Aproximación paralela a la técnica Divide y Vencerás

A. Dorta, L. García, J. R. González, C. León, C. Rodríguez
Dpto. Estadística, I.O. y Computación
Universidad de La Laguna
38271 La Laguna. Tenerife
e-mail: {ajdorta,lgforte,jrgonzal,cleon,casiano}@ull.es

Resumen

En este trabajo se presenta una propuesta de desarrollo del tema dedicado a la técnica Divide y Vencerás para una asignatura de Programación en Paralelo. Tras una breve introducción a la técnica desde el punto de vista secuencial, se abordan, mediante un ejemplo, las distintas posibilidades de paralelización. El nivel de dificultad se incrementa de forma gradual desde una implementación para Memoria Compartida hasta una para Paso de Mensajes. Finalmente se estudia el rendimiento de las propuestas paralelas frente a las secuenciales.

1. Introducción

La estrategia Divide y Vencerás consiste en descomponer un problema en subproblemas más simples del mismo tipo, resolverlos de forma independiente y una vez obtenidas las soluciones parciales combinarlas para obtener la solución del problema original. En este trabajo se presenta un desarrollo del tema dedicado a esta técnica algorítmica desde el punto de vista paralelo. La propuesta se realiza para la asignatura optativa Programación en Paralelo que se imparte en el último curso de los estudios de Ingeniería Informática de la Universidad de la Laguna.

La metodología seguida en el desarrollo de los ejercicios de laboratorio que se proponen consiste en proporcionar al alumno el guión de la práctica con antelación. Se dedica una o varias sesiones teóricas a la explicación de los conceptos nuevos y de los algoritmos propuestos y una sesión práctica tutorizada para resolver las posibles dudas que se planteen delante del ordenador. Por último, el desarrollo de los ejercicios de laboratorio se plantean de forma *abierta* en el sentido de que los alumnos los desarrollan por su cuenta y tienen un plazo para la entrega de los mismos.

El tema empieza con una breve introducción a la técnica Divide y Vencerás desde el punto de vista secuencial tras la cual se abordan distintas posibilidades de paralelización. El conjunto de laboratorios propuestos se centra en el estudio exhaustivo del algoritmo de "ordenación rápida" (*quicksort*) [4]. Las herramientas utilizada para desarrollar los ejercicios prácticos son: *OpenMP* [7] para el paradigma de Memoria Compartida y *MPI* [6] para el paradigma de Paso de Mensajes. El estudio de los resultados computacionales se realiza utilizando las herramientas *CALL* y *LLAC* [2].

El artículo se estructura siguiendo la secuencia de los laboratorios propuestos. La sección 2 está dedicada a la descripción de la técnica secuencial, la definición del caso de estudio y el planteamiento del problema de paralelización. La sección 3 describe el algoritmo paralelo que se ha de implementar en los laboratorios propuestos. La sección 4 está dedicada al ejercicio de análisis del rendimiento. Finalmente, en la sección 5 se presentan las conclusiones y el trabajo futuro.

2. Paradigma Divide y Vencerás

El pseudocódigo de la Figura 1 refleja la estructura general de un algoritmo basado en la técnica Divide y Vencerás. Dada una función para computar una determinada solución

```
 1 procedure DandC(pbm, sol) {
 2   local var aux;
 3   if easy(pbm)
 4     solve(pbm)
 5   else {
 6     divide(pbm, subpbm, aux);
 7     for i := 1 to k do
 8       DandC(subpbm[i], subsol[i]);
 9     combine(subsol, aux, sol);
10   }
11 }
```

Figura 1: Divide y Vencerás secuencial y recursivo

sobre una entrada de tamaño n la estrategia sugiere que se divida la entrada en k subconjuntos distintos, $1 < k \leq n$, que representen los k subproblemas (línea 6). Una vez resueltos los subproblemas, se debe proporcionar un método que permita combinar las soluciones en una solución del problema total (línea 9). Si los subproblemas se consideran relativamente grandes, entonces se puede volver a aplicar la técnica sobre ellos. Los subproblemas resultantes del diseño Divide y Vencerás son del mismo tipo que el problema original. Por esta razón, la reutilización del principio Divide y Vencerás sobre los subproblemas se expresa de forma natural mediante un procedimiento recursivo (líneas 7 y 8). Debido a que los subproblemas son cada vez más pequeños llegará el momento en el que se produzcan subproblemas con el tamaño adecuado para ser resueltos sin necesidad de dividir (líneas 3 y 4).

El *quicksort* es un ejemplo paradigmático de la aplicación de la estrategia Divide y Vencerás. Se pueden encontrar descripciones secuenciales, recursivas e iterativas del mismo en [8]. Su elección como caso de estudio se debe a que ha sido ampliamente estudiado por los alumnos en diversas asignaturas de la carrera. Dados n elementos almacenados en un array a, la Figura 2 muestra una implementación secuencial del mismo. La invocación del método qs() se realizará desde el programa principal con los índices inicializados para indicar todo el array a (línea 24).

Partiendo de la implementación del *quicksort* de la Figura 2, la obtención de una paralelización mediante *OpenMP* es una tarea sencilla. Basta con modificar el procedimiento qs() tal y como se muestra en la Figura 3 y añadir una cláusula #pragma omp parallel antes de la llamada a qs(a, 1, n) (líneas 19-20).

```
 1 procedure qs(v, first, last) {
 2   local var i, j;
 3   if (first < last) {
 4     (i, j) = partition(v, first, last);
 5     qs(v, first, j);
 6     qs(v, i, last);
 7   }
 8 }
 9
10 procedure partition(v, first, last) {
11   local var i = first, j = last;
12   local var pivot = v[(first + last) / 2];
13   while (i <= j) {
14     while (v[i] < pivot)
15       i++;
16     while (pivot < v[j])
17       j--;
18     if (i <= j)
19       swap(v[i++], v[j--]);
20   }
21   return (i, j);
22 }
23
24 ... qs(a, 1, n); ...
```

Figura 2: Pseudocódigo del QuickSort secuencial

Los cambios en qs() permiten que cada llamada recursiva se asigne a una *sección* distinta de *OpenMP* (líneas 13 y 15). A medida que se avanza en las llamadas recursivas (líneas 14 y 16), el equipo (*team*) de *threads* disponibles se va dividiendo por la mitad. Cada mitad ejecuta una de las llamadas, consiguiendo así un reparto razonable de la carga de trabajo. El procedimiento partition() sólo lo debe ejecutar el *thread* maestro para evitar problemas de acceso concurrente al array v (líneas 4-7). La cláusula firstprivate se encarga de inicializar en todos los *threads* los valores de i y j a los calculados por el *thread* maestro (línea 9). La barrera de la línea 8 permite asegurar que el *thread* maestro ha terminado de calcular los valores correctos de i y j.

Desafortunadamente esta implementación de qs() al mezclar paralelismo y recursividad requiere un compilador de *OpenMP* que soporte *paralelismo anidado*, lo que no ocurre con la mayoría de los compiladores disponibles en la actualidad. Si el código de la Figura 3 se le da como entrada a un compilador sin soporte

```
 1 procedure qs(v, first, last) {
 2   local var i, j;
 3   if (first < last) {
 4     #pragma omp master
 5     {
 6       (i, j) = partition(v, first, last);
 7     }
 8     #pragma omp barrier
 9     #pragma omp parallel firstprivate(i, j)
10     {
11       #pragma omp sections nowait
12       {
13         #pragma omp section
14           qs(v, first, j);
15         #pragma omp section
16           qs(v, i, last);
17 } } } }
18 ...
19 #pragma omp parallel
20 { qs(a, 1 ,n); }
21 ...
```

Figura 3: QuickSort - Paralelización *OpenMP*

de paralelismo anidado no se producen errores. Sin embargo, la implementación que llevan a cabo dichos compiladores consiste en serializar la ejecución asignando un único *thread* para las regiones paralelas anidadas, por lo que finalmente se obtendrá una versión sobrecargada del algoritmo secuencial. Por lo tanto, si el objetivo es obtener un programa paralelo eficiente se ha de diseñar un algoritmo que permita simular el comportamiento del paralelismo anidado. Esta es la tarea que se le plantea al alumno en el primer ejercicio práctico.

3. Laboratorio 1: Simulación de Paralelismo Anidado

Considere el esquema genérico Divide y Vencerás de la Figura 1. La solución planteada se basa en la creación de grupos de procesadores. Inicialmente todos los procesadores disponibles forman parte de un grupo que denominaremos *conjunto raíz*. Cuando tiene lugar la primera división del algoritmo divide y vencerás (línea 6) el conjunto raíz se dividirá en k subconjuntos, asignando a cada uno de ellos un subproblema. A continuación de forma recursiva se procede a la resolución de los k subproblemas (línea 8). Si el subproblema i a resolver no es fácil (línea 3) se procederá a dividirlo (línea 6) lo que provocará que el grupo de procesadores que estaba trabajando en él también se divida dando lugar a nuevos subconjuntos de procesadores. El proceso de división en grupos de procesadores descrito puede representarse mediante un árbol. En la Figura 4 los nodos representan conjuntos de procesadores. La división de los subproblemas y de los grupos de procesadores continúa hasta que se detenga la recursividad. Esto ocurre cuando los subproblemas generados sean considerados fáciles. En ese momento los procesadores en los *nodos hoja* invocan a la función que resuelve los problemas simples (línea 4) para obtener la solución al subproblema. La definición de un problema fácil se hace en función del número de procesadores en el grupo. Así, cuando en un *nodo hoja* sólo hay un procesador, se procede a la resolución del subproblema asignado llamando al algoritmo secuencial.

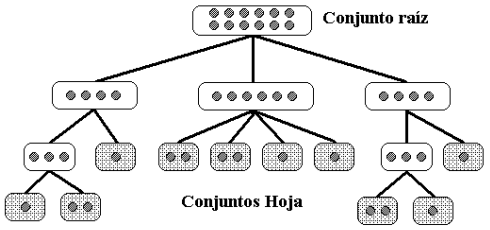

Figura 4: Árbol de conjuntos de procesadores

Al finalizar las llamadas recursivas empieza el proceso de combinación (línea 9). Además de combinar las soluciones parciales, se procederá a la unión de los subgrupos de procesadores para volver a obtener el grupo predecesor. Al finalizar la ejecución del algoritmo se tendrá nuevamente a todos los procesadores en el *conjunto raíz*. Durante el proceso de combinación, en el caso de una implementación con Paso de Mensajes, los grupos de procesadores deben comunicar las soluciones parciales obtenidas al resto de grupos.

El algoritmo descrito se conoce con el nombre de Computación Colectiva. La implementación del mismo de forma genérica para *MPI* puede consultarse en [1].

```
 1 procedure qs(v, first, last) {
 2   local var i, j;
 3   local var addresses[] = {&i, &j};
 4   if (first < last) {
 5     ll_master
 6     {
 7       (i, j) = partition(v, first, last);
 8     }
 9     ll_2_sections(qs(v, first, j),
                      qs(v, i, last),
10       ll_first_private(2, addresses));
11   } }
12 ...
13 #pragma omp parallel
14 { ll_initialize(); qs(a, 1, n); }
15 ...
```

Figura 5: QuickSort - Paralelismo anidado explícito con *OpenMP*

3.1. Implementación con Memoria Compartida

Para el desarrollo del algoritmo descrito en los párrafos previos usando Memoria Compartida, se propone el pseudocódigo de la Figura 5. Nótese que se ha creado a partir del esquema de la Figura 3 eliminado todos los #pragma de *OpenMP* a excepción del de la línea 19 que se conserva en la línea 13 de la Figura 5 para que *OpenMP* realice la creación de los *threads*. Las construcciones *OpenMP* eliminadas del procedimiento qs() se han sustituido por funciones y macros de una librería que ha de implementar el alumno en base a las instrucciones que siguen.

Puesto que se han de controlar los subgrupos de procesadores manualmente, antes de la primera llamada a qs() se realiza una llamada a la función ll_initialize() (línea 14). Esta función creará el *conjunto raíz* a partir de todos los *threads* que genere *OpenMP*.

Asociada a cada *thread* se implementará una pila. En la misma se almacenará la información del subgrupo actual al que pertenece el *thread* antes de profundizar en una nueva división, puesto que dicha información será sustituida por la del nuevo grupo al que dicho *thread* vaya a pertenecer. La información a almacenar ha de incluir el identificador del *thread* en el conjunto actual. Para recuperar la información del conjunto de procesadores durante el proceso de combinación bastará con extraerla de la pila.

La macro ll_2_sections() (línea 9) tiene tres parámetros. Los dos primeros especifican las tareas a realizar en las dos secciones en que se han de repartir los *threads* disponibles, que en el caso del *quicksort* son las llamadas recursivas con los dos subvectores. El tercer parámetro es una llamada a la función ll_first_private() (línea 10). Esta función se encarga de inicializar las variables, cuyas direcciones recibe, al valor que toman en el *thread* maestro.

La forma de proceder de ll_2_sections() consistirá en comprobar si el número de *threads* del subgrupo actual es uno, en cuyo caso dicho *thread* ejecutará ambas secciones una tras otra de forma consecutiva. En otro caso:

- Se copian las variables indicadas en ll_first_private() del *thread* maestro al resto de *threads*.
- Se almacena en la pila la información del subgrupo actual.
- Se dividen los *threads* en subgrupos y cada *thread* almacena localmente los datos del subgrupo que le corresponda.
- Los *threads* del subgrupo 1 ejecutan ahora la primera sección y los *threads* del subgrupo 2, la segunda sección.
- Al finalizar la ejecución de las secciones ambos subgrupos se unen nuevamente en uno recuperando la información del subgrupo de nivel superior de su pila.

Este planteamiento del ejercicio le da al alumno libertad para, dentro del mismo esquema general, desarrollar las tareas de la forma que quiera, poniendo en práctica un aprendizaje por descubrimiento.

Como trabajo opcional se puede proponer la implementación de barreras a nivel de grupo. La necesidad de las mismas puede aparecer en distintos puntos del programa dependiendo de la implementación realizada. Por ejemplo, se pueden utilizar para esperar a que termine la copia de las variables del *thread* maestro, al dividir en subgrupos para no usar la información

```
 1 void qs(int *v, int first, int last) {
 2   if (first < last) {
 3     if(numProc == 1) {
 4       quicksortseq(v, first, last);
 5     }
 6     F=... /* Inicialización */
 7     /* Partición y ejecución */
 8     MPI_Comm_split(...);
 9     qs(v, start[my_grp], end[my_grp]);
10     ...
11     /* Comunicación */
12     for(j=name; j<other_nProcs; j+=my_nProcs)
13       MPI_Send(...)
14     }
15     MPI_Recv(...);
16     ... /* Finalización */
17 } }
```

Figura 6: QuickSort - Paralelismo anidado explícito con *MPI*

de los subgrupos antes de que esté creada, o al reunir los subgrupos en uno para no borrar la información de los subgrupos que dejan de ser utilizados hasta que se esté seguro de que ya no es necesaria, etc.

3.2. Implementación con Paso de Mensajes

Si la implementación de paralelismo anidado supone un problema para el modelo de memoria compartida, para el modelo de paso de mensajes es mayor. Lo que se propone en este ejercicio es la implementación con *MPI* del algoritmo de Computación Colectiva descrito para el caso concreto del *quicksort*. El esquema de modificación del procedimiento `qs()` que se sugiere es el recogido en la Figura 6.

En primer lugar se debe determinar cómo se llevará a cabo la división del conjunto de procesadores disponibles en cada momento (*Fase de Inicialización* - línea 6). Puesto que en el caso del *quicksort* siempre se generarán dos subproblemas, una primera aproximación sería asignar la mitad del número de procesadores a cada subconjunto. Sin embargo, como la elección del pivote es aleatoria es muy probable que el tamaño de los subproblemas no sea el mismo, es decir, lo normal es que uno de los subvectores a ordenar sea de mayor tamaño que el otro. Como el tamaño de cada subvector es conocido se puede utilizar para realizar la asignación de los procesadores a cada subconjunto con *equilibrado de carga*. Por lo tanto, se han de repartir los procesadores proporcionalmente según el tamaño del subvector, pero asegurando siempre que cada subconjunto cuente al menos con un procesador. Para representar los grupos se ha de utilizar un array F con tres elementos. El primer procesador del grupo i tendrá como identificador $F[i]$, y, el último, $F[i+1]-1$, por lo que el número de procesadores puede determinarse a partir de la expresión $numProcs(i) = F[i+1] - F[i] - 1$.

Durante la *Fase de Partición y Ejecución* (líneas 7 y 8), se procederá a generar los dos subconjuntos a partir del conjunto actual. Para ello, se creará un comunicador nuevo para el subconjunto, mediante la función $MPI\_Comm\_split()$ y la información de los grupos guardada en el vector F creado en la fase anterior. Una vez creado el nuevo comunicador, se deberá actualizar la información del identificador del procesador dentro del nuevo grupo y el número de procesadores que lo forman, para luego realizar la llamada recursiva (línea 9). Como los subconjuntos de procesadores deben reunificarse durante la combinación en el conjunto original, adicionalmente, en esta etapa se ha de guardar la información del conjunto antes de la división, para poder restaurarla posteriormente. Esta información debe incluir al vector F y al comunicador existente antes de la división.

Como cada procesador tiene una memoria privada, es necesaria una *Fase de Comunicación* (líneas 11-15), en la que cada subconjunto comunicará los resultados obtenidos al otro subconjunto, de forma que todos los procesadores posean los mismos resultados, condición que deben cumplir para poder reunirse en el conjunto de partida. Durante la comunicación se lleva a cabo el procedimiento de combinación de las soluciones parciales, que se reduce a colocar en el sitio correcto el subvector recibido. Puesto que todos los procesadores de un subconjunto poseen los mismos resultados, se pueden optimizar las comunicaciones y realizarlas también de forma paralela. Un esquema de esta comunicación se muestra en la Figu-

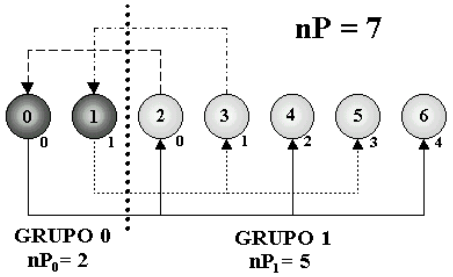

Figura 7: Patrón de comunicaciones

```
 1 int main(int argc, char *argv[])
 2   ...
 3   #pragma cll qs qs[0] + qs[1]*size*log(size)
 4   {
 5     qs(a, 0, size - 1);
 6   }
 7   #pragma cll end qs
 8   ...
 9   #pragma cll report all
10 }
```

Figura 8: Uso de *CALL* en la implementación secuencial

ra 7. Si consideramos que el grupo $G0$ tiene nP_0 procesadores y el grupo $G1$, nP_1 procesadores, entonces, el procesador con identificador i dentro del grupo $G0$ enviará su subvector ordenado a todos los procesadores j dentro del grupo $G1$ tales que $j = m * nP_0 + i$, con $m = 0, 1, 2, ...$ siempre que $j < nP_1$, y recibirá el otro subvector ordenado del procesador $k = i\,\%nP_1$ del grupo $G1$.

Por último, en la *Fase de Finalización*, se han de reunificar los subconjuntos de procesadores en el conjunto de partida y se han de liberar los recursos solicitados durante las fases anteriores, como son la memoria asignada al vector F y el comunicador del subgrupo (para ello utilizar la función $MPI\_Comm\_free()$). Además, se ha de actualizar el identificador del procesador y el número de procesadores.

El esquema de implementación propuesto para esta paralelización del *quicksort* se ha obtenido del código generado por el compilador `llCoMP` [1]. Dicho compilador permite obtener un código paralelo *MPI* a partir de un código secuencial que ha sido previamente anotado usando `#pragmas` de *OpenMP* y/o de *llc*.

4. Laboratorio 2: Análisis del rendimiento

El siguiente paso al de la implementación de los algoritmos paralelos es el de analizar su rendimiento frente a los algoritmos secuenciales. Esta tarea suele ser tediosa puesto que hay que realizar los experimentos y a continuación mediante una hoja de cálculo representar los resultados obtenidos. En esta práctica se introduce el uso de la herramienta *CALL*. La misma proporciona un traductor (*CALL*), una librería en tiempo de ejecución (`cll.h`) y un analizador de resultados (*LLAC*). Se puede utilizar tanto para el análisis de programas secuenciales, como para programas paralelos desarrollados con *OpenMP* o *MPI*. La principal ventaja es la facilidad de instrumentación del código a analizar, tan sólo hay que expresar la intención de hacerlo mediante el uso específico de los #pragmas de *CALL*.

La Figura 8 muestra un ejemplo de instrumentación del código secuencial del *quicksort*. En este caso se sabe que la complejidad del algoritmo es $n\log(n)$. En la línea 3 se le indica a a *CALL* que se está interesado en medir los tiempos de ejecución para comprobar si se ajustan o no a la fórmula teórica establecida. La cadena cll que aparece después de de la palabra #pragma le indica a *CALL* que se trata de una sentencia que ha de analizar mientras que otros compiladores la consideran un comentario. A continuación se especifica el nombre del experimento. En este caso el identificador elegido es qs. A partir de ese momento viene la especificación en *CALL* de la fórmula de complejidad. En este caso la fórmula de complejidad describe el comportamiento del tiempo de ejecución en términos del tamaño de los datos de entrada size. Asociada a la fórmula están las constantes qs[0] y qs[1]. La sintaxis de *CALL* requiere que se utilice en la definición de las constantes el mismo nombre dado al experimento. El #pragma cll end

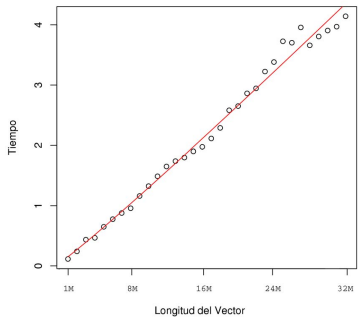

Figura 9: Ajuste de la curva secuencial

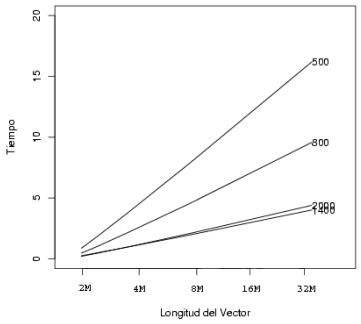

Figura 10: Tiempos secuenciales

de la línea 7 indica que ahí acaba el experimento. Los ficheros de resultados asociados al experimento que se quiere realizar se obtienen añadiendo al final del programa la instrucción #pragma cll report all. De forma alternativa, se puede mencionar una lista de experimentos en vez de todos los experiementos (all).

Una vez instrumentado el código fuente con la descripción del experimento a realizar se ha de compilar con *CALL* con la orden:

```
call nombreFichero.c
```

Como resultado, se obtienen dos nuevos ficheros con extensiones .cll.c y .cll.h. El siguiente paso consiste en ejecutar el compilador de C sobre el fichero generado con la orden:

```
gcc nombreFichero.cll.c -o nombreFichero
```

Al ejecutar nombreFichero se generará un fichero de salida con los resultados obtenidos llamado nombreFichero.c.dat. Con cada ejecución se genera un fichero de resultados diferente con el nombre indicado y acabado en el número de ejecución, esto es, nombreFichero.c.dat.1, nombreFichero.c.dat.2,...

LLAC es una herramienta de análisis estadístico basada en R [5]. Haciendo uso de ella se puede comprobar la validez de la fórmula de complejidad introducida en una sentencia #pragma de la instrumentación. Además, también se puede usar para predecir el comportamiento del sistema de modelado para diferentes valores de las variables introducidas en la fórmula. La Figura 9 muestra el ajuste de los tiempos de ejecución del *quicksort* secuencial a la fórmula de complejidad. Los experimentos se realizaron para 32 tamaños de array diferentes desde 1Kb hasta 32Mb en un $Intel^R$ $Xeon^{TM}$ CPU 1.40GHz.

También se pueden realizar comparaciones entre el comportamiento de los algoritmos en arquitecturas diferentes. En la Figura 10 se representan los tiempos de ejecución secuenciales del *quicksort* obtenidos con cinco tamaños diferentes del array a ordenar (2Mb, 4Mb, 8Mb, 16Mb y 32Mb) en cuatro tipos de máquinas. Se ha denotado por 500 a un AMD-K6(tm) 3D a 500Mhz, por 800 a un AMD $Duron^{TM}$ a 800Mhzx), por 1400 a un $Intel^R$ $Xeon^{TM}$ CPU 1.40GHz y por 2000 a un AMD $Athlon^{TM}$ XP 2000+ 1663Mhz. Nótese que no sólo influye la velocidad de las máquinas sino también su arquitectura como se puede apreciar entre los equipos $Xeon^{TM}$ y $Athlon^{TM}$.

Finalmente, indicar que las herramientas *CALL* y *LLAC* permiten también la instrumentación de programas paralelos. Por ejemplo, en la Figura 11 se muestran los tiempos de ejecución con uno, dos y cuatro procesadores del *quicksort* implementado con *OpenMP* para tamaños del array entre 2Mb y 32Mb. Se plantea como ejercicio la instrumentación *CA-*

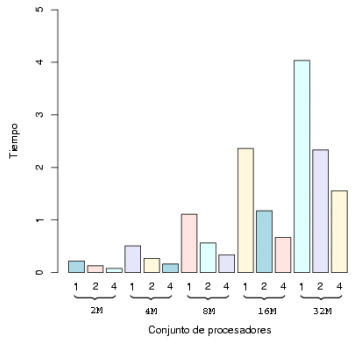

Figura 11: Tiempos para *OpenMP*

LL de los códigos paralelos desarrollados en el laboratorio anterior y el análisis de los resultados con *LLAC*.

5. Conclusiones

En este trabajo se ha presentado una propuesta de laboratorios prácticos para el estudio de la paralelización de la técnica Divide y Vencerás. El nivel de dificultad de los mismo está pensado para una asignatura que se imparta en los últimos años de estudio. Se toma como caso de estudio un algoritmo secuencial ampliamente conocido por el alumnado y a partir de él se abordan las distintas posibilidades de paralelización. También como trabajo práctico se plantea el análisis del rendimiento de las propuestas paralelas frente a las secuenciales utilizando las herramientas *CALL* y *LLAC*.

La propuesta de ejercicios prácticos es novedosa porque plantea un problema abierto como es la implementación eficiente de *paralelismo anidado*. La propuesta aquí realizada se basa en el modelo de Computación Colectiva implementado por la herramienta `llCoMP`.

En la actualidad se está trabajando en la elaboración de otros ejercicios de laboratorio basados en la herramienta *MaLLBa :: DC* [3]. Esta herramienta proporciona un esqueleto genérico para la técnica Divide y Vencerás con implementaciones paralelas completamente distribuidas. Tanto esta herramienta como `llCoMP` han sido desarrolladas por los autores como Proyectos Fin de Carrera.

El tiempo medio para desarrollar los laboratorios propuestos en este artículo es de 15 horas. El alumnado recibe junto con el enunciado del ejercicio una clase en la que se le detallan los objetivos específicos. Además, se le proporcionan esqueletos del código a desarrollar. El grado de aceptación de los alumnos de esta asignatura optativa es alto, puesto que muchos de ellos continúan interesados en el tema y desarrollan el Proyecto Fin de Carrera en trabajos relacionados.

Referencias

[1] Dorta, A. *llCoMP: Paralalelismo Anidado en OpenMP. PFC.* Universidad de La Laguna, Tenerife, 2002. Disponible en `http://nereida.deioc.ull.es/~llCoMP/`.

[2] Dorta I., León C., Rogríguez C., Rodríguez G., Rojas A., *Complejidad Algorítmica: de la Teoría a la Práctica*, Actas JENUI'2003, 523-530, 2003.

[3] González, J.R. *Esqueletos Paralelos Distribuidos. PFC.* Universidad de La Laguna, Tenerife, 2003.

[4] Hoare, C. A. R., *Algorithm 64: Quicksort*, Communications of the ACM, 4, 321, 1961.

[5] Ihaka R., Gentleman R., *R, A Language for Data Analysis and Graphics*, Journal of Computational and Graphical Statistics 5 (3), 299-314, 1996.

[6] Message Passing Interface Forum, *MPI: A message-passing interface standard*, International Journal of Supercomputing, Applications and High Performance Computing, 8, (3/4), 1994.

[7] OpenMP Architecture Review Board, *OpenMP C and C++ Application Program Interface. Version 1.0*, 1998. Disponible en `http://www.openmp.org` .

[8] Wirth N. *Algoritmos + Estructuras de Datos = Programas*, Ediciones el Castillo, 1980.