Paralelización de la Biblioteca Científica de GNU para Sistemas Heterogéneos

J. Aliaga[†], F. Almeida[‡], J.M. Badía[†], S. Barrachina[†], V. Blanco[‡], M. Castillo[†], U. Dorta[‡], R. Mayo[†], E.S. Quintana[†], G. Quintana[†], C. Rodríguez[‡], F. de Sande[‡]

Resumen— **Presentamos en este artículo el trabajo conjunto que estamos realizado para desarrollar una versión paralela de la Biblioteca Científica GNU para sistemas heterogéneos. Describiremos tanto el diseño como la implementación de la biblioteca haciendo uso de dos operaciones ampliamente utilizadas en matemáticas discretas y en álgebra lineal dispersa. Utilizaremos también dichas operaciones para presentar resultados experimentales obtenidos con *clusters* de computadores personales.**

Palabras clave— **Biblioteca Científica GNU, computación científica, algoritmos y arquitecturas paralelas, sistemas paralelos heterogéneos.**

I. Introducción

La Biblioteca Científica GNU (*GNU Scientific Library*, GSL [**?**]) está formada por cientos de rutinas para la realización de cálculo numérico que abarcan aritmética de números complejos, matrices y vectores, álgebra lineal, estadística y optimización, etc.

Actualmente no existe una versión paralela de GSL, probablemente debido a que cuando se inició el proyecto no existía un estándar universalmente aceptado para el desarrollo de aplicaciones paralelas. Sin embargo, creemos que con la introducción de MPI la situación ha cambiado sustancialmente. MPI está actualmente aceptado como la interfaz estándar para el desarrollo de aplicaciones paralelas que utilicen el modelo de paso de mensajes.

En el área de la computación paralela y distribuida estamos interesados en desarrollar una versión paralela integrada de la GSL que pueda ser utilizada como un entorno para la resolución de problemas relacionados con la computación numérica científica. En particular, pretendemos que dicha biblioteca sea portable a varias arquitecturas paralelas, incluyendo procesadores con memoria compartida y distribuida, sistemas híbridos (consistentes en una combinación de ambos tipos de arquitecturas) y *clusters* de nodos heterogéneos. Creemos que nuestra aproximación es diferente a las de otras bibliotecas científicas paralelas existentes (ver, p.e., *http://www.netlib.org*) en el hecho de que nuestra biblioteca tiene como objetivo múltiples clases de arquitecturas.

Describimos en este artículo el diseño y la implementación de aquella parte de nuestra biblioteca relacionada con los sistemas de memoria distribuida heterogéneos. Para simplificar, en la versión actual de la biblioteca tan sólo hemos considerado la heterogeneidad en el rendimiento de los procesadores, ignorando las diferencias en la velocidad de acceso a memoria o en el ancho de banda de la red de interconexión. Una simplificación adicional es que hemos ignorado la diferente naturaleza de las arquitecturas del sistema, lo que podría producir resultados erróneos o provocar *dead locks*. Aunque existen soluciones razonablemente directas para algunos de estos problemas, su implementación conlleva una importante sobrecarga en el proceso de comunicación [**?**].

Para ilustrar nuestra aproximación hemos utilizado dos operaciones sencillas y clásicas que aparecen en matemáticas discretas y en álgebra lineal dispersa. Aún así, consideramos que los resultados presentados en este artículo son extensibles a un amplio rango de las rutinas de GSL. Actualmente estamos centrados en la definición de la arquitectura, en la especificación de las interfaces y en el diseño e implementación paralela de una cierta parte de la GSL.

El resto del artículo está estructurado como sigue. En la Sección II describimos la arquitectura software de nuestra biblioteca paralela. En la Sección III presentamos nuestra visión del sistema paralelo y de la interfaz del modelo de programación. Más adelante, en las Secciones IV y V, utilizamos dos ejemplos para ilustrar las soluciones que hemos adoptado. Finalmente, presentamos resultados experimentales en la Sección VI y algunas conclusiones en la Sección VII.

II. Arquitectura software

Hemos diseñado nuestra biblioteca utilizando una arquitectura software multinivel (ver Figura 1): cada capa ofrece ciertos servicios a las capas superiores a la vez que oculta cómo son implementados.

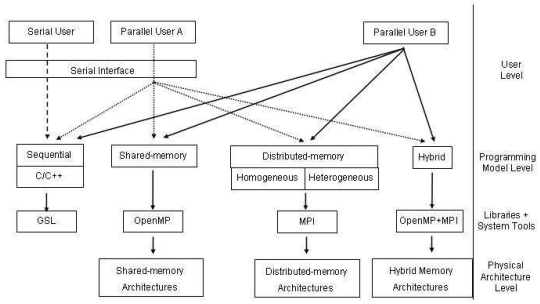

Fig. 1. Arquitectura software de la biblioteca paralela integrada para el cálculo numérico científico.

[†]Depto. de Ingeniería y Ciencia de Computadores, Universidad Jaume I, 12.071–Castellón, Spain; {*aliaga,badia,barrachi,castillo,mayo,quintana,gquintan*}*@icc.uji.es.*

[‡]Depto. de Estadística, Investigación Operativa y Computación, Universidad de La Laguna, 38.271–La Laguna, Spain; {*falmeida,vblanco,casiano,fsande*}*@ull.es.*

El *nivel de usuario*, el nivel superior, proporciona una interfaz secuencial que oculta los detalles de la programación paralela a aquellos usuarios sin experiencia en este tipo de programación, proporcionando servicios a través de funciones C/C++ que concuerdan con los prototipos especificados por la interfaz secuencial de la GSL.

El *nivel de modelo de programación* proporciona instancias de la biblioteca GSL para varios modelos de computación, incluyendo el modelo de memoria distribuida. Este nivel implementa los servicios ofrecidos al nivel superior utilizando bibliotecas estándares y herramientas de paralelización como GSL (versión secuencial), MPI y OpenMP.

En el *nivel de arquitectura física* el diseño incluye plataformas con memoria compartida (p.e. SGI Origin 3000), arquitecturas de memoria distribuida (tales como *clusters* de PCs heterogéneos) y sistemas híbridos (*clusters* de nodos con memoria compartida). Claramente, el rendimiento de las rutinas paralelas dependerá de la adecuación entre el paradigma de programación elegido y la arquitectura

III. Modelo de programación con memoria distribuida

A. Visión del sistema paralelo

En nuestro sistema paralelo utilizamos un modelo SPMD sobre p procesos, P_0, P_1,...,P_{p-1}, [**?**]. Preferimos esta organización a la de maestro-esclavo debido a que es más sencilla para ciertas aplicaciones [**?**], [**?**].

Cada nodo ejecuta un proceso con una carga proporcional a su rendimiento. Se asume que el usuario proporciona un fichero con su evaluación del rendimiento de cada nodo. Un ejemplo de fichero de configuración simplificado con cuatro nodos sería el siguiente:

```
1: ucmp0:  2.0
2: ucmp12: 2.0
3: sun:    3.5
4: linex:  1.0
```

Los rendimientos han sido normalizados con respecto al nodo más lento, `linex`, a quien siempre se le asigna un valor 1.0. Una versión más elaborada de nuestra biblioteca puede incluir la opción de generar de forma automática estos valores por medio de la ejecución de una serie de pruebas (*benchmarks*) [**?**], o a partir de una versión secuencial de los códigos paralelos con un número de datos más reducido [**?**].

Nuestro código paralelo también es apropiado para sistemas paralelos homogéneos, ya que pueden ser considerados como un caso particular de los heterogéneos. Sin embargo, en dicho caso, el código sufre una pequeña merma en el rendimiento debido al especial procesamiento de determinadas estructuras de datos que es necesario realizar si se quiere tener en cuenta la heterogeneidad.

B. Interfaz del modelo de programación

Todos los modelos de programación presentan una interfaz similar en este nivel. La Tabla I relaciona los nombres de varias rutinas secuenciales a nivel de usuario con las rutinas correspondientes en el modelo de memoria distribuida. Las letras "`dm`" especifican el modelo, mientras las letras "`dd`", indican que los datos están distribuidos. Otras instancias de la biblioteca incluyen rutinas para memoria compartida y sistemas híbridos.

TABLA I
Relación entre las rutinas del nivel de usuario y las correspondientes en su versión paralela.

Nivel de usuario (Secuencial)	Nivel de modelo de programación (Memoria distribuida)
`fscanf()`	`gsl_dmdd_fscanf()`
`gsl_sort_vector()`	`gsl_dmdd_sort_vector()`
`gsl_usmv()`	`gsl_dmdd_usmv()`

Valga el siguiente programa, que ordena un vector, para mostrar la interfaz del modelo de programación de memoria distribuida:

```
1: #include <mpi.h>
2: #include <gsl_dmdd_sort_vector.h>
3: void main (int argc, char * argv []) {
4:    int n = 100, status;
5:    gsl_dmdd_vector * v;
      ...
6:    MPI_Init (& argc, & argv);
7:    gsl_dmdd_set_context (MPI_COMM_WORLD);
8:    v = gsl_dmdd_vector_alloc (n, n);      // Reserva
9:    gsl_dmdd_vector_scanf (v, n);          // Entrada
10:   status = gsl_dmdd_sort_vector (v);     // Ordenación
11:   printf ("Test sorting: %d\n", status); // Salida
12:   gsl_dmdd_vector_free (v);              // Liberación
13:   MPI_Finalize ();
14:   exit(0);
15: }
```

En este caso, es el usuario el que debe encargarse de inicializar y terminar la máquina paralela, con la invocación a las rutinas `MPI_Init()` y `MPI_Finalize()`, respectivamente. Además, puesto que el núcleo de la GSL requiere cierta información sobre el contexto paralelo, el usuario debe invocar la función `gsl_dmdd_set_context()` para transferir dicha información desde el programa MPI al núcleo y así crear el contexto GSL adecuado. Por lo tanto, dicha rutina debe ser llamada antes que cualquier otra rutina GSL paralela. Esta función sirve además para que el nodo tome conciencia de su rendimiento. El programa MPI anterior asume que el vector está distribuido entre todos los procesos, de tal forma que, cuando se invoca la función `gsl_dmdd_vector_alloc()`, la asignación de los elementos sigue una determinada política de distribución. En caso necesario, el usuario puede acceder a elementos individuales por medio de los métodos *Setters* y *Getters* de la estructura. La llamada a la función `gsl_dmdd_sort_vector()` ordena el vector distribuido.

IV. Estudio de casos para vectores: Ordenación

Una de las primeras decisiones que debe tomarse al paralelizar programas que involucran vectores, es la de cómo distribuir el trabajo computacional entre

los procesos de tal forma que se maximice el grado de paralelismo, a la vez que se minimizan los costes provocados, por ejemplo, por la comunicación y los tiempos de inactividad.

A. Tratando con vectores

Nuestra propuesta consiste en realizar una distribución cíclica por bloques de tamaño variable del vector. El tamaño de cada bloque dependerá del rendimiento del nodo al que se asigne.

La siguiente es la estructura de datos utilizada para manejar vectores distribuidos heterogéneos:

```
1: typedef struct {
2:   size_t global_size;        // Global size
3:   gsl_vector * local_vector; // Local data
4:   size_t cycle_size;         // Cycle size
5:   size_t global_stride;      // Global stride
6:   size_t * local_sizes;      // Elements per process
7:   size_t * block_sizes;      // Block sizes per process
8:   size_t * offset;           // Offsets per process
9: } gsl_dmdd_vector;
```

Aquí, `local_vector`, de tipo GSL `gsl_vector`, se utiliza para almacenar la información local. Vectores adicionales replican en cada proceso la información requerida por los métodos *Setters* y *Getters* para transformar direcciones globales a locales.

Para crear un vector distribuido, la rutina `gsl_dmdd_vector_alloc()` recibe un parámetro, `cycle_size`, que determina la política de distribución. La Figura 2 muestra los contenidos de esta estructura cuando se utiliza para almacenar un vector de 16 elementos distribuido en 4 procesos. En el ejemplo se asume que los procesos P_0 y P_1 se ejecutan en nodos tres veces más rápidos que aquellos en los que han sido asignados los otros dos procesos. Además, los tamaños de bloque son 3 y 1 para los nodos rápidos y lentos, respectivamente.

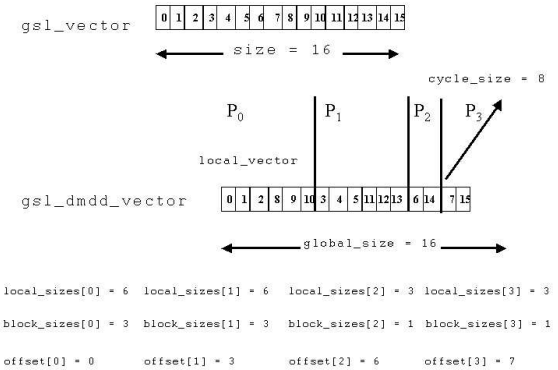

Fig. 2. Una distribución de datos de bloques cíclicos de un vector distribuido con 16 elementos y 2 nodos rápidos y 2 lentos.

B. Ordenación de vectores en sistemas heterogéneos

Hemos escogido el conocido algoritmo Ordenación Paralela por Muestreo Regular (*Parallel Sort by Regular Sampling*, PSRS) introducido en [**?**]. Este algoritmo ha sido concebido para arquitecturas de memoria distribuida con nodos homogéneos y presenta unas buenas propiedades de equilibrado de carga, unos requisitos modestos de comunicación y una localidad de las referencias en los accesos a memoria razonable. Hemos adaptado dicho algoritmo a un entorno heterogéneo utilizando la estructura `gsl_dmdd_vector` y una distribución por bloques pura (sin ciclos).

El algoritmo PSRS heterogéneo propuesto está compuesto de los siguientes cinco pasos:

1. Cada proceso ordena sus datos locales, escoge $p-1$ muestras equidistantes y las envía a P_0. El paso utilizado para la selección de las muestras es, en este contexto heterogéneo, diferente para cada proceso y se calcula en función del tamaño del vector local que debe ordenarse.
2. El proceso P_0 ordena las muestras recogidas, encuentra $p-1$ "pivotes" y los transmite a los restantes procesos. Los pivotes son seleccionados de tal forma que la unión que se realizará en el paso 4 dé lugar a tamaños de vectores de datos locales adecuados al rendimiento computacional de los nodos.
3. Cada proceso parte sus datos y envía su i-ésima partición al proceso P_i.
4. Cada proceso mezcla las particiones recibidas.
5. Todos los procesos participan en la redistribución de los resultados de acuerdo con la plantilla de datos especificada para el vector de resultados.

Una redistribución de los datos es necesaria incluso para un vector distribuido en bloques, ya que los tamaños resultantes de los trozos después del paso 4 no tienen porqué concordar con una distribución apropiada en bloques. El tiempo gastado en esta última redistribución es proporcional al desequilibrio global y es dependiente de los datos del problema.

C. Matrices: ¿una generalización de los vectores?

A pesar de que las matrices pueden ser consideradas inicialmente como una generalización de los vectores, su uso en programas paralelos es más complejo [**?**], [**?**].

En particular, si en un sistema heterogéneo los procesos están organizados siguiendo una topología bidimensional (malla), se ha demostrado que el problema de asignar los procesos a la malla y la subsiguiente distribución de los datos es NP-completo [**?**]. Sin embargo, puesto que los elementos de matrices y vectores se combinan en ciertas operaciones como el producto matriz-vector o en la solución de un sistema lineal, el rendimiento paralelo de estas operaciones podría beneficiarse en gran medida utilizando distribuciones "compatibles" para matrices y vectores.

Las distribuciones bidimensionales cíclicas por bloques para matrices (densas) han sido propuestas y utilizadas en bibliotecas tales como ScaLAPACK [**?**] para sistemas paralelos homogéneos. Más recientemente, la distribución de datos de ScaLAPACK ha sido extendida para acomodar a los sistemas heterogéneos [**?**].

En este punto, aún no hemos decido cómo tratar

a las matrices densas en nuestra biblioteca paralela para sistemas heterogéneos. Sin embargo, ya hemos incorporado a dicha biblioteca un caso especial de estructuras de datos bidimensionales, las matrices dispersas, que describimos a continuación.

V. Un problema especial: matrices dispersas

Las matrices dispersas ocurren en un gran número de campos, algunos tan diferentes como el análisis de estructuras, el reconocimiento de formas, el control de procesos o la tomografía. Sorprendentemente, GSL no incluye rutinas para la computación de álgebra lineal dispersa. Esto sólo puede ser explicado por la falta de estándares en este área: sólo recientemente el *BLAS Technical Forum* propuso un estándar [**?**] para el diseño de interfaces de los subprogramas de álgebra lineal básica (BLAS) para matrices dispersas no estructuradas.

A pesar de que se han desarrollado en los últimos años varios paquetes paralelos para álgebra lineal dispersa (`www.netlib.org/utk/people/JackDongarra/la-sw.html`), todos ellos están destinados a sistemas paralelos homogéneos. Por lo tanto, creemos que nuestro trabajo es único, dado que sigue la especificación estándar para la interfaz BLAS dispersa y está orientado a sistemas paralelos heterogéneos.

A. Trabajando con matrices dispersas

La implementación, paralelización y rendimiento de las computaciones paralelas dispersas depende en gran medida del esquema empleado para el almacenamiento de la matriz dispersa, que suele depender de los datos que la aplicación proporciona.

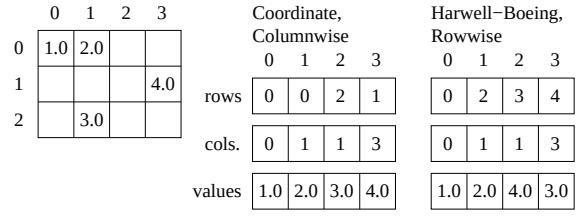

Fig. 3. Esquema de almacenamiento de una matriz dispersa 3×4 con $nz = 4$ elementos no nulos.

Dos de los esquemas de almacenamiento más ampliamente usados para matrices dispersas son los formatos de *coordenadas* (*coordinate*) y de *Harwell-Boeing* [**?**]. En el formato de coordenadas se utilizan tres vectores de tamaño nz que almacenan los valores no nulos de la matriz y los índices de sus filas y columnas. Estos valores pueden ordenarse por filas (*rowwise*) o por columnas (*columnwise*). La variante por filas del formato Harwell-Boeing utiliza un par de vectores de longitud nz para los índices de las columnas y los valores. Un tercer vector, de tamaño igual al número de filas de la matriz más 1, determina los índices de comienzo/final de cada una de las filas. La variante por columnas de este formato intercambia las funciones de los vectores fila y columna. La Figura 3 ilustra el uso de los dos formatos de almacenamiento comentados por medio de un sencillo ejemplo.

Parece que ninguna de las variantes arriba mencionadas posea una ventaja definitiva sobre las demás; en nuestros programas hemos optado por la variante por filas del formato de coordenadas.

Nuestro método distribuye la matriz en p bloques de filas de aproximadamente el mismo tamaño, uno por proceso.

La siguiente estructura de datos (simplificada) se utiliza para manejar matrices dispersas en un sistema distribuido heterogéneo:

```
1: typedef struct {
4:   size_t global_size1;   // Número de filas
5:   size_t global_size2;   // Número de columnas
6:   size_t global_nz;      // Número de no nulos
7:   void   * local_matrix; // Datos locales
5:   size_t * owns;         // Propietario
6:   size_t * permutation;  // Permutación de filas
8: } internal_dmdd_sparse_matrix;
```

En esta estructura, `local_matrix` almacena los datos locales de un proceso en formato de coordenadas por filas. El "vector" `permutation` permite definir una permutación de las filas de la matriz (por motivos de rendimiento). El vector `owns` especifica los índices de comienzo del bloque de filas de cada proceso.

B. Paralelización del producto matriz dispersa vector

Describimos a continuación la implementación paralela del producto matriz dispersa vector:

$$y \leftarrow y + \alpha \cdot A \cdot x, \tag{1}$$

donde un vector (denso) y, de tamaño m, es actualizado con el producto de una matriz dispersa A, de tamaño $m \times n$, por un vector (denso) x, de n elementos, escalado por un valor α. Es conveniente destacar que ésta es de lejos la operación que más frecuentemente aparece en el álgebra lineal dispersa [**?**], puesto que se preserva la dispersión de A permitiendo aprovechar la existencia de ceros en la matriz para reducir su coste computacional. Para simplificar, ignoraremos de aquí en adelante el caso más general, donde A puede reemplazarse en (1) por su transpuesta o por su transpuesta conjugada.

El producto matriz (dispersa) vector es habitualmente implementado mediante una secuencia de operaciones *saxpy* o productos escalares, prefiriendo una u otra dependiendo de la arquitectura destino y de cómo se almacenen los datos.

Teniendo en cuenta que utilizamos el formato de almacenamiento de coordenadas por filas, hemos optado por implementar la operación mediante una secuencia de productos escalares. Asumiremos que los dos vectores involucrados, x e y, están inicialmente replicados en todos los procesos. También consideraremos una partición en bloques del vector $y = (y_0, y_1, \ldots, y_{p-1})$, con aproximadamente el mismo número de elementos por bloque, y una partición de la matriz dispersa A en bloques de aproximadamente m/p filas como $A = \left(A_0^T, A_1^T, \ldots, A_{p-1}^T\right)^T$. El algo-

ritmo propuesto está compuesto por los siguientes dos pasos:

1. Cada proceso P_i calcula su parte correspondiente del producto, $z_i = A_i \cdot x$; a continuación escala y acumula el resultado como $z_i = y_i + \alpha \cdot z_i$.
2. Cada proceso recoge los resultados locales calculados por los restantes procesos y crea una copia replicada de y.

Es conveniente destacar que, debido a la replicación de x, el primer paso no requiere de ninguna comunicación. El segundo paso requiere una comunicación colectiva (de tipo `MPI_Allgather`) para replicar los resultados. Esta operación está (casi) perfectamente equilibrada desde el punto de vista de la comunicación puesto que todos los procesos tienen un número similar de elementos de y. Sin embargo, dependiendo del patrón de dispersión de los datos, puede que la carga computacional no esté en absoluto equilibrada.

C. *Equilibrando la carga computacional en el producto matriz dispersa vector*

Para equilibrar la carga computacional proponemos que la matriz sea almacenada con sus filas permutadas, de tal forma que el número de entradas distintas de cero en todos los bloques de filas, A_0, A_1,...,A_{p-1}, sea aproximadamente proporcional al rendimiento del nodo haya sido asignado. Esta permutación de filas es calculada una sola vez, y su coste se amortiza si se realizan varios productos con la misma matriz, como ocurre p.e., en la solución de sistemas lineales mediante métodos iterativos [**?**].

El siguiente heurístico nos permite obtener de forma rápida una buena permutación. En primer lugar se asigna la fila con el mayor número de elementos no nulos al proceso más rápido. Luego, para cada uno de los procesos restantes, se busca una fila (no asignada) con un número de elementos no nulos que haga que el número total de elementos no nulos asignados a cada proceso sea proporcional a su rendimiento. El procedimiento se repite, empezando con el proceso más rápido, hasta que no queden filas sin asignar.

Los resultados de este heurístico pueden ser fácilmente refinados por un procedimiento iterativo que compare las filas asignadas a un par de procesos y que las intercambie en el caso de conseguir con ello un mejor equilibrio de la carga.

VI. Resultados experimentales

En esta sección mostramos los resultados experimentales en dos *clusters* heterogéneos formados por 14 y 6 nodos (ver Tabla II), conectados en ambos casos por medio de un *switch Fast Ethernet* (100 Mbits/sec.). El *Cluster* 1 se ha utilizado para la operación de ordenación, mientras que el *Cluster* 2 ha sido el banco de pruebas utilizado para la operación producto matriz dispersa por vector. Se han añadido nodos a cada experimento comenzando con aquellos clasificados como tipo 0, para continuar con los nodos tipo 1 (hasta que no quedan nodos de este tipo) y finalizando con tantos nodos tipo 2 como hayan sido necesarios. Se compara el rendimiento de una distribución de datos heterogénea frente a una distribución homogénea (esto es, una que no tiene en cuenta los diferentes rendimientos de cada nodo).

TABLA II
Plataformas heterogéneas utilizadas en la evaluación experimental.

Nombre	Tipo de nodo	Arquitectura	Frec. del procesador	#Nodos : #Proc./nodo
Cluster 1	0	Pentium Xeon	1.4 GHz	1 : 4
	1	AMD (Duron)	800 MHz	4 : 1
	2	AMD-K6	500 MHz	6 : 1
Cluster 2	0	Pentium III	550 MHz	1 : 1
	1	AMD (Athlon)	900 MHz	4 : 1
	2	Pentium Xeon	700 MHz	1 : 4

Para la operación de ordenación hemos generado vectores de enteros con entradas distribuidas de forma aleatoria. Las Figuras 4 y 5 contienen la aceleración obtenida con la distribución de datos heterogénea y homogénea. El esquema de datos homogéneo produce el habitual descenso en el rendimiento y picos en la aceleración como consecuencia de la introducción de procesadores más lentos. Por otro lado, la utilización de una distribución de datos heterogénea da como resultado una curva de aceleración más suave y mejorada (ver Figura 4). La comparación entre los tiempos de ejecución entre estos dos esquemas se hace de forma explicita en la Figura 6.

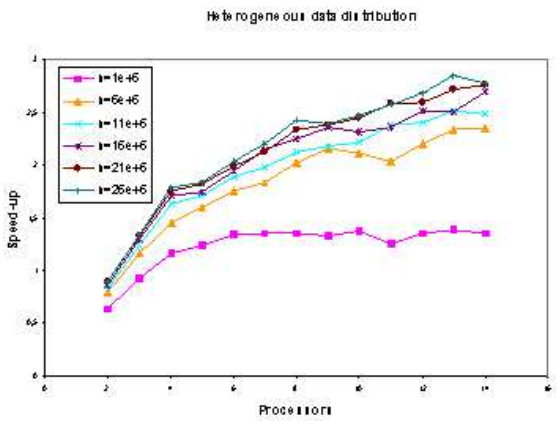

Fig. 4. Aceleración de la operación de ordenado con una distribución de datos heterogénea en el *cluster* 1.

A continuación realizamos experimentos similares para el producto matriz dispersa vector. En la evaluación hemos utilizado dos matrices dispersas de gran tamaño procedentes de la colección *Matrix Market* (`math.nist.gov/MatrixMarket`), conocidas como e40r000 y fidap011.

Las aceleraciones del producto matriz dispersa vector con datos distribuidos siguiendo los esquemas homogéneo y heterogéneo se muestran en la Figura 7. Los resultados muestran mejores aceleraciones para la distribución heterogénea, además de una mejor

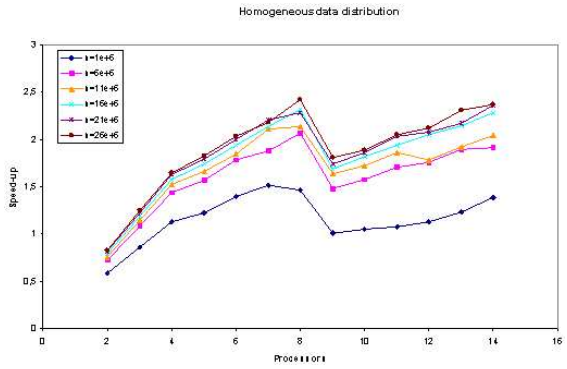

Fig. 5. Aceleración de la operación de ordenado con una distribución de datos homogénea en el *cluster* 1.

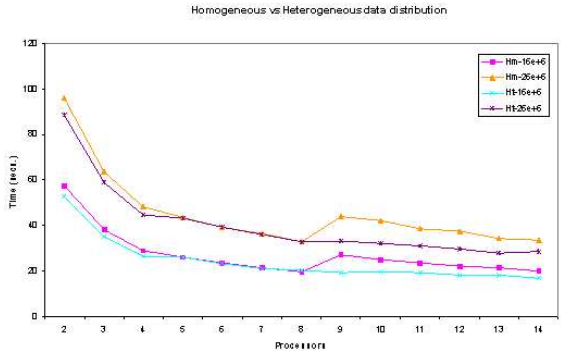

Fig. 6. Tiempos de ejecución de la operación de ordenado en el *cluster* 1.

"escalabilidad" cuando el número de nodos se incrementa hasta 6. Los tiempos de ejecución para ambas distribuciones se muestran en la Figura 8. Los tiempos de ejecución del esquema heterogéneo son consistentemente mejores que los del homogéneo, siendo mayor la distancia entre ambas distribuciones cuanto mayor es la heterogeneidad de los nodos que intervienen en el experimento.

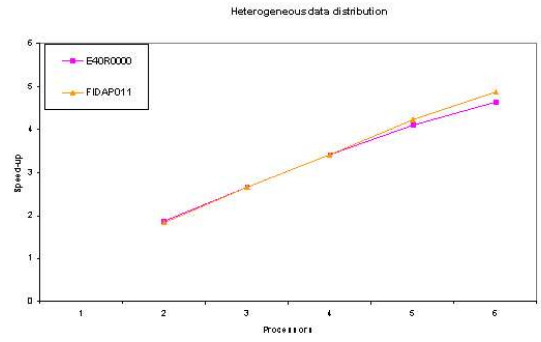

Fig. 7. Aceleración del producto matriz dispersa vector con una distribución de datos heterogénea en el *cluster* 2.

VII. Conclusiones

Hemos presentado el diseño y el desarrollo de una versión paralela de la biblioteca científica GNU orientada a arquitecturas paralelas con formadas por nodos heterogéneos. Se han utilizado dos operaciones sencillas procedentes del álgebra lineal dispersa y de

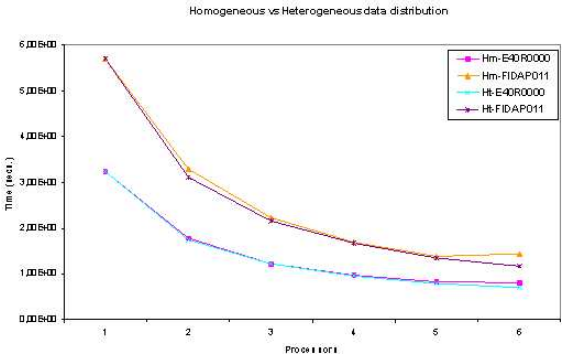

Fig. 8. Tiempos de ejecución del producto matriz dispersa vector en el *cluster* 2.

la matemática discreta para exponer la arquitectura e interfaz del sistema, y a la vez, para obtener resultados experimentales que muestren el rendimiento de dicha librería. Estos ejemplos acreditan que nuestro método puede extenderse a un amplio número de rutinas de la GSL.

Agradecimientos

Este trabajo ha sido financiado por la CE (FEDER) y el MCyT (Plan Nacional de I+D+I, TIC2002-04498-C05-05 y TIC2002-04400-C03).