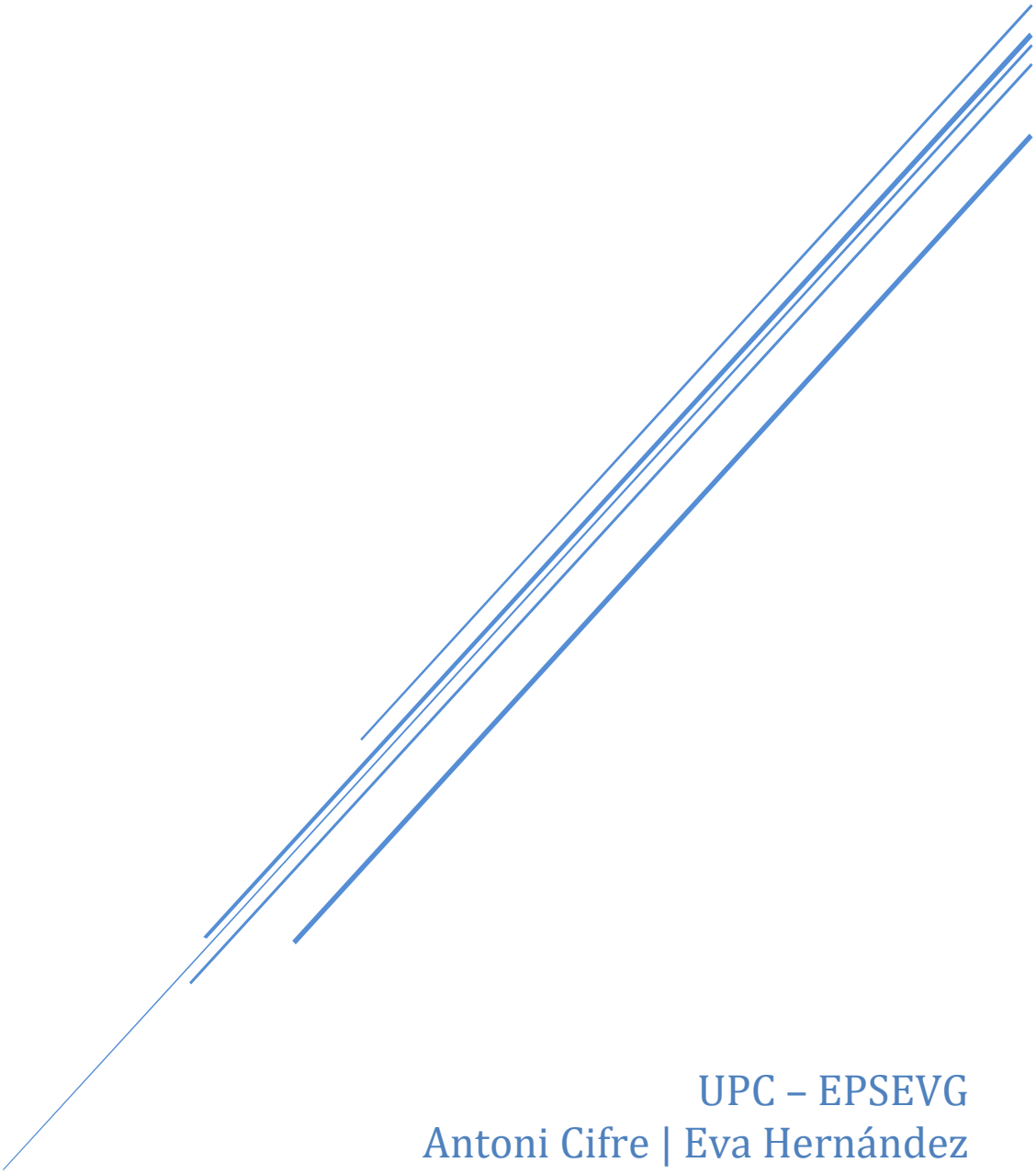
DELIVERABLE

LAB 4
DIVIDE AND CONQUER PARALLELISM WITH OPENMP: SORTING



UPC – EPSEVG
Antoni Cifre | Eva Hernández
Fecha: 9/12/2019
3r año, 1r cuatrimestre

Contenido

Session 1

Task decomposition analysis for Mergesort

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.

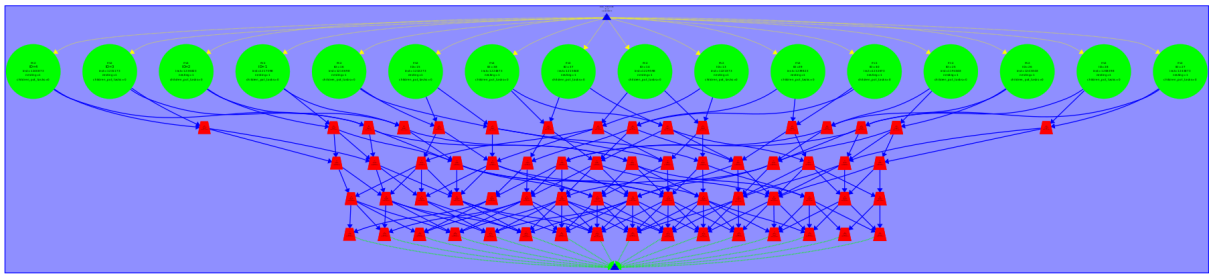
Para observar cual es la mayor manera de paralelizar hemos hecho dos métodos de paralelización, por rama i por hoja. En este caso paralelizaremos las funciones de multisort y merge y observaremos el grafico generado por el tareador para poder generar una buena estrategia de paralelización a la hora de aplicarlo con el open omp.

Leaf:

Code:

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

        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);

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

Graph

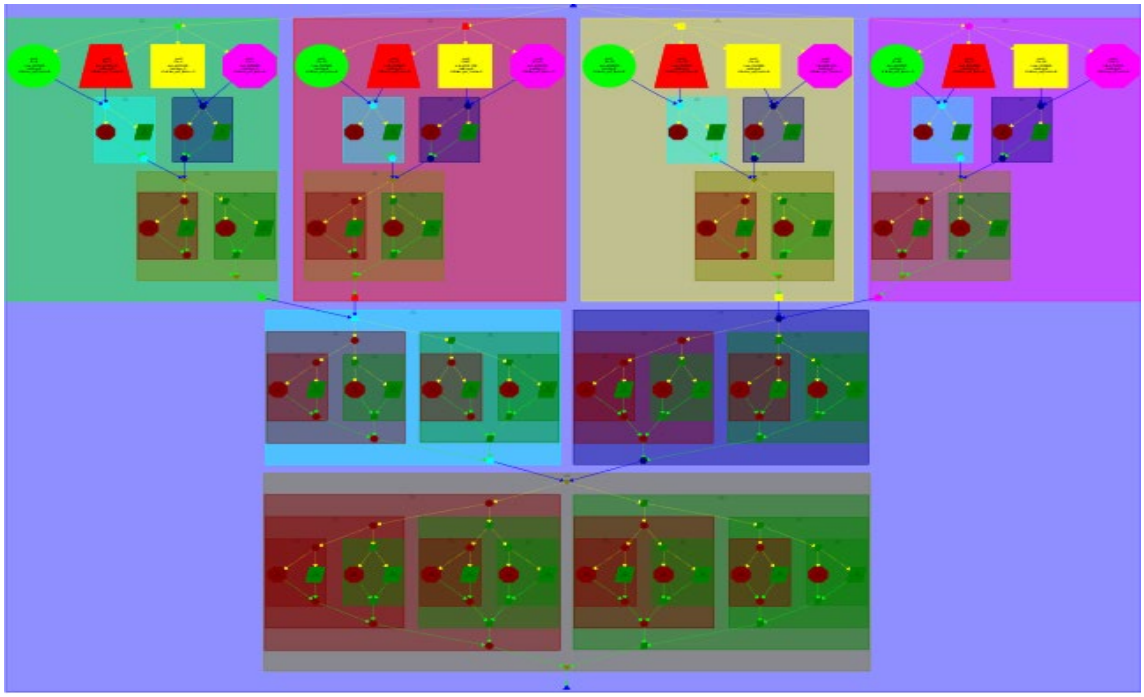


Tree:

Code:

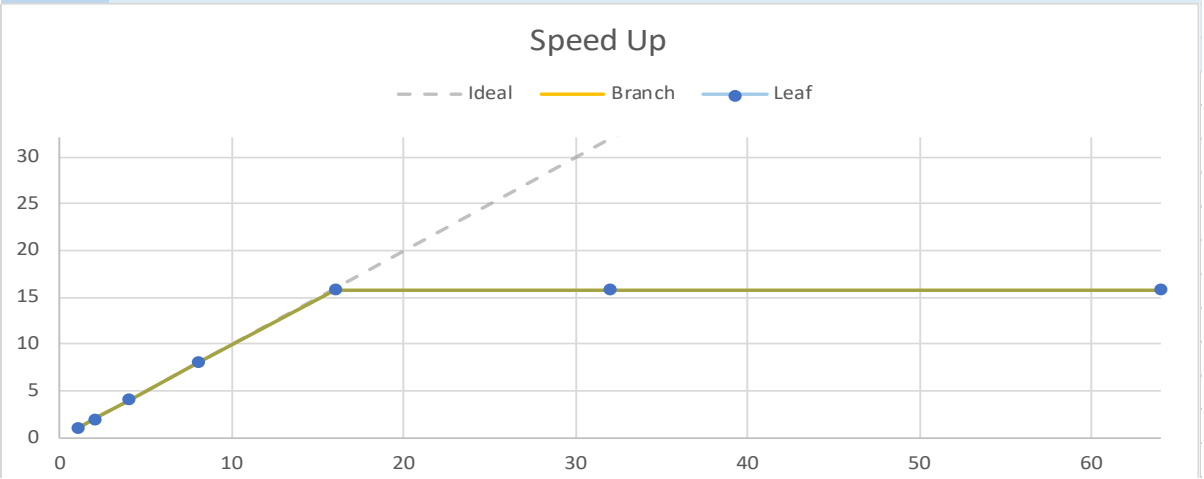
```
void multisort(long n, T data[n], T tmp[n]) {
  if (n >= MIN_SORT_SIZE*4L) {
    tareador_start_task("eva1");
    multisort(n/4L, &data[0], &tmp[0]);
    tareador_end_task("eva1");

    tareador_start_task("eva2");
    multisort(n/4L, &data[n/4L], &tmp[n/4L]);
    tareador_end_task("eva2");

    tareador_start_task("eva3");
    multisort(n/4L, &data[n/2L], &tmp[n/2L]);
    tareador_end_task("eva3");

    tareador_start_task("eva4");
    multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L]);
    tareador_end_task("eva4");

    tareador_start_task("eva5");
    merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);
    tareador_end_task("eva6");

    tareador_start_task("eva7");
    merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L);
    tareador_end_task("eva7");

    tareador_start_task("eva8");
    merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
    tareador_end_task("eva8");
  } else {
    basicsort(n, data);
  }
}
```

Gaph



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.

nºP	Tiempo		Speed-up	
	Leaf	Branch	Leaf	Branch
1	20.334.411 ns	20.334.411 ns	1,000 ns	1,000 ns
2	10.172.763 ns	10.173.716 ns	1,999 ns	1,999 ns
4	5.088.142 ns	5.086.725 ns	3,996 ns	3,998 ns
8	2.547.227 ns	2.550.595 ns	7,983 ns	7,972 ns
16	1.290.557 ns	1.289.922 ns	15,756 ns	15,764 ns
32	1.290.557 ns	1.289.909 ns	15,756 ns	15,764 ns
64	1.290.557 ns	1.289.909 ns	15,756 ns	15,764 ns



Después de ejecutar y ver los resultados generados por el tareador, a simple vista, no se puede llegar a una conclusión de que método es mejor, pero si nos fijamos en el grafo de tareas generadas en lugar de en el tiempo de ejecución, hemos llegado a la conclusión de que será más veloz el método de paralelización por rama debido a que el método de hojas solo aprovecharía 4 hilos, es decir, que si disponemos de más hilos, no se llegaríamos al rendimiento máximo.

Session 2

Shared-memory parallelization with OpenMP tasks

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:

```
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);
        }
}

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

En este método, crearemos las tareas en la parte base del código y crearemos barreras de sincronización donde se generan dependencias de datos.

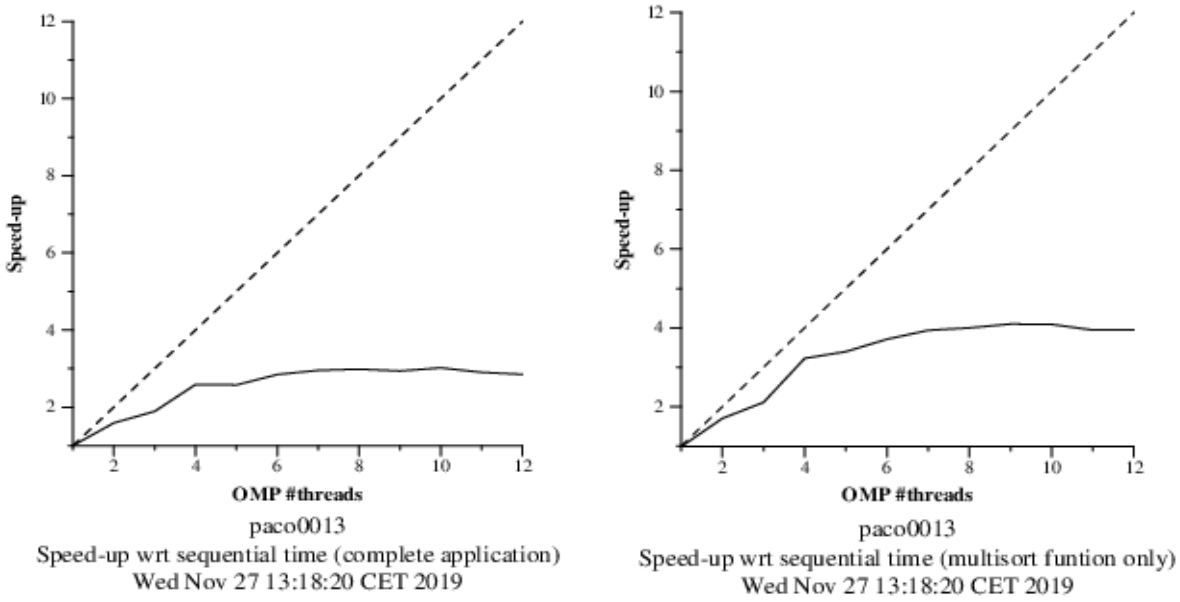
Tree:

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

                #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 taskwait

                #pragma omp task
                merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
                #pragma omp taskwait
        } else {
                // Base case
                basicsort(n, data);
        }
}
```

En esta parte crearemos una tarea por cada llamada a una función dentro de la descomposición recursiva para paralelizar cada una de las ramas del árbol y como en el apartado anterior, crearemos barreras de sincronización en las zonas que crean dependencias de datos.

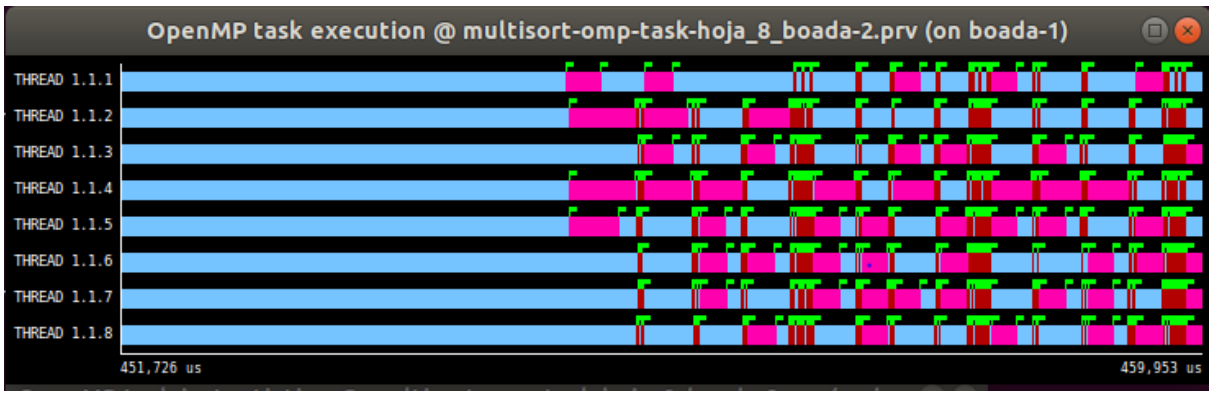
2. For the the Leaf and Tree strategies, include the speed–up (strong scalability) plots that have been obtained for the different numbers of processors. Reason about the performance that is observed, including captures of Paraver windows to justify your explanations.

Leaf strong scalability:



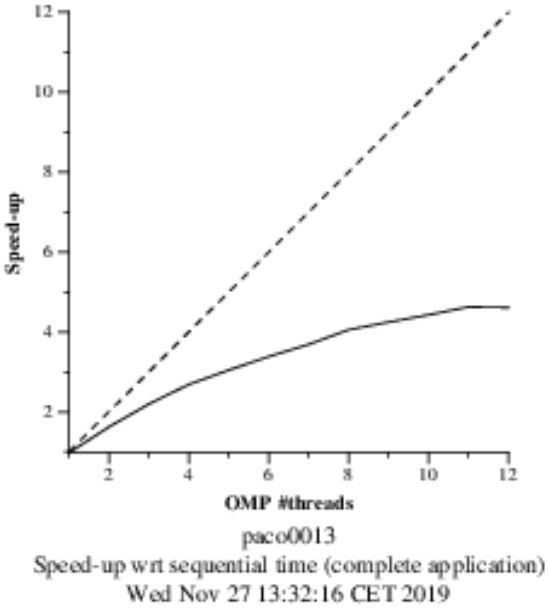
La primera gráfica muestra la escalabilidad de todo el código y la segunda muestra la escalabilidad de la parte que en este caso hemos paralelizado, la cual concuerda con la salida del paraver (límite 4).

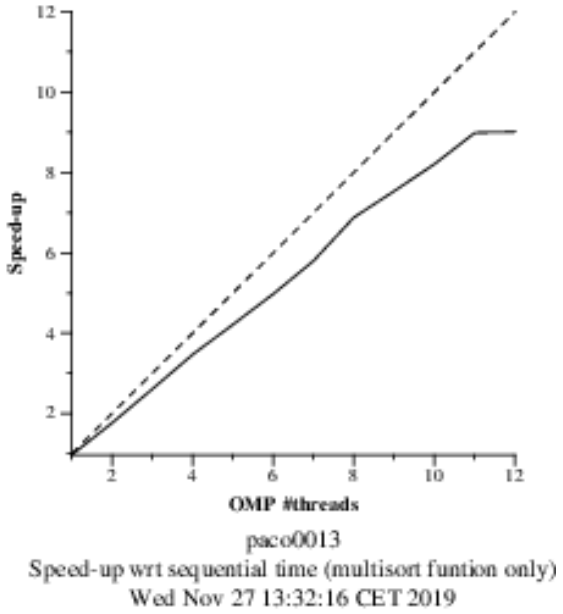
wxparaver hoja:



Como habíamos predicho en la sesión anterior, al dispones mas de 4 hilos no se llegara al máximo rendimiento.

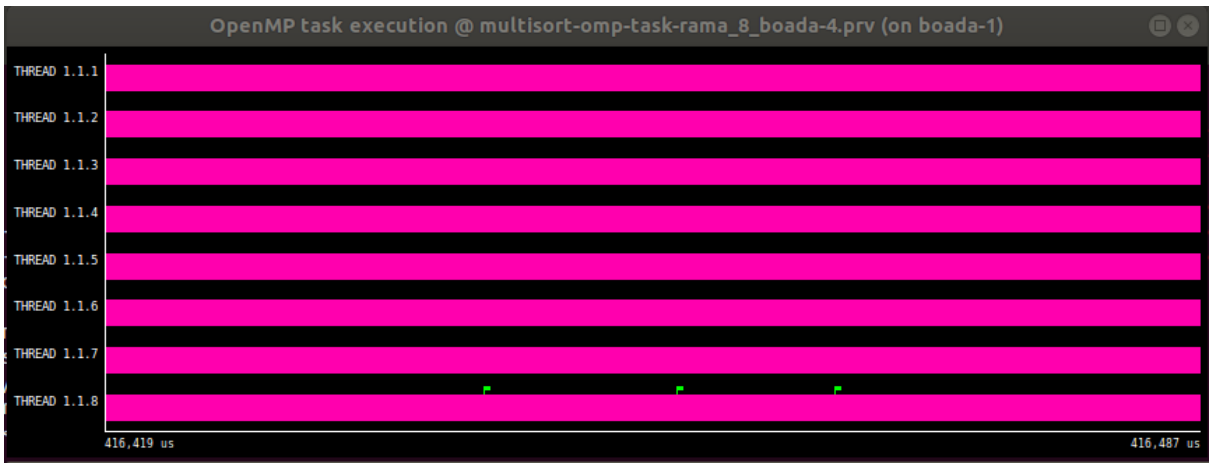
Tree strong scalability:

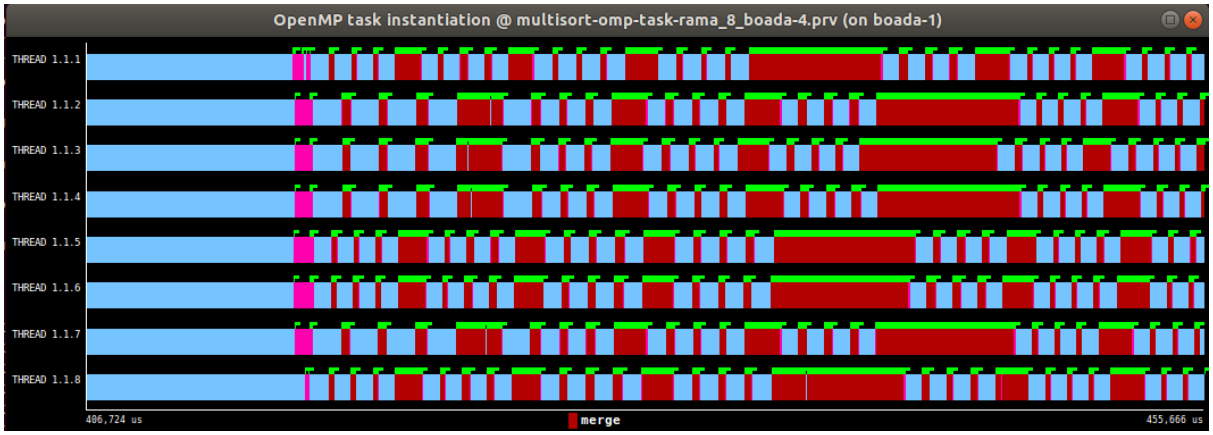




Como se puede observar en relación a la paralelización de las hojas este caso obtiene un speed up notablemente mejor debido a que aprovecha todos los hilos disponibles haciendo una paralelización de hasta los 8 hilos a la vez.

Tree wxparaver:





Como hemos podido observar con las gráficas de speed up, la mejor opción de paralelización seria en rama, debido a que como se puede observar en el paraver, el nombre de tareas que se ejecutan en paralelo en el caso de la hoja es 4 debido a que en la hoja siempre se ejecutan 4 tareas, mientras que en el caso de la rama, el nombre de tareas ejecutada en paralelo ocupan todos los hilos disponibles.
Esto nos aporta una mejor gráfica de escalabilidad, al poder aprovechar todos los hilos, sobre todo a partir de los 4 procesadores.

3. Show the changes you have done in the code in order to include a cut-off mechanism based on recursion level. Is there a value for the cut-off argument that improves the overall performance? Analyze the scalability of your parallelization with this value.

Modificaciones del código:

Multisort

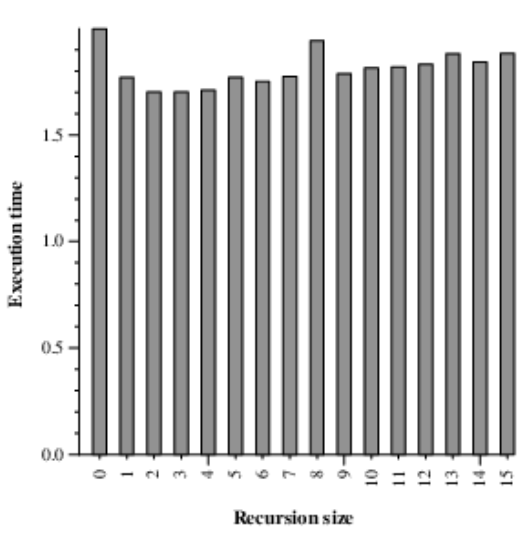
```
void multisort(long n, T data[n], T tmp[n], int depth) {
        if (n >= MIN_SORT_SIZE*4L) {
                // Recursive decomposition
                if(!omp_in_final()){
                        #pragma omp task final (depth >= CUTOFF)
                        multisort(n/4L, &data[0], &tmp[0], depth+1);
                        #pragma omp task final (depth >= CUTOFF)
                        multisort(n/4L, &data[n/4L], &tmp[n/4L], depth+1);
                        #pragma omp task final (depth >= CUTOFF)
                        multisort(n/4L, &data[n/2L], &tmp[n/2L], depth+1);
                        #pragma omp task final (depth >= CUTOFF)
                        multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L], depth+1);
                        #pragma omp taskwait

                        #pragma omp task final (depth >= CUTOFF)
                        merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L,0);
                        #pragma omp task final (depth >= CUTOFF)
                        merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L,0);
                        #pragma omp taskwait

                        #pragma omp task final (depth >= CUTOFF)
                        merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n,0);
                        #pragma omp taskwait
                }else{

                        multisort(n/4L, &data[0], &tmp[0], depth+1);
                        multisort(n/4L, &data[n/4L], &tmp[n/4L], depth+1);
                        multisort(n/4L, &data[n/2L], &tmp[n/2L], depth+1);
                        multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L], depth+1);

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

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

Merge:

```
void merge(long n, T left[n], T right[n], T result[n*2], long start, long length, int depth) {
        if (length < MIN_MERGE_SIZE*2L) {
                // Base case
                basicmerge(n, left, right, result, start, length);
        } else {
                // Recursive decomposition
                if(!omp_in_final()){
                        #pragma omp task final (depth >= CUTOFF)
                        merge(n, left, right, result, start, length/2, depth+1 );
                        #pragma omp task final (depth >= CUTOFF)
                        merge(n, left, right, result, start + length/2, length/2, depth+1);
                        #pragma omp taskwait
                }else{
                        merge(n, left, right, result, start, length/2, depth+1);
                        merge(n, left, right, result, start + length/2, length/2, depth+1);
                }
        }
}
```

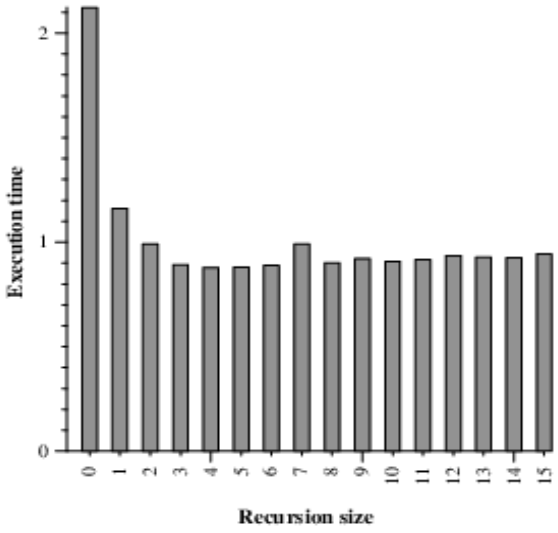
Graficas de la ejecución con diferentes curoff

Cutoff con 4 threads



paco0013
Average elapsed execution time (multisort only)
Wed Dec 4 13:16:56 CET 2019

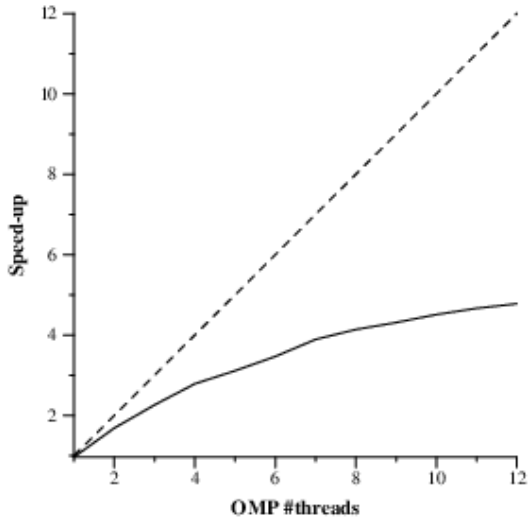
Cutoff con 8 threads



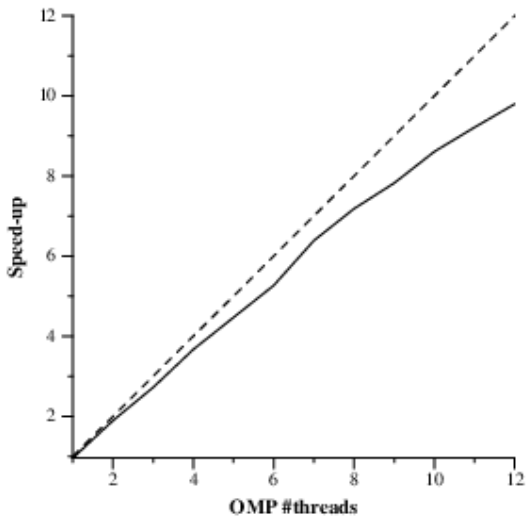
paco0013
Average elapsed execution time (multisort only)
Wed Dec 4 13:24:49 CET 2019

Como se puede observar, el mejor curoff de profundidad 4 y utilizando 8 threads.

Strong scalability con cut-off 4 y 8 threads



paco0013
Speed-up wrt sequential time (complete application)
Wed Dec 4 13:33:59 CET 2019



paco0013
Speed-up wrt sequential time (multisort funtion only)
Wed Dec 4 13:33:59 CET 2019

Session 3

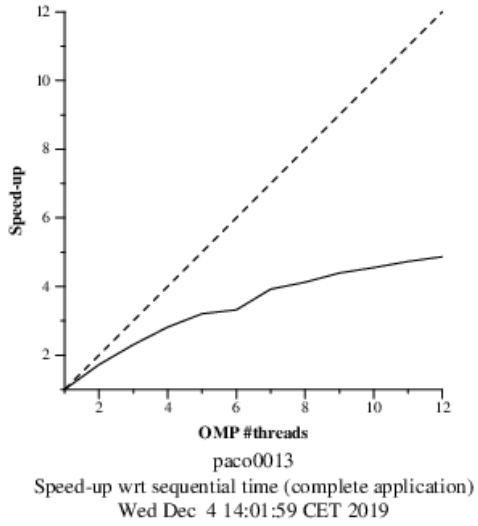
Using OpenMP task dependencies

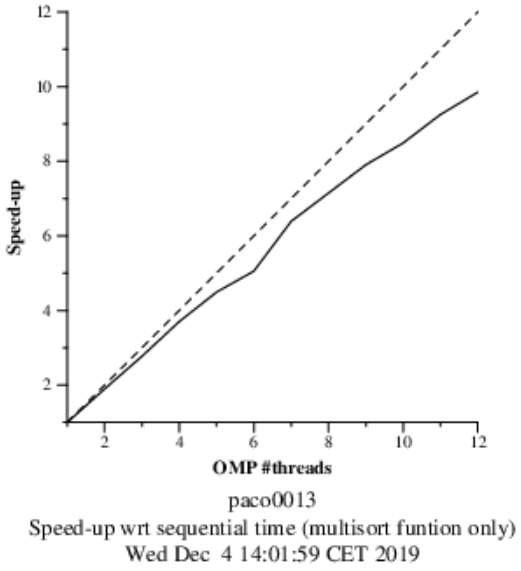
1. Include the relevant portion of the code that implements the Tree version with task dependencies, commenting whatever necessary.

```
void merge(long n, T left[n], T right[n], T result[n*2], long start, long length, int depth) {
  if (length < MIN_MERGE_SIZE*2L) {
    basicmerge(n, left, right, result, start, length);
  }
  else {
    if(!omp_in_final()){
      #pragma omp task final (depth >= CUTOFF)
      merge(n, left, right, result, start, length/2, depth+1 );
      #pragma omp task final (depth >= CUTOFF)
      merge(n, left, right, result, start + length/2, length/2, depth+1);
      #pragma omp taskwait
    }else{
      merge(n, left, right, result, start, length/2, depth+1);
      merge(n, left, right, result, start + length/2, length/2, depth+1);
    }
  }
}
void multisort(long n, T data[n], T tmp[n], int depth) {
  if (n >= MIN_SORT_SIZE*4L) {
    if(!omp_in_final()){
      #pragma omp task final (depth >= CUTOFF) depend(out: data[0])
      multisort(n/4L, &data[0], &tmp[0], depth+1);
      #pragma omp task final (depth >= CUTOFF) depend(out: data[n/4L])
      multisort(n/4L, &data[n/4L], &tmp[n/4L], depth+1);
      #pragma omp task final (depth >= CUTOFF)depend(out: data[n/2L])
      multisort(n/4L, &data[n/2L], &tmp[n/2L], depth+1);
      #pragma omp task final (depth >= CUTOFF)depend(out: data[3L*n/4L])
      multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L], depth+1);

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

      #pragma omp task final (depth >= CUTOFF) depend(in: tmp[0], tmp[n/2L]) depend(out: data[0])
      merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n,0);
      #pragma omp taskwait
    }else{
      multisort(n/4L, &data[0], &tmp[0], depth+1);
      multisort(n/4L, &data[n/4L], &tmp[n/4L], depth+1);
      multisort(n/4L, &data[n/2L], &tmp[n/2L], depth+1);
      multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L], depth+1);

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

      merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n,0);
    }
  } else {
    basicsort(n, data);
  }
}
```

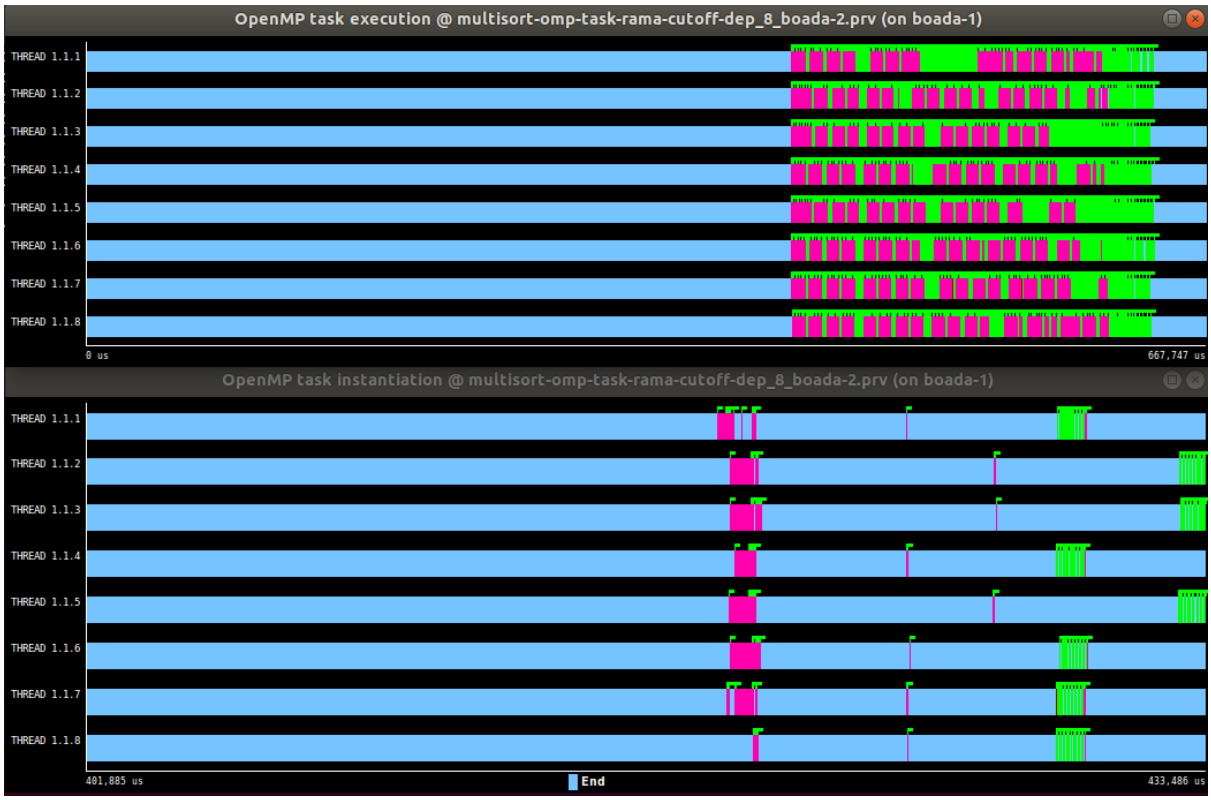
2. Reason about the performance that is observed, including the speed–up plots that have been obtained for different numbers of processors and with captures of Paraver windows to justify your reasoning.

Cut-off and depend





Paraver:



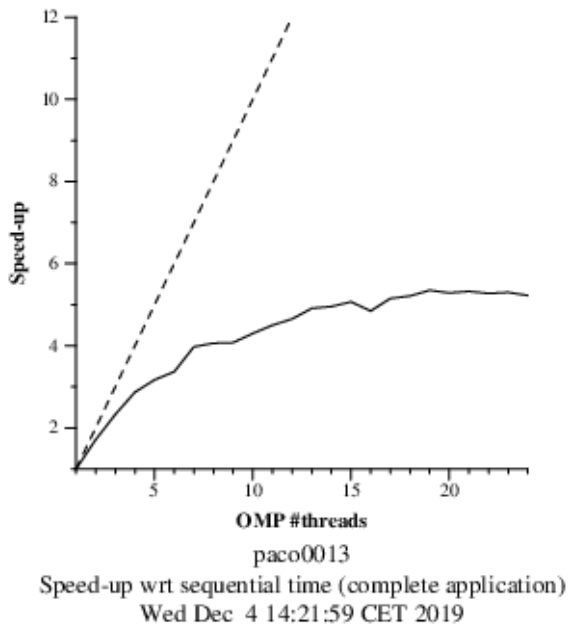
Si comparamos el resultado con el anterior, podremos observar que es mínimamente peor. Todo y eliminar dos barreras de sincronización, el depend, genera 3 barreras de sincronización, una en cada in del depend. todo y ser mas eficientes, al ponerlo en practica podemos observar como en este caso genera un peor resultado.

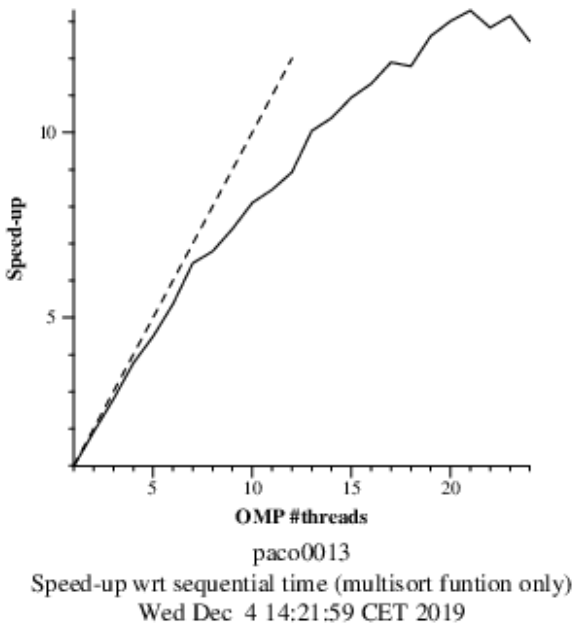
Problemas Opcionales

Optional 1:

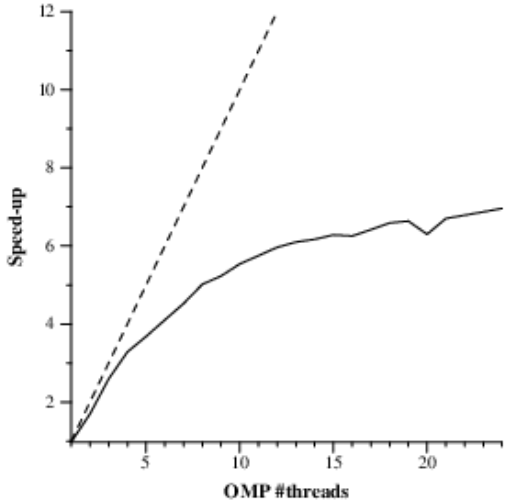
Complete your scalability analysis for the Tree version on the other node types in boada. Remember that the number of cores is different in boada-1 to 4, boada-5 and boada-6 to 8 as well as the enablement of the hyperthreading capability. Set the maximum number of cores to be used (variable np NMAX) by editing submit-strong-omp.sh in order to do the complete analysis. HINT: consider the number of physical/logical cores in each node type and set np NMAX accordingly.

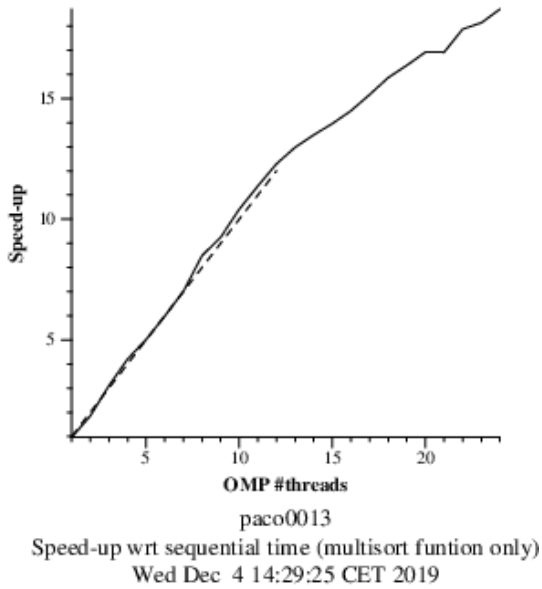
Speed up Boada 1-4 con 24 cores



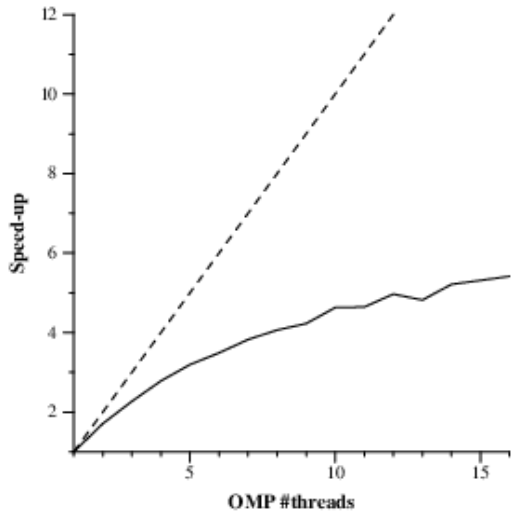


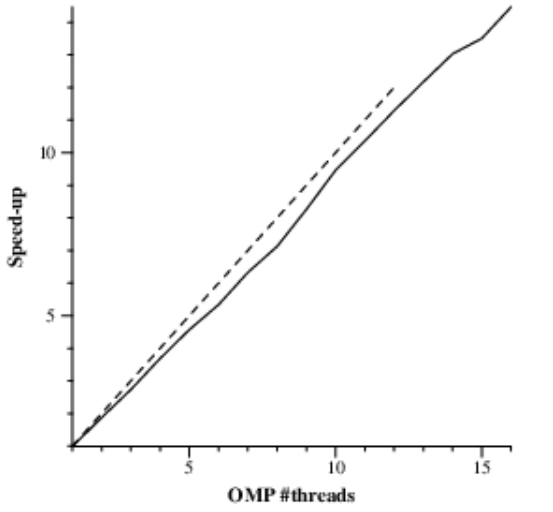
Speed up Boada 5 con 24 cores





Speed up Boada 6-8 con 24 cores

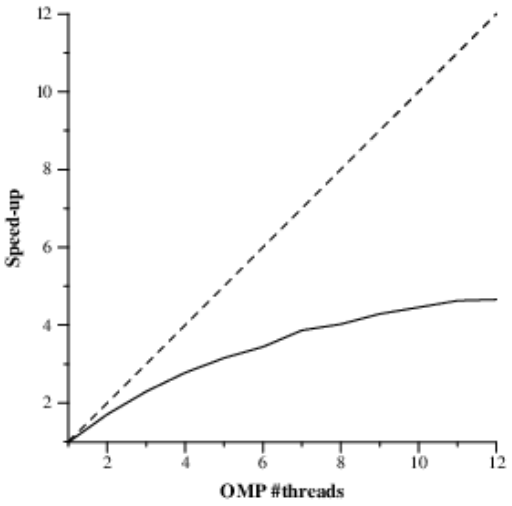


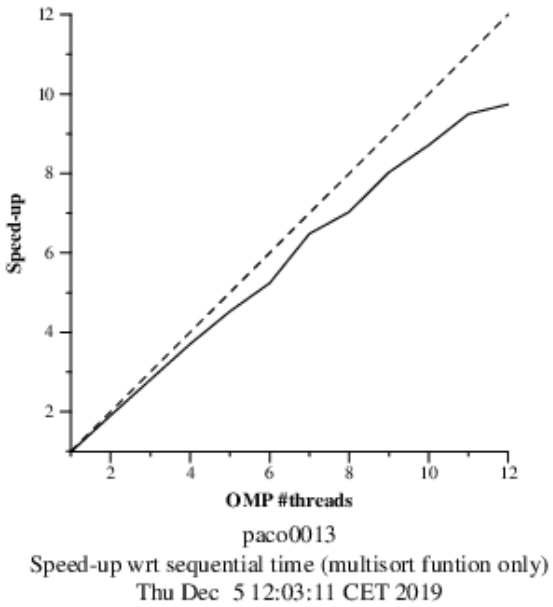


Tal i como podíamos suponer al observar los rendimientos de cada uno de los boada, hemos obtenido un resultado más o menos parecido al esperado según las especificaciones de hardware de cada uno.

Optional 2:

Complete the parallelization of the Tree version by parallelizing the two functions that initialize the data and tmp vectors1 . 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.





Lo hemos conseguido mejorar ligeramente pero el principal problema viene al intentar paralelizar la función de initialize debido a que cada iteración depende de la anterior, esto nos impide hacer una buena paralelización y evita que el tiempo mejore notablemente. Por otra banda la paralelización de la función clear es fácil de implementar debido a que no hay dependencias de datos, pero no nos libera de una gran carga de tiempo debido a que es una función que no genera un coste excesivo de tiempo.