Equilibrado Dinámico de Carga en Sistemas Heterogéneos Dedicados

I. Galindo F. Almeida V. Blanco J.M. Badía

Dpto. Estadística, .I.O. y Computación

Universidad de La Laguna, España

alu2621@etsii.ull.es, falmeida@ull.es, vblanco@ull.es, badia@icc.uji.es

7 de junio de 2007

Resumen

La computación en paralelo en ambientes heterogeneos está siendo un importante foco de atracción debido a la creciente difusión de este tipo de sistemas. Se convierte en un problema de base la portabilidad a estos sistemas de códigos y librerías ya existentes. El rendimiento de este código se ve afectado por la gran dependencia que existe entre el código y estas arquitecturas paralelas. Hemos desarrollado una librería de equilibrado dinámico que permite, para una amplia clase de problemas, la adaptación de códigos paralelos a sistemas heterogéneos en tiempo de ejecución. La sobrecarga introducida por nuestro sistema es mínima y el coste de uso para el programador es despreciable. La estrategia ha sido validada sobre una amplia batería de problemas, lo que confirma la validez de nuestra propuesta.

1. Introducción

La difusión de arquitecturas heterogéneas es previsible que aumente en los próximos años debido a la creciente tendencia a utilizar múltiples recursos de cómputo (habitualmente heterogéneos) de una institución como un recurso computacional único. El rendimiento de este tipo de sistemas está fuertemente comprometido por la fuerte dependencia que existe entre el código paralelo y la arquitectura. En particular, el proceso de asignación de tareas a procesadores se convierte en un problema que en ocasiones requiere de un elevado esfuerzo por parte del programador.

Hemos desarrollado una librería que permite realizar el equilibrado dinámico de tareas en un programa paralelo sobre un sistema heterogéneo dedicado, adaptándose a las condiciones del sistema durante la ejecución. Con el uso de esta librería, se facilita la labor del programador de adaptar a sistemas heterogéneos, códigos paralelos desarrollados para sistemas homogéneos. Se ha implementado la librería de forma que para usarla no haya que variar ninguna línea de código de un programa ya construido, de modo que la intrusión en el código es mínima. Sólo hay que añadir llamadas a tres nuevas funciones:

- El inicio de la librería.
  - `ULL_MPI_init_library(...)`
- La finalización de la librería.
  - `ULL_MPI_shutdown()`
- La función para el equilibrado.
  - `ULL_MPI_calibrate(...)`

Hemos validado nuestra propuesta sobre tres problemas de test, productos de matrices, resolución de sistemas lineales mediante Jacobi y optimización de recursos mediante algoritmos de programación dinámica. Los resultados computacionales muestran los beneficios que proporciona el uso de nuestra librería de equilibrado produciendo reducciones sustanciales de tiempo en todos los casos. El nivel de eficiencia alcanzado, añadido a la mínima intrusión en el código hacen de esta librería una herramienta útil en el contexto de las plataformas heterogéneas.

El trabajo lo hemos estructurado del siguiente modo: en la sección 2 introducimos algunos de los elementos que han motivado este trabajo y los principales objetivos que perseguimos con nuestro desarrollo. La sección 3 muestra el modo de uso de nuestra librería ilustrando los beneficios obtenidos de la aproximación. En la sección 4 describimos el algoritmo de equilibrado utilizado por la librería y en la sección 5 mostramos la validación realizada sobre los problemas seleccionados. Finalizamos con algunas conclusiones y lineas de trabajo futuras.

2. Motivación y Objetivos

La programación en los sistemas paralelos heterogéneos, es claramente dependiente de la arquitectura y el rendimiento obtenido dependerá fuertemente del grupo de máquinas que interviene en la computación. Este hecho hace que, en la mayoría de los casos las técnicas utilizadas para sistemas paralelos homogéneos deban ser revisadas para su aplicación en sistemas no necesariamente homogéneos.

En particular, nos planteamos el problema del sintonizado de programas paralelos en arquitecturas heterogéneas. Dado un programa desarrollado para un sistema homogéneo, pretendemos obtener una versión del mismo que haga uso de las capacidades heterogéneas del sistema, realizando una asignación de tareas adecuada a la capacidad computacional de cada elemento de proceso. La aproximación más simple al problema consiste en la adaptación manual del código de acuerdo a las características de la arquitectura [1]. Esta aproximación, generalmente implica conocer, al menos, características de la arquitectura de modo que las tareas del programa paralelo se puedan asignar de acuerdo a las capacidades computacinales de cada procesador. Una aproximación más general la obtenemos en el contexto de las estrategias de la auto-optimización basadas en el modelo del tiempo de ejecución [4], [2]. En esta segunda aproximación, un modelo analítico que parametriza a la arquitectura y al algoritmo se instancia para cada cada caso particular de modo que se optimice la ejecución del programa. Esta estrategia es bastante más general que la anterior, pero, ciertamente, más difícil de aplicar puesto que el proceso de modelización no es una tarea sencilla y su instanciación y minimización posterior para cada caso tampoco lo es.

Nos planteamos como objetivo el desarrollo de una estrategia de adaptación dinámica del código sobre sistemas heterogéneos de una forma sencilla y eficiente, intentando ser poco intrusivos con el código del usuario de modo que el programa pueda adaptarse sin conocimientos previos de la arquitectura y sin la necesidad de desarrollar modelos analíticos. Pretendemos aplicar la técnica a una amplia clase de problemas, en particular, a aquellos programas paralelos que pueden expresarse como una serie de iteraciones síncronas. Para ello, hemos desarrollado una librería con la que instrumentar determinadas secciones en el código. La instrumentación requerida es mínima y la sobrecarga introducida a consecuencia de la misma es también despreciable. A partir de esta instrumentación, el programa se adaptará dinámicamente a la arquitectura destino.

El diseño de la librería está orientado a resolver las diferencias de tiempos que se obtienen en la ejecución de código paralel basado en un esquema iterativo como el que aparece en el código de la figura 1. En él se dispone de un bucle que ejecuta N iteraciones, donde para cada iteración se tiene que realizar una operación de cómputo. Cada procesador realizará cálculos de acuerdo al tamaño de la tarea que le haya sido asignada y, después de este cálculo, se realiza una operación de comunicación colectiva en la que todos los procesadores se sincronizan antes de pasar a la siguiente iteración realizando algún tipo de multirecogida o recogida simple de datos.

Si ejecutamos el código de la figura 1, por ejemplo, en un cluster heterogéneo compuesto por 3 procesadores de forma que:

- El procesador 2 es dos veces más rápido que el procesador 1.
- El procesador 2 es cuatro veces más rápido que el procesador 0.

```
// proc = Identificador del procesador
// tamaño del problema en cada nodo
count[proc] = Size_problema / Num_procs

for (i = 0; i < N; i++) {
   // Cada nodo realiza el cálculo de
   // tamaño ''count'' que le corresponde
   Computo (count)

   // cada nodo debe de enviar su trabajo
   // al resto, ya que dependen de él
   Operacion_colectiva ()
}
```

Figura 1: Algoritmo básico en un esquema iterativo

la implementación de una asignación homogénea de tareas en la que se asigna el mismo tamaño de problema a computar para cada nodo, lleva a que el tiempo de ejecución obtenido se corresponda con el tiempo que tarda el procesador más lento, en este caso el procesaodor 0, permaneciendo ociosos durante determinado tiempo al resto de procesadores en cada iteración, como se muestra en la figura 2 para un código sintético con tamaño de problema 1500 en el que se distribuyen subproblemas de tamaño 500.

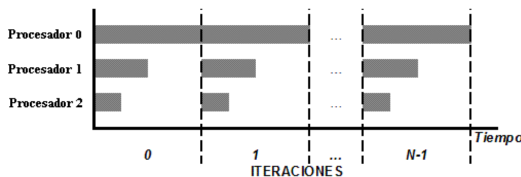

Figura 2: Diagrama de tiempo para una asignación homogénea sobre un sistema heterogéneo.

En la siguiente sección mostramos la estrategia que utilizamos para equilibrar la asignación de tareas, de modo que el tiempo de ejecución en el que interviene cada procesador sea el mismo.

3. Asignación dinámica de tareas

La librería que hemos desarrollado permite el equilibrado dinámico sin más que introducir dos llamadas a la función `ULL_MPI_calibrate()` en la sección del código que se desea equilibrar, tal y como se muestra en el código de la figura 3. Se introduce una llamada al comienzo de la sección a equilibrar y una llamada al final de la misma, de modo que cada procesador puede conocer, en tiempo de ejecución, cuánto tiempo tarda en ejecutar la tarea asignada. De la comparación de este tiempo de ejecución en cada procesador y la consecuente redistribución de tareas, se obtiene el equilibrado de la carga.

Cabe destacar que la comunicación colectiva al final de cada iteración actúa como una especie de barrera que fuerza un alto grado de sincronización de todos los procesos.

```
 // proc = Identificador del procesador
 count[proc] = Size_problem / Num_procs
 displ[proc] = proc * count[proc];
 for (i = 0; i < N; i++) {

   ULL_MPI_calibrate(INIT, i, count, displ,
                     umbral, Size_problem)
   Computo (count)
   ULL_MPI_calibrate(FIN, i, count, displ,
                     umbral, Size_problem)
   Operacion_colectiva ()
}
```

Figura 3: Código instrumentado para el equilibrado dinámico

La función de equilibrado recibe los siguientes argumentos:

- **INIT:** Indica que estamos al inicio de la sección a equilibrar.
- **FIN:** Indica que se ha llegado al final de la sección a equilibrar.
- **count[], displ[]:** Expresan el tamaño de la tarea que debe computar cada procesador, entendida como tamaño de problema más desplazamiento, para que todos los

procesadores inviertan el mismo tiempo de ejecución, .

- **umbral:** Se corresponde con un número de microsegundos que indicarán si se debe de equilibrar o no. La semántica de este parámetro en una iteración es la siguiente:
  - Sea T_i el tiempo que el procesador i tarda en ejecutar **Computo(count)**
  - $Tmax = Maximo(T_i)$
  - $Tmin = Mnimo(T_i)$
  - Si $(Tmax - Tmin) > umbral$ entonces, hay que equilibrar. En caso contrario el sistema ya ha equilibrado la carga de trabajo.
- **i:** Indica con un valor 0 si estamos en la primera iteración o en el resto.
- **Size_problem:** Se corresponde con el tamaño total del problema para ser coherente en los cálculos de los nuevos tamaños de las tareas asociadas a cada procesador **count, displ**.

Realizando una nueva ejecución del código sintético, sobre el cluster anterior de tres procesadores, con un tamaño de problema igual a 1500 y un umbral de 100 microsegundos, obtenemos los siguientes valores para los tamaños de los problemas (count[]) y para los tiempos de ejecución (T_i):

- Iteración i = 0. El algoritmo comienza con una asignación homogénea de tareas:
  - count[proc0] = 500, $T_0 = 400$ us
  - count[proc1] = 500, $T_1 = 200$ us
  - count[proc2] = 500, $T_2 = 100$ us
  - si $((Tmax = 400) - (Tmin100)) > (umbral = 100)$ entonces equilibrar(count[])
- Iteración i = 1. Se realiza una operación de equilibrado de forma automática:
  - count[proc0] = 214, $T_0 = 171$ us
  - count[proc1] = 428, $T_1 = 171$ us
  - count[proc2] = 858, $T_2 = 171$ us

En la figura 4 se muestra el diagrama de tiempos en el que se corrige el desequilibrio de la carga.

Como puede observarse el uso de la librería es simple y la intrusión en el código del usuario es mínima, únicamente será necesario añadir llamadas a las funciones `ULL_MPI_init_calibratelib()`, `ULL_MPI_shutdown_calibratelib()` al comienzo y finalización del código para realizar las tareas de inicialización y liberado de memoria.

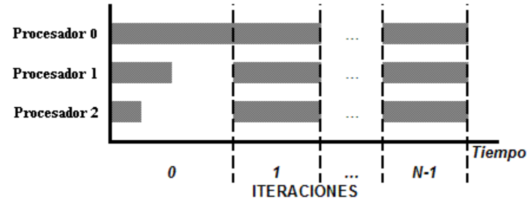

Figura 4: Diagrama de tiempos para un sistema equilibrando de forma dinámica.

4. El algoritmo de equilibrado

La llamada a la función `ULL_MPI_calibrate(...)` debe ser realizada por todos los procesadores y en ella se implementa el algoritmo de equilibrado. Aunque en la literatura podemos encontrar un importante número de algoritmos para realizar el equilibrado [3], hemos optado por una estrategia sencilla y eficiente que nos ha permitido obtener buenos resultados. Sin embargo, la metodología seguida admite una sencilla adaptación de algoritmos de equilibrado que pudieran ser más eficientes. La forma en que cada procesador realiza el equilibrado dinámico es la siguiente: Todos los procesadores involucrados en el equilibrado realizan las mismas operaciones:

- Como entrada al algoritmo se necesitan los tiempos que tarda cada procesador

en realizar el cómputo en cada iteración. Como cada procesador necesita los tiempos del resto de procesadores se realiza el intercambio mediante una operación colectiva.

- $T[]$ = vector donde cada procesador recoge todos los tiempos.
- $SIZE$ = Tamaño del problema.
- $COUNT[]$ = vector de tamaños que computa cada procesador.

- El primer paso es comprobar que no se está superando el umbral

  si (MAX($T[]$) - MIN($T[]$)) $<UMBRAL$ entonces, NO EQUILIBRAR

- Sea TS la suma de tiempos total, es decir, la suma de todos los tiempos de cada procesador: $TS = \sum_{i=0}^{Num_procs-1} T[i]$

- Obtenemos el tiempo teórico (TC) que se ha tardado en realizar una unidad del tamaño del problema: $TC = \frac{TS}{SIZE}$

- Para cada procesador se calcula la relación de tiempos $RT[]$. Se corresponde con la relación que hay entre el tiempo $T[i]$ que se invierte en realizar el cómputo para un tamaño $COUNT[i]$ en relación al tiempo que se tarda por unidad TC en función del tamaño del problema $SIZE$: $RT[i] = \frac{TC*COUNT[i]}{T[i]*SIZE}$,

  $0 \leq i \leq Num_procs - 1$

  $SRT = \sum_{i=0}^{Numprocs-1} RT[i]$

- Finalmente se calcula el tamaño del nuevo $COUNT$ para cada procesador: $COUNT[i] = SIZE * \frac{RT[i]}{SRT}$

De este modo cada procesador ha ajustado el tamaño de la tarea asignada de acuerdo a su capacidad computacional.

5. Resultados Computacionales

Para comprobar las ventajas de nuestra propuesta hemos realizado una amplia experiencia computacional en la que equilibramos varias aplicaciones sobre distintos sistemas heterogéneos. Hemos aplicado la técnica a tres problemas test, el producto de matrices, el método de Jacobi y un algoritmo de programación dinámica para un problema de asignación de recursos. Los experimentos han sido ejecutados sobre tres clusters (cuadro 1) de diferentes tamaños para comprobar cómo se comporta la librería al aumentar el número de procesadores de diferente capacidad de cómputo.

Se han implementado 3 algoritmos para resolver cada uno de los problemas anteriores: secuencial, paralelo y calibrado. En la versión calibrada sólo se ha añadido la función de calibrar donde correspondía, sin alterar el código de la versión paralela. La ejecución de la versión secuencial se ha realizado en el procesador de mayor velocidad en cada cluster. Los tiempos mostrados en los cuadros están expresados en segundos.

Nombre	Procesador	Frecuencia
Cluster 1	0	3.20 GHz
	1	2.66 GHz
	2	1.40 GHz
	3	3.00 GHz
Cluster 2	0, 1	3.20 GHz
	2, 3	2.66 GHz
	4, 5	1.40 GHz
	6, 7	3.00 GHz
Cluster 3	0, 1	3.20 GHz
	2, 3, 4, 5	2.66 GHz
	6, 7	1.40 GHz
	8, 9, 10, 11	3.00 GHz

Cuadro 1: Plataforma heterogénea utilizada para las pruebas, todos los procesadores son del tipo Intel(R) Xeon(TM).

5.1. Multiplicación de Matrices

En el caso de la multiplicación de matrices, asumimos que cada elemento de la matriz producto puede obtenerse aplicando de la siguiente fórmula:

$$C[i,j] = \sum_{k=1}^{n} A[i,k] * B[k,j]$$

La versión secuencial se corresponde con el código de la figura 5. La versión paralela (figura 6) replica la matriz A en todos los procesadores y asocia a cada procesador un número de columnas de la matriz B. En cada iteración

cada procesador calcula parte de la fila C que corresponda, enviando al procesador 0 el resultado computado. El código de la figura 7 se corresponde con la versión equilibrada, donde sólo se han añadido las llamadas a las funciones necesarias para el equilibrio, sin alterar el código paralelo anterior. Obsérvese que se ha utilizado la notación A[I(i,j)] para acceder a los elementos A[i][j] de la matriz.

```
void matrices_sec(double *a, double *b,
                  double *c) {
 int i, j, k;
 for (i = 0; i < N; i++) {
   for (j = 0; j < N; j++) {
     c[I(i,j)] = 0;
     for (k = 0; k < N; k++)
       c[I(i,j)] += a[I(i,k)] * b[I(k,j)];
   }
 }
}
```

Figura 5: Versión secuencial de la multiplicación de matrices.

Se han realizado varias ejecuciones de las tres versiones del algoritmo sobre los tres clusters y se han tomado las siguientes medidas:

- T_{sec}: Tiempo en segundos de la versión secuencial.
- T_{par}: Tiempo en segundos de la versión paralela.
- T_{cal}: Tiempo en segundos de la versión paralela equilibrada.
- $G_R = \frac{T_{par}-T_{cal}}{T_{par}} * 100$: Ganancia relativa.

Las matrices de entrada, en los test utilizados, son matrices cuadradas de dimensión $Size * Size$. Para el algoritmo de equilibrado se ha escogido un umbral de 100 microsegundos. Para evitar innecesarias complicaciones se han elegido tamaños de problemas que resulten múltiplos de los números de procesadores elegidos. Los resultados se muestran en el cuadro 2. Se observa una importante ganancia como consecuencia del equilibrado dinámico que en algún caso supera el 50 %.

```
void matrices_par(double *a, double *b,
                  double *c, int miid) {
 int i, j, k;
 for (i = 0; i < N; i++) {
   // cómputo
   for (j = despl[miid];
        j < (despl[miid]+count[miid]);
        j++) {

      c[I(i,j)] = 0;
      for (k = 0; k < N; k++)
        c[I(i,j)] += a[I(i,k)] * b[I(k,j)];
   }
   // comunicación colectiva
   MPI_Gatherv (&c[I(i,despl[miid])],
     count[miid], MPI_DOUBLE, &c[I(i,0)],
     count, despl, MPI_DOUBLE, 0,
     MPI_COMM_WORLD);
 }
}
```

Figura 6: Versión paralela de la multiplicación de matrices.

Cluster	$Size$	T_{sec}	T_{par}	T_{cal}	G_R
1	1152	30.98	47.21	18.94	59.8
	2304	720.81	400.49	248.46	37.9
	4608	5840.44	3344.19	2035.84	39.1
2	1152	30.98	29.92	15.36	48.6
	2304	720.81	247.98	184.62	25.5
	4608	5840.44	2239.31	1639.96	26.7
3	1152	30.98	20.009	15.42	22.9
	2304	720.81	165.40	134.15	18.8
	4608	5840.44	1487.51	1093.37	26.5

Cuadro 2: Resultados para el problema de multiplicación de matrices.

En el cuadro 3 se muestran los resultados obtenidos por algunas iteraciones, sobre el cluster 1 con un tamaño de 2304. En ella se puede observar que los tiempos, en la iteración 0, de los procesadores 1, 2 son mucho mayores al resto. El calibrado realiza un reparto de trabajo proporcionando mayor carga a los procesadores 0 y 3 y liberando la carga de los procesadores 1 y 2.

5.2. Método de Jacobi

El método de Jacobi es un método iterativo que permite resolver el sistema lineal $Ax = b$.

```
void matrices_cal(double *a, double *b,
                  double *c, int miid) {
 int i, j, k;
 for (i = 0; i < N; i++) {

  ULL_MPI_calibrate(INIT , i,
          &count, &despl, umbral, 1, N);

  for (j = despl[miid];
       j < (despl[miid]+count[miid]);
       j++) {

    c[I(i,j)] = 0;
    for (k = 0; k < N; k++)
      c[I(i,j)] += a[I(i,k)] * b[I(k,j)];
  }

  ULL_MPI_calibrate(FIN , i,
         &count, &despl, umbral, 1, N);

  MPI_Gatherv (&c[I(i,despl[miid])],
               count[miid], MPI_DOUBLE,
               &c[I(i,0)], count, despl,
               MPI_DOUBLE, 0, new_com);
 }
}
```

Figura 7: Versión paralela calibrada de la multiplicación de matrices.

Comienza con una aproximación inicial $x(0)$ a la solución x y genera una sucesión de vectores $x(k)$ que converge a x.

Se han implementado y ejecutado las mismas versiones que el problema anterior tomando las mismas medidas.

Para el algoritmo calibrado se ha colocado como un umbral de 100 microsegundos. Los resultados se muestran en el cuadro 4.

En el Cuadro 5 se muestran los resultados obtenidos por algunas iteraciones, sobre el cluster 1 con un tamaño de 2304. En él se puede observar que el tiempo, en la iteración 0, del procesador 2 es mucho mayor al resto, por lo que se realiza un reparto de trabajo proporcionando mayor carga a los procesador 0, 1 y 3 y liberando la carga del procesador 2.

Iterac.	Resultados
0	Carga: 576 576 576 576 Tiempos: 78792 120623 177158 93472
1	Carga: 783 511 348 662 Tiempos: 106778 106618 103219 106545
2	Carga: 778 508 357 661 Tiempos: 106050 107725 108092 106952
3	Carga: 784 504 353 663 Tiempos: 106813 104668 106701 106665
4	Carga: 780 511 351 662 Tiempos: 106159 107643 108126 106437
5	Carga: 785 507 346 666 Tiempos: 107503 105684 103590 107482

Cuadro 3: Resultado por iteración para el problema de multiplicación de matrices en el cluster 1 con un tamaño de 2304. $Carga$ y $Tiempo$ son los vectores donde cada posición se corresponde con un procesador.

Cluster	$Size$	T_{sec}	T_{par}	T_{cal}	G_R
1	1152	34.76	30.26	14.77	51.18
	2304	138.74	116.44	50.71	56.44
	4608	553.46	463.89	190.74	58.88
2	1152	34.76	17.54	17.05	2.79
	2304	138.74	63.43	53.44	15.74
	4608	553.46	256.80	220.49	14.13
3	1152	34.76	19.79	14.37	27.38
	2304	138.74	54.02	39.38	27.10
	4608	553.46	178.81	162.53	9.10

Cuadro 4: Resultados para el problema del Método de Jacobi.

Iterac.	Resultados
0	Carga: 576 576 576 576 Tiempos: 11590 13757 34056 12446
1	Carga: 739 623 251 691 Tiempos: 14864 14488 14513 14909
2	Carga: 732 633 254 685 Tiempos: 14719 14752 14715 14764
3	Carga: 732 633 254 685 Tiempos: 14740 14755 14706 14765
4	Carga: 732 633 254 685 Tiempos: 14732 14736 14709 14761
5	Carga: 732 633 254 685 Tiempos: 14733 14970 14710 14917

Cuadro 5: Resultado por iteración para el Método de Jacobi en el cluster 1 con un tamaño de 2304. $Carga$ y $Tiempo$ son los vectores donde cada posición se corresponde con un procesador.

5.2.1. Problema de Asignación de Recursos

Se resuelve el problema de asignación de recursos: Se dispone de m recursos de capacidades $c1, c2, , cm$ y n tareas a ejecutar que consumen parte de los recursos. La tarea i-ésima consume w_{ij} partes del recurso j. La ejecución de la tarea i-ésima produce un beneficio b_i. Se trata de decidir qué tareas se ejecutan de manera que se maximice el beneficio total. La solución implementada se corresponde con la técnica de **programación dinámica**.

Se han implementado y ejecutado las mismas versiones que el problema anterior tomando las mismas medidas.

Para el algoritmo calibrado se ha colocado un umbral de 100 microsegundos. Los resultados se muestran en el cuadro 6.

Cluster	$Size$	T_{sec}	T_{par}	T_{cal}	G_R
1	1152	7.59	7.04	3.38	51.98
	2304	60.44	55.36	22.62	59.14
	4608	483.72	430.37	168.22	60.91
2	1152	7.59	4.35	3.92	9.68
	2304	60.54	30.70	23.94	22.01
	4608	483.72	237.75	121.77	48.19
3	1152	7.59	5.32	4.30	19.17
	2304	60.54	24.81	18.90	23.82
	4608	483.72	167.04	95.89	42.59

Cuadro 6: Resultados para el problema de Asignación de Recursos.

En el cuadro 7 se muestran los resultados obtenidos por algunas iteraciones, sobre el cluster 1 con un tamaño de 2304.

Iterac.	Resultados
0	Carga: 577 576 576 576 Tiempos: 1951 7634 28140 14473
1	Carga: 1579 403 109 214 Tiempos: 14150 10397 6102 5723
2	Carga: 1251 434 200 420 Tiempos: 22620 24049 24207 27057
3	Carga: 1312 428 196 369 Tiempos: 8435 9037 8856 8134
4	Carga: 1325 403 188 389 Tiempos: 8604 7909 8665 8531
5	Carga: 1303 431 183 388 Tiempos: 8318 9077 8204 8515

Cuadro 7: Resultado por iteración para el Problema de Asignación de Recursos en el cluster 1 con un tamaño de 2304. $Carga$ y $Tiempo$ son los vectores donde cada posición se corresponde con un procesador.

6. Conclusiones y Trabajos Futuros

Hemos desarrollado una librería que permite realizar el equilibrado de carga dinámico en sistemas heterogéneos. El uso de la librería, en su ámbito de aplicabilidad, es altamente beneficioso y el esfuerzo requerido por parte del programador es mínimo ya que la aproximación seguida exige una mínima intrusión en el código del usuario. Como trabajo futuro nos planteamos ampliar el ámbito de aplicación de la librería a otras clases de programas. La metodología debería ser aplicable también a entornos homogéneos y heterogéneos no dedicados en los que la carga no es variable.

Referencias

[1] José Ignacio Aliaga, Francisco Almeida, José Manuel Badía-Contelles, Sergio Barrachina-Mir, Vicente Blanco Pérez, María Isabel Castillo, U. Dorta, Rafael Mayo, Enrique S. Quintana-Ortí, Gregorio Quintana-Ortí, Casiano Rodríguez, and Francisco de Sande. Parallelization of the gnu scientific library on heterogeneous systems. In *ISPDC/HeteroPar*, pages 338–345, 2004.

[2] Francisco Almeida, Daniel González, and Luz Marina Moreno. The master-slave paradigm on heterogeneous systems: A dynamic programming approach for the optimal mapping. *Journal of Systems Architecture*, 52(2):105–116, 2006.

[3] José Luis Bosque, David Gil Marcos, and Luis Pastor. Dynamic load balancing in heterogeneous clusters. In *Parallel and Distributed Computing and Networks*, pages 37–42, 2004.

[4] Javier Cuenca, Domingo Giménez, and Juan-Pedro Martinez. Heuristics for work distribution of a homogeneous parallel dynamic programming scheme on heterogeneous systems. *Parallel Comput.*, 31(7):711–735, 2005.