel de cache privada, L1 de 32 KB y L2 de 256 KB respectivamente, y un tercer nivel de cache, L3 de 12 MB, compartida.

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.

Para poder medir el tiempo entre dos puntos en la ejecución de un programa necesitaremos hacer uso de la librería time del sistema, que la incluiremos mediante el header sys/time.h, además haremos uso de 2 declaraciones:

```
#define START_COUNT_TIME stamp = getusec_();
```

-que pondremos en el punto de inicio de nuestro cálculo.

```
#define STOP_COUNT_TIME(_m) stamp = getusec_() - stamp;\
    stamp = stamp/1e6;\
    printf ("%s: %0.6fs\n",(_m), stamp);
```

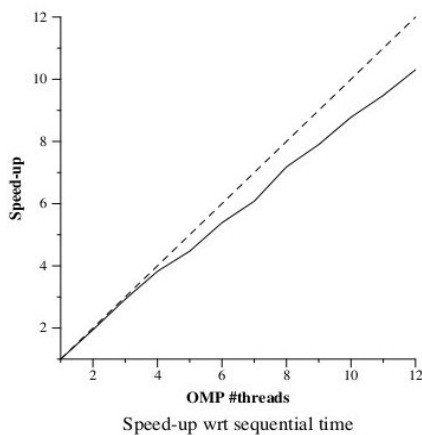
```
#endif
```

-que pondremos en el punto de parada de nuestro cálculo.

Dentro de la función `getusec_()` utilizaremos el struct `timeval` "time", cuyos campos `tv_sec` y `tv_usec` representan los segundos y microsegundos respectivamente, en la función `"gettimeofday"` cuyo resultado será el tiempo desde el 1/1/1970 00:00:00

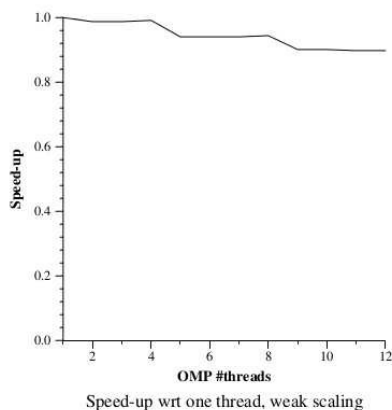
El tiempo de ejecución que queremos será el tiempo transcurrido entre `START_COUNT_TIME` y `STOP_COUNT_TIME` que obtendremos por pantalla.

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

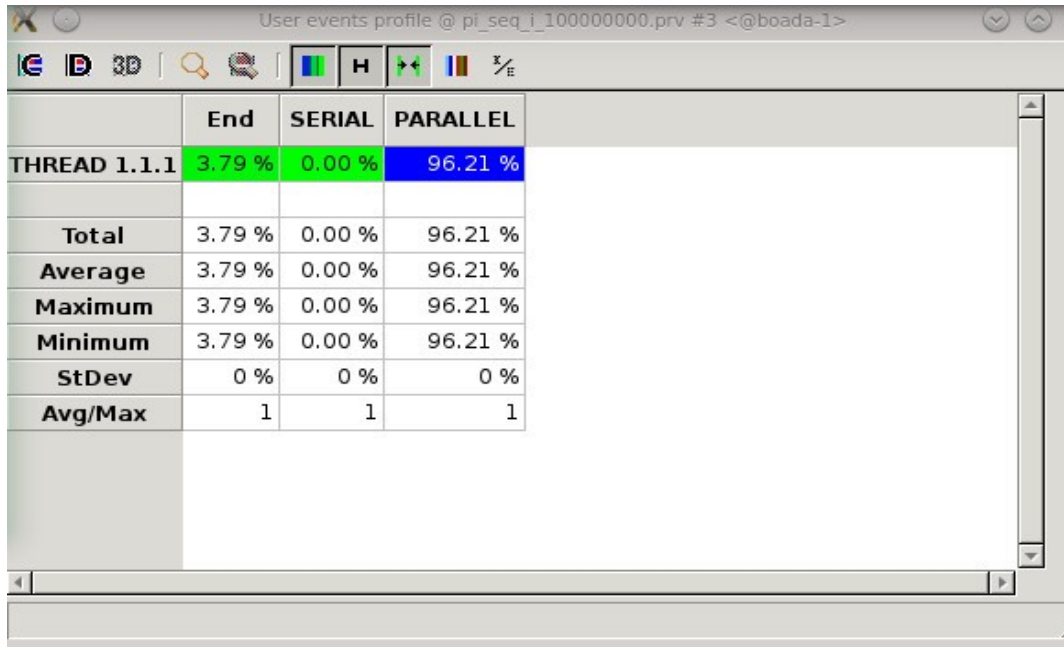
El speedup va incrementando ya que aumentamos el número de threads en cada ejecución, y por lo tanto, como la ejecución del programa se divide entre el número de threads, el tiempo de ejecución del programa se reduce.



Weak scalability

Aquí se ve que la mida va incrementando respecto al número de threads que tenemos disponibles y eso provoca que cuanto más mida haya un speedup menor, es decir, que el programa tarde más

4. 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.



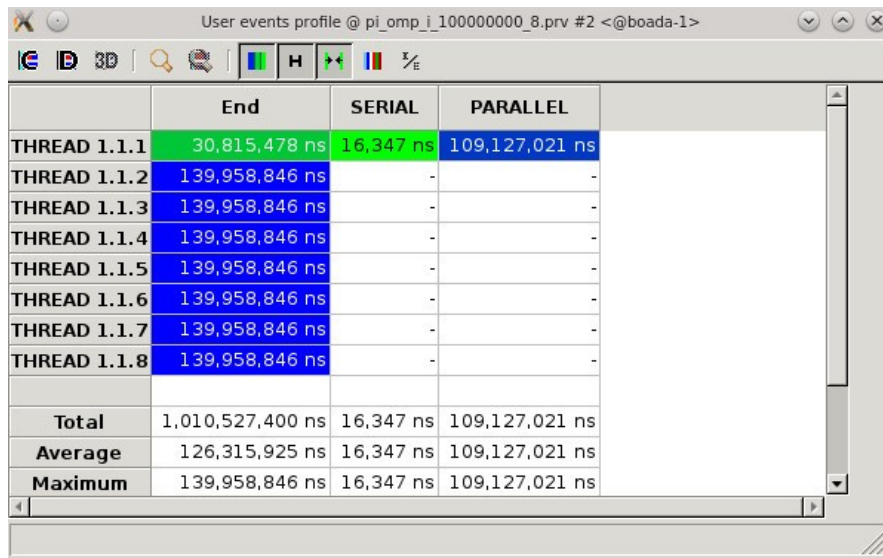
The screenshot shows a window titled "User events profile @ pi_seq_i_100000000.prv #3 <@boada-1>". The window contains a table with the following data:

	End	SERIAL	PARALLEL
THREAD 1.1.1	3.79 %	0.00 %	96.21 %
Total	3.79 %	0.00 %	96.21 %
Average	3.79 %	0.00 %	96.21 %
Maximum	3.79 %	0.00 %	96.21 %
Minimum	3.79 %	0.00 %	96.21 %
StDev	0 %	0 %	0 %
Avg/Max	1	1	1

Primero encolamos el script "submit-seq-i.sh", después de su ejecución abrimos la traza "pi_seq_i_100000000.prv " con paraver, dentro de paraver abrimos la configuración "APP_userevents_profile.cfg" y obtendremos una descripción numérica de la traza. Observamos que la fracción paralela ocupa un 96.21% del tiempo del programa.

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.

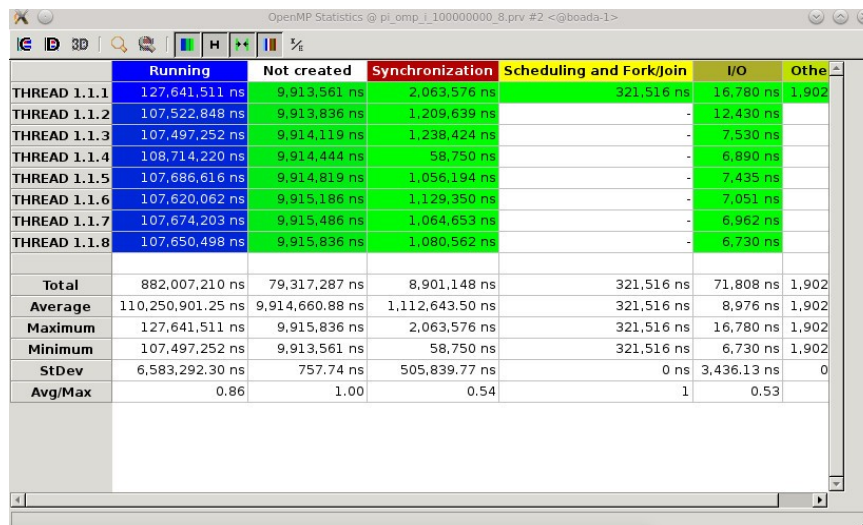
Encolamos el script “submit-omp-i.sh” y abrimos la traza “pi_omp_i_100000000_8.prv” con paraver, luego vamos a configurations y cargamos el fichero de configuración “APP_userevents.cfg” y vemos que hay una zona paralela blanca que tarda 109127021 ns.



User events profile @ pi_omp_i_100000000_8.prv #2 <@boada-1>

	End	SERIAL	PARALLEL
THREAD 1.1.1	30,815,478 ns	16,347 ns	109,127,021 ns
THREAD 1.1.2	139,958,846 ns	-	-
THREAD 1.1.3	139,958,846 ns	-	-
THREAD 1.1.4	139,958,846 ns	-	-
THREAD 1.1.5	139,958,846 ns	-	-
THREAD 1.1.6	139,958,846 ns	-	-
THREAD 1.1.7	139,958,846 ns	-	-
THREAD 1.1.8	139,958,846 ns	-	-
Total	1,010,527,400 ns	16,347 ns	109,127,021 ns
Average	126,315,925 ns	16,347 ns	109,127,021 ns
Maximum	139,958,846 ns	16,347 ns	109,127,021 ns

A partir de aquí abrimos el fichero de configuración “OMP_profile.cfg” con el cual vemos una descripción numérica de el tiempo de running, syncro y fork/join que corresponden a la parte paralela del programa.



OpenMP Statistics @ pi_omp_i_100000000_8.prv #2 <@boada-1>

	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Other
THREAD 1.1.1	127,641,511 ns	9,913,561 ns	2,063,576 ns	321,516 ns	16,780 ns	1,902
THREAD 1.1.2	107,522,848 ns	9,913,836 ns	1,209,639 ns	-	12,430 ns	-
THREAD 1.1.3	107,497,252 ns	9,914,119 ns	1,238,424 ns	-	7,530 ns	-
THREAD 1.1.4	108,714,220 ns	9,914,444 ns	58,750 ns	-	6,890 ns	-
THREAD 1.1.5	107,686,616 ns	9,914,819 ns	1,056,194 ns	-	7,435 ns	-
THREAD 1.1.6	107,620,062 ns	9,915,186 ns	1,129,350 ns	-	7,051 ns	-
THREAD 1.1.7	107,674,203 ns	9,915,486 ns	1,064,653 ns	-	6,962 ns	-
THREAD 1.1.8	107,650,498 ns	9,915,836 ns	1,080,562 ns	-	6,730 ns	-
Total	882,007,210 ns	79,317,287 ns	8,901,148 ns	321,516 ns	71,808 ns	1,902
Average	110,250,901.25 ns	9,914,660.88 ns	1,112,643.50 ns	321,516 ns	8,976 ns	1,902
Maximum	127,641,511 ns	9,915,836 ns	2,063,576 ns	321,516 ns	16,780 ns	1,902
Minimum	107,497,252 ns	9,913,561 ns	58,750 ns	321,516 ns	6,730 ns	1,902
StDev	6,583,292.30 ns	757.74 ns	505,839.77 ns	0 ns	3,436.13 ns	0
Avg/Max	0.86	1.00	0.54	1	0.53	

Para saber los porcentajes de cada parte de la ejecución paralela, hemos hecho los siguientes cálculos:

(fork/join) $\text{join-fork/tiempo paralelo} = 321,516/109,127,021 = 0,29\%$

(synchronization) $\text{synchronization/tiempo paralelo} = 2,063,576/109,127,021 = 1,89\%$

(running) $\text{tiempofork} + \text{tiempo sync} = 2,18\%$; $100-2,18 = 97,81\%$

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).

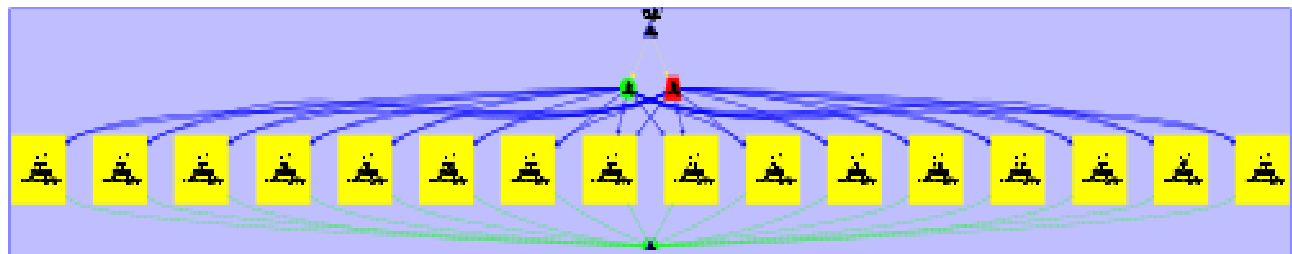
```
void dot_product (long N, double A[N], double B[N], double *acc){
    double prod;
    *acc=0.0;
    int i;

    for (i=0; i<N; i++) {
        tareador_start_task("LOOP");
        tareador_disable_object(acc);
        prod = my_func(A[i], B[i]);
        *acc += prod;
        tareador_enable_object(acc);
        tareador_end_task("LOOP");
    }
}
```

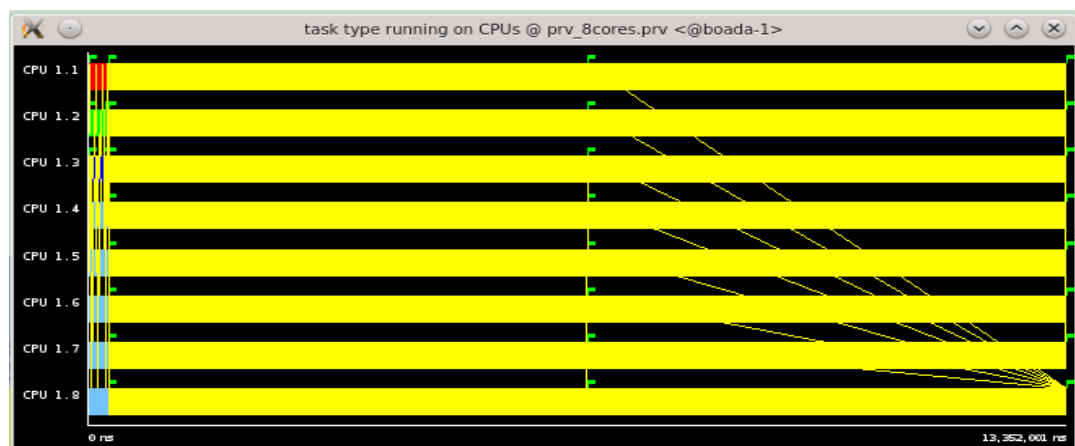
- "tareador_start_task("LOOP")" : sirve para identificar una tarea del grafo producido por tareador
- "tareador_disable_object(acc)" : elimina las dependencias que hay con "acc"

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

Grafo de dependencia:



Timeline:



Para conseguirlo hemos ejecutado el siguiente comando “./run_tareador.sh dot_product” el cual nos da el grafo de dependencia del programa “dot_product” y para el timeline hemos clicado a “view simulation”, y debemos poner 8 procesadores.

9. Complete the following table for the initial and different versions generated for 3dfft seq.c

Para calcular T1 hemos tenido en cuenta las instrucciones totales del programa y para calcular T infinito hemos sumado el máximo de cada nivel.

Version	T1	T infinito	Paralelismo
seq	593772000 ns	593758000 ns	1.00
v1	593772000 ns	593758000 ns	1.00
v2	593772000 ns	315523000 ns	1.88
v3	593772000 ns	109063000 ns	5.44
v4	593772000 ns	60148000 ns	9.87

v1:

```
START_COUNT_TIME;

tareador_start_task("ffts1_and_transpositions");
tareador_start_task("1");
ffts1_planes(pld, in_fftw);
tareador_end_task("1");
tareador_start_task("2");
transpose_xy_planes(tmp_fftw, in_fftw);
tareador_end_task("2");
tareador_start_task("3");
ffts1_planes(pld, tmp_fftw);
tareador_end_task("3");
tareador_start_task("4");
transpose_zx_planes(in_fftw, tmp_fftw);
tareador_end_task("4");
tareador_start_task("5");
ffts1_planes(pld, in_fftw);
tareador_end_task("5");
tareador_start_task("6");
transpose_zx_planes(tmp_fftw, in_fftw);
tareador_end_task("6");
tareador_start_task("7");
transpose_xy_planes(in_fftw, tmp_fftw);
tareador_end_task("7");
tareador_end_task("ffts1_and_transpositions");

STOP_COUNT_TIME("Execution FFT3D");
```

v2:

```
void ffts1_planes(fftwf_plan pld, 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( pld, (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("LOOP");
        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("LOOP");
    }
}

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("LOOP2");
        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("LOOP2");
    }
}
```

v4:

```
void init_complex_grid(fftwf_complex in_fftw[][N][N])
{
    int k,j,i;

    for (k = 0; k < N; k++) {
        tareador_start_task("loop3");
        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("loop3");
}
}
```

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).

Para obtener cada tiempo de ejecución hemos ejecutado los siguientes comandos:

- `./run_tareador 3dfft_seq` para la versión secuencial.
- `./run_tareador 3dfft_v4` para las versiones paralelas con “x” cores.

Después en la opción view simulation escogemos cuantos cores vamos a usar para la simulación. Se abrirá un timeline en paraver del cual hemos obtenido el tiempo de ejecución del programa.

Para el Speed-up, en cada caso, hemos dividido el tiempo de 3dfft_seq(que siempre es el más lento) por el tiempo de 3dfft_v4.

3dfft	TIEMPO DE EJECUCIÓN	SPEED UP
seq	593664000	
2c	297147000	1,9978798373
4c	177972000	3,3357157306
8c	118682000	5,0021401729
16c	59904000	9,9102564103
32c	59904000	9,9102564103

Como podemos observar a partir de 16 cores el programa 3dfft_v4 no puede paralelizarse más.

