Práctica 4

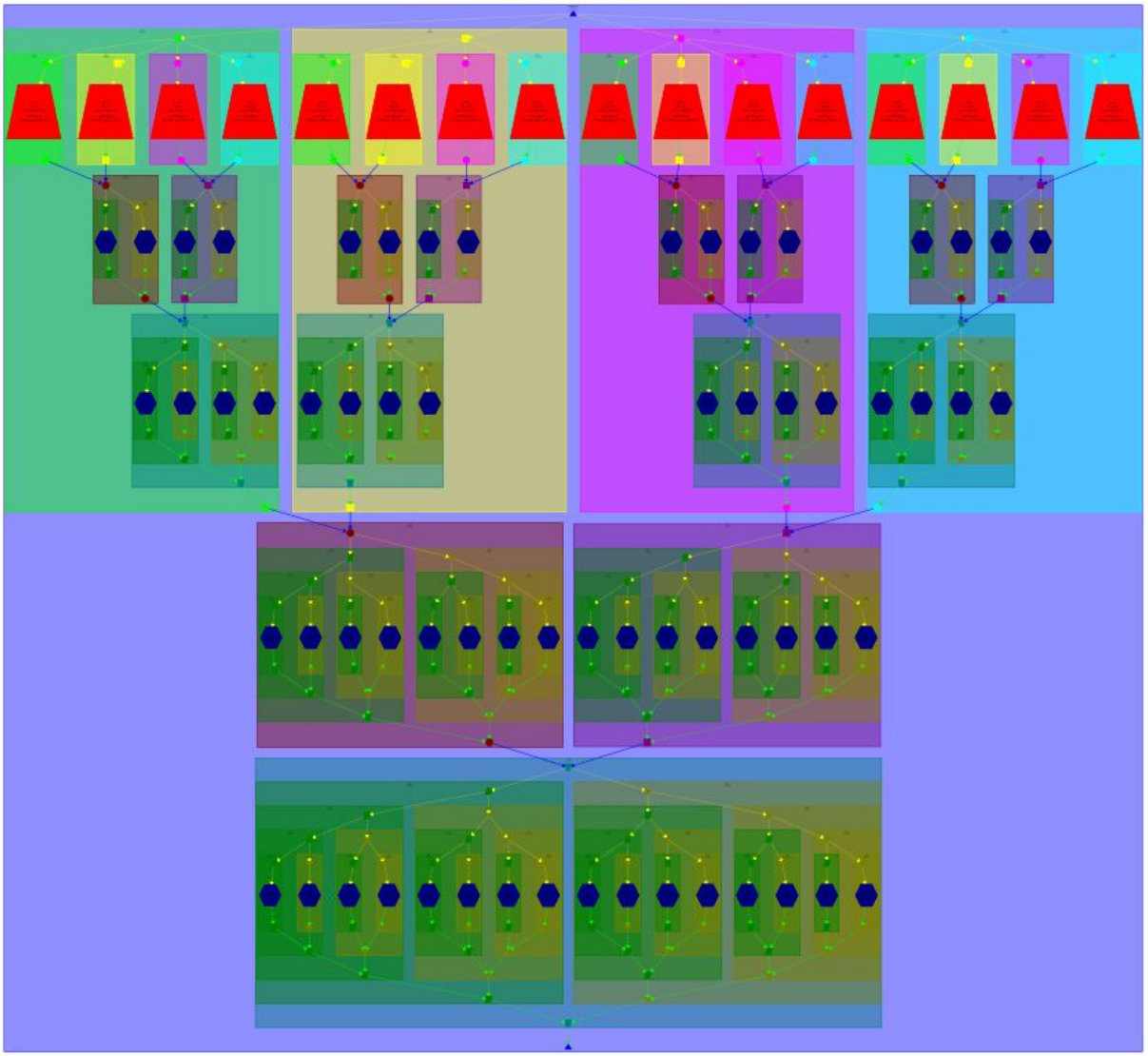
Noel Carmona García

Analysis with tareador

1. Include the relevant parts of the modified `multisort-tareador.c` code and comment where the calls to the Tareador API have been placed. Comment also about the task graph generated and the causes of the dependences that appear.

```
void merge(long n, T left[n], T right[n], T result[n*2], long start, long length) {
        if (length < MIN_MERGE_SIZE*2L) {
                // Base case
                tareador_start_task("bm1");
                basicmerge(n, left, right, result, start, length);
                tareador_end_task("bm1");
        } else {
                // Recursive decomposition
                tareador_start_task("rm1");
                merge(n, left, right, result, start, length/2);
                tareador_end_task("rm1");

                tareador_start_task("rm2");
                merge(n, left, right, result, start + length/2, length/2);
                tareador_end_task("rm2");
        }
}
```

```
void multisort(long n, T data[n], T tmp[n]) {
       if (n >= MIN_SORT_SIZE*4L) {
               // Recursive decomposition
               tareador_start_task("mu1");
               multisort(n/4L, &data[0], &tmp[0]);
               tareador_end_task("mu1");

               tareador_start_task("mu2");
               multisort(n/4L, &data[n/4L], &tmp[n/4L]);
               tareador_end_task("mu2");

               tareador_start_task("mu3");
               multisort(n/4L, &data[n/2L], &tmp[n/2L]);
               tareador_end_task("mu4");

               tareador_start_task("mu5");
               multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L]);
               tareador_end_task("mu5");

               tareador_start_task("me1");
               merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);
               tareador_end_task("me1");

               tareador_start_task("me2");
               merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L);
               tareador_end_task("me2");

               tareador_start_task("me3");
               merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
               tareador_end_task("me3");
       } else {
               // Base case
               tareador_start_task("base");
               basicsort(n, data);
               tareador_end_task("base");
       }
```



Se ha creado una tarea para cada función recursiva del código, las dependencias se deben a que no podemos hacer merge si aún no tenemos la parte del vector ordenada.

2. Write a table with the execution time and speed-up predicted by Tareador (for 1, 2, 4, 8, 16, 32 and 64 processors) for the task decomposition specified with Tareador. Are the results close to the ideal case? Reason about your answer.

Núm. procesadores	Tiempo de ejecución	Speed-up
1	20,334,421,001ns	1
2	10,173,712,001ns	1,99872
4	5,086,801,001ns	3,99748
8	2,550,377,001ns	7,97310
16	1,289,899,001ns	15,7643
32	1,289,899,001ns	15,7643
64	1,289,889,001ns	15,7643

Podemos observar que a medida que duplicamos el número de procesadores el tiempo de ejecución disminuye aproximadamente a la mitad. A partir de los 16 procesadores no es necesario añadir más ya que el speed-up se mantendrá constante a 15,7643.

Parallelization and performance analysis with tasks

1. Include the relevant portion of the codes that implement the two versions (Leaf and Tree), commenting whatever necessary.

Leaf

```
#pragma omp parallel
#pragma omp single
multisort(N, data, tmp);
```

```
void merge(long n, T left[n], T right[n], T result[n*2], long start, long length) {
        if (length < MIN_MERGE_SIZE*2L) {
                // Base case
                #pragma omp task
                basicmerge(n, left, right, result, start, length);
        } else {
                // Recursive decomposition
                merge(n, left, right, result, start, length/2);
                merge(n, left, right, result, start + length/2, length/2);
                #pragma omp taskwait
        }
}

void multisort(long n, T data[n], T tmp[n]) {
        if (n >= MIN_SORT_SIZE*4L) {
                // Recursive decomposition
                multisort(n/4L, &data[0], &tmp[0]);
                multisort(n/4L, &data[n/4L], &tmp[n/4L]);
                multisort(n/4L, &data[n/2L], &tmp[n/2L]);
                multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L]);
                #pragma omp taskwait

                merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);
                merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L);
                #pragma omp taskwait

                merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);

        } else {
                // Base case
                #pragma omp task
                basicsort(n, data);
        }
}
```

En la versión leaf creamos tareas en los casos base. En la función merge ponemos un taskwait al final del caso recursivo y en multisort creamos taskwait justo después de hacer las llamadas recursivas de multisort y de merge.

Tree

```
void merge(long n, T left[n], T right[n], T result[n*2], long start, long length) {
        if (length < MIN_MERGE_SIZE*2L) {
                // Base case
                basicmerge(n, left, right, result, start, length);
        } else {
                // Recursive decomposition
                #pragma omp taskgroup
                {
                        #pragma omp task
                        merge(n, left, right, result, start, length/2);

                        #pragma omp task
                        merge(n, left, right, result, start + length/2, length/2);
                }
        }
}
```

```
void multisort(long n, T data[n], T tmp[n]) {
        if (n >= MIN_SORT_SIZE*4L) {
                // Recursive decomposition
                #pragma omp taskgroup
                {
                        #pragma omp task
                        multisort(n/4L, &data[0], &tmp[0]);

                        #pragma omp task
                        multisort(n/4L, &data[n/4L], &tmp[n/4L]);

                        #pragma omp task
                        multisort(n/4L, &data[n/2L], &tmp[n/2L]);

                        #pragma omp task
                        multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L]);

                }

                #pragma omp taskgroup
                {
                        #pragma omp task
                        merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);

                        #pragma omp task
                        merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L);
                }
                        #pragma omp task
                        merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);

        } else {
                // Base case
                basicsort(n, data);
        }
}
```

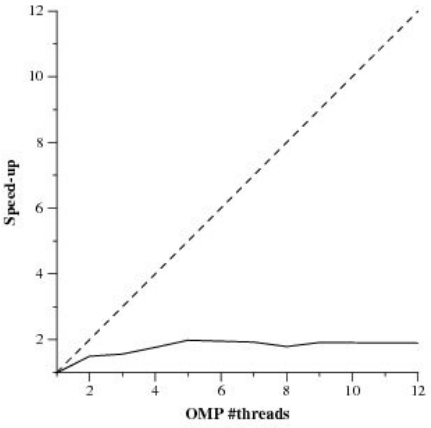
```
#pragma omp parallel
#pragma omp single
multisort(N, data, tmp);
```

En la versión Tree generamos una tarea para cada función recursiva. Tanto en la versión leaf como en la Tree tenemos que indicar que la función multisort contendrá la zona paralela y que la primera llamada al multisort la hará un solo procesador.

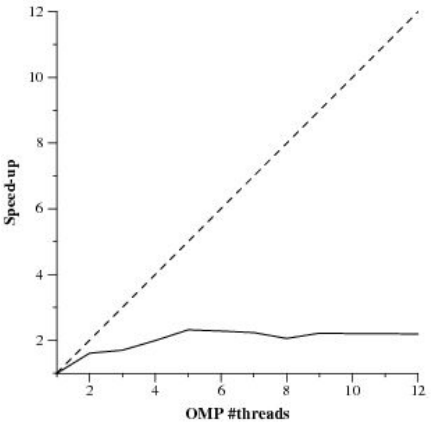
En la versión leaf creamos una tarea para los casos base y dejamos el recorrido del árbol para un único thread.

2.For the the Leaf and Tree strategies, include the speed–up (strong scalability) plots that have been obtained for the different numbers of processors. Reason about the performance that is observed, including captures of Paraver windows to justify your explanations.

Leaf

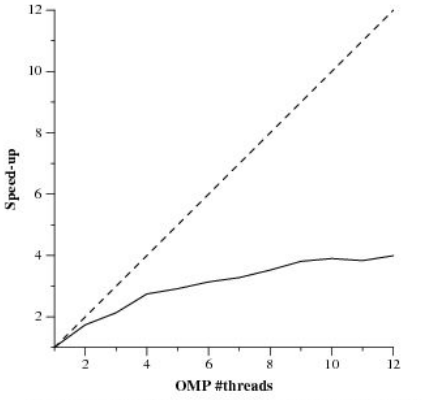


Speed-up wrt sequential time (complete application)

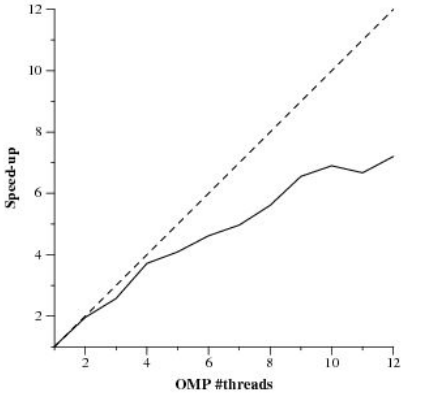


Speed-up wrt sequential time (multisort funtion only)

Tree



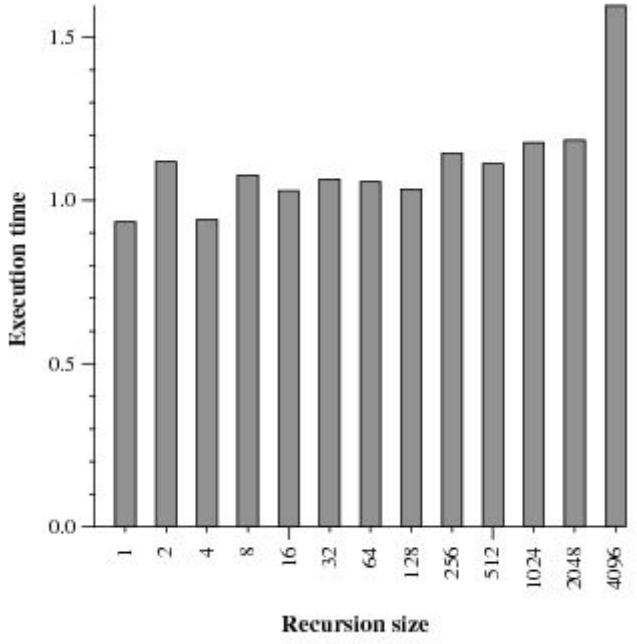
Speed-up wrt sequential time (complete application)



Speed-up wrt sequential time (multisort funtion only)

La versión tree obtiene mejores resultados ya que en esta versión no tenemos una sola tarea que hace el recorrido, tenemos una tarea para cada llamada recursiva, así que es normal que los resultados sean mejores que en la versión leaf.

3. Analyze the influence of the recursivity depth in the Tree version, including the execution time plot, when changing the recursion depth and using 8 threads. Reason about the behavior observed. Is there an optimal value?



Average elapsed execution time (multisort only)

Tiene que existir un valor óptimo, sin embargo la eficiencia que se consigue se pierde con la propia variabilidad de la ejecución.

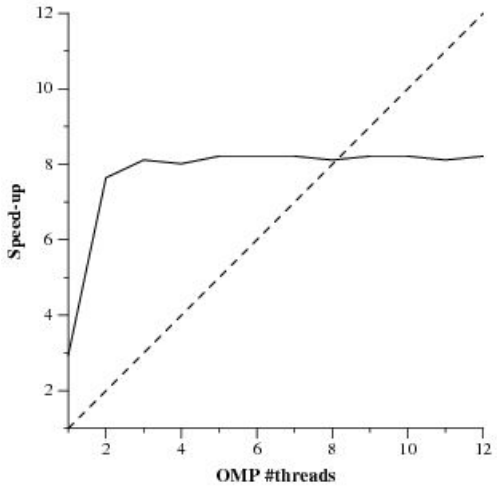
Parallelization and performance analysis with dependent tasks

1. Include the relevant portion of the code that implements the Tree version with task dependencies, commenting whatever necessary.

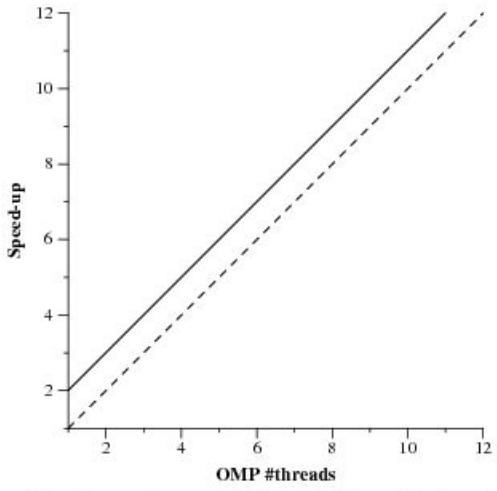
```
void multisort(long n, T data[n], T tmp[n]) {
        if (n >= MIN_SORT_SIZE*4L) {

                // Recursive decomposition
                #pragma omp task depend(out: data[0])
                multisort(n/4L, &data[0], &tmp[0]);

                #pragma omp task depend(out: data[n/4L])
                multisort(n/4L, &data[n/4L], &tmp[n/4L]);

                #pragma omp task depend(out: data[n/2L])
                multisort(n/4L, &data[n/2L], &tmp[n/2L]);

                #pragma omp task depend(out: data[3L*n/4L])
                multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L]);

                #pragma omp task depend( in: data[0], data[n/4L]) depend( out: tmp[0])
                merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);

                #pragma omp task depend( in: data[n/2L], data[3L*n/4L]) depend( out: tmp[n/2L])
                merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L);

                #pragma omp task depend( in: tmp[0], tmp[n/2L])
                merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);

        } else {
                // Base case
                basicsort(n, data);
        }
}
```

Al utilizar dependencia de tareas no necesitamos utilizar taskwait ya que en el momento que una tarea ha terminado y puede satisfacer las dependencias de otras tareas estas se ejecutarán.

2.Reason about the performance that is observed, including the speed–up plots that have been obtained different numbers of processors and with captures of Paraver windows to justify your reasoning.



Speed-up wrt sequential time (complete application)



Speed-up wrt sequential time (multisort funtion only)

Opcionales

1. Complete the parallelization of the Tree version by parallelizing the two functions that initialize the `data` and `tmp` vectors[1] . Analyze the scalability of the new parallel code by looking at the two speed–up plots generated when submitting the `submit-strong-omp.sh` script. Reason about the new performance obtained with support of Paraver timelines.

Tanto para inicializar el vector data como para asignar 0 a todos sus elementos utilizamos un for para recorrer el vector, lo único que podemos paralelizar en estas funciones son los for

```
static void initialize(long length, T data[length]) {
   long i;
   #pragma omp for
   for (i = 0; i < length; i++) {
      if (i==0) {
         data[i] = rand();
      } else {
         data[i] = ((data[i-1]+1) * i * 104723L) % N;
      }
   }
}

static void clear(long length, T data[length]) {
   long i;
   #pragma omp for
   for (i = 0; i < length; i++) {
      data[i] = 0;
   }
}
```

Hacemos lo mismo que hemos hecho con la tarea multisort en el main.

```
#pragma omp parallel
{
        #pragma omp single
        initialize(N, data);

        #pragma omp single
        clear(N, tmp);
}
```