Biblioteca para el manejo automático de los buffers de comunicación en MPI

Reynaldo Gil-García y José Manuel Badía-Contelles

Resumen— **Una de las tareas más tediosas que realiza un programador que utiliza la biblioteca de paso de mensajes MPI es el control del tamaño de los buffers de comunicación. Cada vez que se envía un mensaje, hay que garantizar en el programa que el proceso destino tenga un buffer suficientemente grande para contener el mensaje que se le envía. Además es complicada la interfaz que brinda MPI para encapsular datos de distintos tipos en un solo mensaje. En este trabajo proponemos una librería en forma de varias clases en C++ que automatizan de forma eficiente el manejo de los buffers y garantiza que exista la capacidad necesaria para recibir los mensajes. Por tanto, libera al programador de esta tediosa tarea y le permite concentrarse en su algoritmo paralelo. Adicionalmente los mecanismos implementados facilitan el empaquetamiento de datos de distintos tipos en un solo mensaje, lo que aumenta la eficiencia del algoritmo paralelo sin necesidad de usar las complicadas funciones de empaquetamiento que ofrece MPI. La librería se ha utilizado con éxito en la construcción de varios algoritmos de agrupamiento paralelos y está disponible en forma de código libre.**

I. Introducción

Cuando se necesita ejecutar algoritmos de elevada complejidad computacional o se requieren procesar grandes volúmenes de datos, el poder de cómputo de un solo procesador no es suficiente para poder realizar el trabajo en un tiempo razonable. Una forma de tratar de resolver el problema es ejecutarlo en un conjunto de procesadores interconectados entre sí llamado computadora paralela. La computadora paralela tiene la ventaja adicional de tener generalmente más memoria que un monoprocesador, lo que contribuye también a disminuir los tiempos cuando se procesan conjuntos de datos muy grandes, pues evita la utilización de memoria secundaria.

Existen dos tipos fundamentales de computadoras paralelas: de memoria compartida o multiprocesador y de memoria distribuida o multicomputador. En los multiprocesadores, todos los procesadores pueden acceder a cualquier lugar de la memoria compartida. En los multicomputadores, cada procesador tiene su memoria privada y los procesadores intercambian información a través de una red de interconexión.

Aunque los multiprocesadores se consideran generalmente más fáciles de programar, pues cada procesador tiene acceso a todos los datos del problema en cualquier momento, el hardware es considerablemente más caro, ya que requiere dispositivos especiales para permitir la interconexión entre todas las memorias y los procesadores. Además el problema se hace cada vez más difícil a medida que aumenta la cantidad de procesadores, por lo que es un esquema poco escalable.

gil@app.uo.edu.cu, Universidad de Oriente, Cuba.
badia@icc.uji.es, Universidad Jaume I, Castellón, España.

Un multicomputador es esencialmente un conjunto de computadoras en red. Aunque los multicomputadores de mayor nivel pueden construirse a partir de computadoras y redes especialmente diseñadas para ellos, se han hecho muy populares los llamados clusters de PC. Con ellos, se ha logrado construir multicomputadores muy baratos aprovechando los bajos costos y elevadas prestaciones que alcanzan en la actualidad las computadoras personales y sus redes de interconexión. Los programas paralelos que se ejecutan en un multicomputador necesitan intercambiar mensajes entre sí para llevar a cabo el cómputo paralelo. Aunque estos mensajes podrían enviarse utilizando los mecanismos que ofertan los sistemas operativos de red para la comunicación, se han creado bibliotecas de programación paralela que independizan al programador del hardware y el sistema operativo. Esto permite la creación de programas paralelos portables a nivel de código fuente. Además las bibliotecas ofrecen un gran número de facilidades a los programadores para desarrollar y depurar su código.

Una de las bibliotecas que más se utiliza en la programación de multicomputadores y se ha convertido en un estándar es la biblioteca de paso de mensajes MPI. Esta biblioteca tiene versiones de código libre disponibles en Internet [1]. Además los mayores fabricantes de computadoras paralelas brindan versiones optimizadas de la misma para sus arquitecturas, por lo que los programas pueden aprovechar las ventajas presentes en las diversas arquitecturas con sólo recompilar el código.

MPI brinda funciones para comunicaciones individuales entre dos procesadores y para comunicaciones colectivas donde están involucrados un grupo de procesadores [4]. En cualquier mecanismo de comunicación los procesadores que reciben la información deben brindar un buffer de comunicación con tamaño suficiente para contener los datos que les llegan. Si esto no ocurre, el programa paralelo falla con un error grave de MPI. Garantizar esta capacidad de recepción es una tarea del programador que debe manejar cuidadosamente los buffers de recepción. El problema es que no siempre es posible conocer de antemano de qué tamaño van a ser los mensajes enviados. En estos casos el programador debe enviar primero el tamaño del buffer necesario en la recepción y después enviar realmente el mensaje deseado. Otra solución es tener siempre un buffer de recepción muy grande para garantizar que cualquier mensaje pueda recibirse, pero tiene el inconveniente de que en muchos casos no se conoce cuan grande pueda ser un mensaje, por lo que

esta solución provoca desperdicios de memoria y no da ninguna garantía de que no ocurra el problema.

Para resolver el problema proponemos una solución combinada, donde cada procesador tiene un buffer cuyo tamaño es conocido por el resto. En ese caso un procesador sólo envía un mensaje informativo cuando supone que éste es mayor que el buffer del procesador al que va a ser enviado. Además este mecanismo se implementa a nivel de la biblioteca por lo que es transparente al programador, que sólo debe preocuparse de enviar los mensajes.

Otro aspecto complicado de la programación en MPI es el paso de un mensaje con datos de distintos tipos o con tipos estructurados. En esos casos es necesario hacer una serie de llamadas a funciones que definen el tipo de dato a comunicar. Para paliar este problema implementamos un mecanismo de serialización que permite guardar en el mensaje cualquier combinación de tipos de datos. En particular el mecanismo es transparente para guardar en el buffer cualquier contenedor de la conocida biblioteca de clases STL de C++, por lo que facilita su uso en el programa paralelo.

El uso de mensajes de control adicionales no supone en la mayoría de los casos un gran sobrecoste, ya que éstos tan sólo se envían cuando es necesario modificar el tamaño de los buffers de recepción. Además, el mecanismo de serialización introducido permite agrupar de manera sencilla distintos datos en un solo mensaje, lo que en muchas ocasiones facilita la reducción del número de mensajes. Nuestra experiencia con el uso de este mecanismo en la resolución paralela de problemas de agrupamiento demuestra el poco efecto del mismo sobre la eficiencia [3], [2].

La biblioteca presentada se encuentra disponible en forma de código libre. El resto del trabajo se estructura como sigue. En primer lugar explicamos las soluciones teóricas dadas al problema. A continuación mostramos la interfaz de programación desarrollada y, por último, mostramos dos versiones de un programa paralelo simple utilizando directamente los mecanismos que brinda MPI y los que brinda nuestra biblioteca.

II. Soluciones teóricas

Para resolver el problema de los buffers creamos un mecanismo donde cada procesador tiene un buffer de comunicación y conoce el tamaño de los buffers en los restantes procesadores. Este mecanismo lo hemos desarrollado para dos casos: cuando en el programa paralelo se envían solamente mensajes individuales y cuando el programa utiliza solamente mecanismos de comunicaciones colectivas. En ambos casos los mensajes tienen una pequeña sobrecarga en forma de encabezado que permite el paso de información para el funcionamiento del algoritmo. El mecanismo pudiera generalizarse para tener en cuenta ambos casos, pero en aras de la sencillez y como no hemos necesitado esa generalización, nos hemos limitado a estos dos casos particulares.

El mecanismo para guardar y recuperar información a los buffers ha sido desarrollado para facilitar el uso de los contenedores de la STL. El mecanismo es completamente automático, tanto para tipos simples como para contenedores con ellos. Para guardar tipos estructurados el programador define funciones que especifican cómo llevar a cabo la serialización de la información.

A. Comunicaciones punto a punto

La idea para resolver el problema del manejo automático del buffer cuando se utilizan mensajes individuales es que cada procesador tenga un vector con el tamaño de los buffers en todos los procesadores. Inicialmente el buffer en cada procesador se crea de un tamaño que puede ser definido por el usuario. A medida que se ejecuta el algoritmo paralelo, el buffer de comunicación puede ir aumentando de tamaño dependiendo de los mensajes que necesite enviar o recibir el procesador.

Cuando un procesador necesita mandar un mensaje a otro y según la información que guarda en su vector, el mensaje no cabe en su destino, le envía un mensaje de control al procesador destino con el tamaño que se necesita que tenga el buffer en destino para poder recibir el mensaje. El procesador destino aumenta su buffer al menos hasta el tamaño necesario y responde con el nuevo tamaño del buffer. El procesador origen actualiza en su vector el nuevo tamaño del buffer en destino y envía el mensaje de datos.

Desde el punto de vista del algoritmo de manejo automático, se envían mensajes de datos y mensajes de control. Para distinguir el tipo de los mensajes se añade a los mismos un encabezado. Este encabezado se elimina en la recepción de forma tal que es transparente al programador. Los algoritmos 1 y 2 muestran los pasos para llevar a cabo el envío y la recepción. En los mismos *tamaBuff* es un vector con el tamaño de los buffer en cada procesador y cada procesador sabe a quién envía o de quién recibe el mensaje.

Algorithm 1 Envío del mensaje

```
Si tamaBuff[Destino] <Tamaño del mensaje de datos
    Crear mensaje de control de aumento de tamaño
    Poner en el mismo el tamaño necesario
    Enviar mensaje de control a Destino
    Recibir mensaje de control de Destino con el nuevo tamaño del buffer en destino
    Actualizar en tamaBuff el tamaño del buffer en Destino
Enviar mensaje de datos a Destino
```

B. Comunicaciones colectivas

En este caso optamos por tener en cada procesador un buffer del mismo tamaño, por lo que cada procesador sólo tiene en cuenta su tamaño del buffer. Cuando un procesador que participa en una operación colectiva incrementa el tamaño de su buffer, su envío

Algorithm 2 Recepción del mensaje

```
Recibir mensaje
Si mensaje es de control
    Leer del mensaje tamaño necesario del buffer
    Aumentar tamaño del buffer
    Enviar mensaje de control al procesador origen con el nuevo tamaño del buffer
    Recibir mensaje
```

no cabe en algún procesador destino. Por tanto, envía un mensaje de control que a la larga provoca que todos los procesadores incrementen su buffer hasta alcanzar el tamaño necesario. En particular, hemos desarrollado los algoritmos para implementar la difusión, la recogida y lo que llamamos procesamiento centralizado. Obviamente las ideas pueden extenderse para incluir todas las operaciones colectivas, pero sólo hemos desarrollado las que hemos necesitado en nuestra aplicación.

B.1 Difusión

Para resolver el problema en la difusión (*broadcast*), el procesador origen de la misma analiza si su buffer ha crecido respecto al tamaño anterior. Si esto no ocurre, sencillamente realiza la difusión poniendo en el encabezado que es un mensaje de datos. Si el buffer ha crecido, se realiza primero la difusión con un mensaje de control que informa del tamaño que se necesita en el buffer y posteriormente se envía el mensaje de datos. Por su parte, los procesadores receptores de la difusión realizan la operación, si les llega el mensaje de control aumentan el tamaño de sus buffer y vuelven a realizar la difusión. El algoritmo 3 muestra los pasos de la rutina para realizar la difusión. En la misma *origen* es el procesador que va a enviar los datos a los restantes procesadores y *TamaBuff* es un entero con el tamaño del buffer en todos los procesadores.

Algorithm 3 Difusión

```
Si es el procesador origen
    Si el tamaño del mensaje es mayor que TamaBuff
        Realizar difusión de mensaje de control con el tamaño del mensaje
        Aumentar el tamaño del buffer
        Actualizar TamaBuff al tamaño del buffer
    Realizar difusión con el mensaje de datos
Sino
    Realizar difusión
    Si es un mensaje de control
        Aumentar el tamaño del buffer
        Actualizar TamaBuff al tamaño del buffer
        Realizar difusión
```

B.2 Recogida

La recogida (*gatherv*) es un poco más complicada, pues varios procesadores envían datos y puede ocurrir que los datos que va a enviar otro procesador no tengan cabida en el procesador destino. Para resolver el problema hemos tenido que hacer una difusión al final de la recogida, de forma tal que todos los procesadores sepan si la recogida se llevó a cabo con éxito. En primer lugar, cada procesador analiza si su mensaje cabe en el buffer del procesador destino, teniendo en cuenta que este buffer está distribuido equitativamente entre los procesadores. Por tanto, el tamaño de cada mensaje debe ser menor que el tamaño del buffer dividido por la cantidad de procesadores. Si esto no ocurre en algún procesador, este manda un mensaje de control especificando el tamaño necesario del buffer. El procesador destino de la recogida analiza si le llegaron mensajes de control. Si todos son de datos, realiza una difusión para que todos los procesadores sepan que la recogida terminó. Si esto no ocurre, busca el máximo tamaño del buffer que le ha enviado un procesador y lo difunde. Con esta información los procesadores incrementan su buffer y repiten la recogida. Los pasos se muestran en el algoritmo 4.

Algorithm 4 Recogida

```
Si el tamaño del mensaje de datos es mayor que TamaBuff/CantidadProcesadores
    Crear Mensaje de Control con el tamaño requerido
    Realizar recogida enviando mensaje de control
Sino
    Realizar recogida enviando el mensaje de datos
Si es el procesador destino de la recogida
    Si alguno de los datos recogidos es de control
        Buscar el mayor tamaño de buffer requerido entre ellos
        Difundir mensaje de control con ese valor
        Aumentar el tamaño del buffer y actualizar la variable TamaBuff
        Realizar recogida enviando el mensaje de datos
    Sino
        Difundir que la recogida terminó OK
Sino
    Recoger difusión con el nuevo tamaño u OK
    Si es el mensaje de cambio de tamaño del buffer
        Aumentar el tamaño del buffer y actualizar la variable TamaBuff
        Realizar recogida enviando el mensaje de datos
```

B.3 Procesamiento centralizado

La operación de procesamiento centralizado realiza una recogida, un procesamiento en el procesador cen-

tro de la recogida y la difusión de los resultados del procesamiento. La implementamos como una operación independiente, pues la combinación de los tres pasos nos permite lograr una mayor eficiencia que su realización por separado. La idea es combinar las dos operaciones anteriores y aprovechar el hecho de que la difusión final de la recogida se puede utilizar para difundir los resultados del procesamiento. Los pasos se muestran en el algoritmo 5. Este procedimiento tiene además como parámetro la función que realiza el procesamiento en el procesador centro. En el mismo se necesita diferenciar entre tres tipos de mensajes de control:

- Tipo 1: Lo envían los procesadores cuando determinan que su mensaje no cabe en el buffer destino al hacer la recogida inicial.
- Tipo 2: Lo envía el procesador centro de la recogida para anunciarle a todos los procesadores que ocurrió en alguno de ellos el problema anterior. Por tanto, hay que volver a realizar la recogida.
- Tipo 3: Lo envía el procesador centro de la recogida para anunciarle a todos los procesadores que el resultado del procesamiento no cabe en sus respectivos buffers.

Algorithm 5 Procesamiento centralizado

```
Si el tamaño del mensaje de datos es mayor que TamaBuff/CantidadProcesadores
    Crear mensaje de control tipo 1 con el tamaño requerido
    Realizar recogida enviando mensaje de control
Sino
    Realizar recogida enviando el mensaje de datos
Si es el procesador centro de la operación centralizada
    Si alguno de los datos recogidos es de control
        Buscar el mayor tamaño del buffer requerido entre ellos
        Difundir mensaje de control tipo 2 con ese valor
        Aumentar el tamaño del buffer y actualizar la variable TamaBuff
        Realizar recogida enviando el mensaje de datos
    Realizar el procesamiento
    Si el mensaje resultado del procesamiento es mayor que TamaBuff
        Difundir mensaje de control tipo 3 con el tamaño que se necesita
        Aumentar el tamaño del buffer y actualizar la variable TamaBuff
    Difundir el mensaje con los resultados del procesamiento
Sino
    Realizar difusión
    Si es el mensaje de control tipo 2
        Aumentar el tamaño del buffer y actualizar la variable TamaBuff
        Realizar recogida enviando el mensaje de datos
        Realizar difusión
    Si es el mensaje de control tipo 3
        Aumentar el tamaño del buffer y actualizar la variable TamaBuff
        Realizar difusión
```

C. Manejo del buffer

Para guardar o extraer los mensajes del buffer de comunicación creamos mecanismos particulares de serialización, de forma tal que el programador debe extraer los datos del buffer en el mismo orden en que se introducen. También habilitamos una función para moverse a una posición determinada dentro del buffer, con el fin de posibilitar que el procesador que realiza la recogida pueda leer los datos que le envía cada procesador. En particular, automatizamos el manejo de los contenedores de la STL, de forma tal que guardar o leer del buffer cualquier contenedor de tipos simples es completamente automático. Para almacenar en el buffer tipos estructurados, es necesario que hereden de un objeto base e implementen las funciones de almacenamiento y restauración.

Otro aspecto de interés es el manejo del aumento del tamaño del buffer. La política que seguimos fue multiplicar por un factor el tamaño cada vez que fuera necesario modificarlo, con lo que tratamos que el buffer alcance rápidamente un tamaño estable y no sea necesario enviar más mensajes de control. Esta es la misma estrategia utilizada en la librería STL para adaptar el tamaño de contenedores como el vector.

III. Implementación

La biblioteca se implementó mediante clases C++. La clase *buffSTL* implementa el manejo del buffer. La misma hace un fuerte uso de las plantillas de C++ para lograr el manejo automático de los tipos simples y los contenedores de la STL. Sus métodos básicos son *write* y *read*. Ambos son métodos plantillas que permiten agregar objetos al buffer o leerlos del mismo respectivamente. Esta clase brinda además métodos auxiliares para cambiar el tamaño del buffer y desplazar el puntero donde se escribe dentro del buffer. Esto es necesario para las clases que implementan el manejo automático del buffer de comunicación. Otra clase muy relacionada con la anterior es *ObjetoBuff*, de la cual deben heredar todos los tipos estructurados que sea necesario guardar en el buffer. Los mismos deben sobrecargar los métodos virtuales puros *write* y *read* de *ObjetoBuff*.

El mecanismo de manejo automático se implementó en una jerarquía de clases. La clase base, *OMPI*, proporciona los mecanismos básicos de todo programa MPI, como la inicialización y la consulta de la cantidad de procesadores en la computación paralela. Proporciona además funciones auxiliares, como la que permite obtener el procesador siguiente en una lista circular o escribir mensajes de error colec-

tivos. Esta clase tiene un miembro privado del tipo *buffSTL* y redefine varias funciones que permiten el trabajo con el buffer de comunicación, como *writeBuff* y *readBuff*. De esta clase heredan las clases *OMPI_Local* y *OMPI_Global* que implementan los mecanismos para enviar mensajes punto a punto y comunicaciones colectivas respectivamente. Para garantizar la existencia única de las instancias de estas clases, sus métodos son estáticos. Con el objetivo de facilitar la programación, las funciones tienen los parámetros imprescindibles. Por supuesto, esto se logra a expensas de perder parte de la flexibilidad que ofrece MPI.

La clase *OMPI_Local* ofrece las funciones *send* y *recv*. Como estas funciones usan el buffer para la transmisión, el único parámetro que necesitan es el destino u origen del mensaje.

La clase *OMPI_Global* ofrece las funciones *Bcast, Gather* y *Centralized*. Las mismas tienen como parámetro el procesador centro de la comunicación. La función *Gather* devuelve un entero con el desplazamiento en el buffer de los datos de cada procesador. La función *Centralized* recibe el puntero a la función *callback* que se necesita llamar para realizar el procedimiento en el procesador centro. Esta función *callback* recibe como parámetro un entero con el desplazamiento en el buffer de los datos de cada procesador.

Al probar la implementación de nuestra librería, hemos encontrado problemas con algunas implementaciones de MPI. En una la implementación del broadcast no admite que en los procesadores que reciben el mensaje se pase un buffer grande. Aunque la lógica indica que ese buffer sólo se debe utilizar para la recepción, parece que la recepción provoca efectos colaterales. En ese caso, tuvimos que realizar una difusión adicional mandando el tamaño de los datos antes de enviar realmente los datos. Otro problema de algunas implementaciones es que al procesador que hace la recogida no le llega lo que él envió. En ese caso, tuvimos que copiar en el buffer de recepción lo que enviaba el procesador.

IV. Ejemplo

En esta sección mostramos el código fuente de dos implementaciones de un mismo problema. Una implementación utiliza directamente las herramientas que brinda MPI y las que ofrece nuestra biblioteca. Para ello escogimos un problema sencillo que requiere el paso de mensajes colectivos donde los receptores no conocen el tamaño que tendrán los mensajes. El problema que paralelizamos puede describirse del siguiente modo: Dado un conjunto de n puntos en R^2, buscar cuáles son los puntos que tienen más de k puntos en un entorno de radio r. Se trata de un problema completamente paralelo.

Para resolverlo, distribuimos equitativamente el trabajo con los puntos entre los procesadores para lograr un equilibrio de carga. Inicialmente el procesador 0 difunde n, k y r y las coordenadas de los puntos. Posteriormente cada procesador calcula la distancia de cada uno de sus puntos a los restantes y cuenta la cantidad de vecinos de cada punto. Los índices de los puntos que tienen más de k vecinos se recogen en el procesador 0 para reunir los resultados.

El algoritmo 6 muestra el programa utilizando directamente las funciones de MPI. En el mismo se observa cómo, utilizando las funciones de MPI, hay que garantizar que el buffer de recepción tenga suficiente espacio. Además para pasar datos de distintos tipos se requiere usar varios mensajes. Otra alternativa sería usar estructuras, mantener los datos en las estructuras y utilizar los mecanismos que brinda MPI para tipos de datos derivados, que son más complicados.

Algorithm 6 Programa usando MPI

```
#include <mpi.h>
#define ALL MPI_COMM_WORLD
int main(int argc, char **argv) {
int n,k;
float r,*puntos;
int nProc,iProc;
int *indices; // Índices de los puntos
int cant; // Cantidad de puntos
int *indicesG; // Índices a nivel global
int *cants; // Cantidad por procesador
int *desplas; // Posiciones de recogida
int cantG; // Cantidad total

MPI_Init(&argc,&argv);
MPI_Comm_size(ALL, &nProc);
MPI_Comm_rank(ALL, &iProc);
if ( iProc==0 )
  Leer_Datos(&n,&k,&r,puntos);
MPI_Bcast(&n,1,MPI_INT,0,ALL);
MPI_Bcast(&k,1,MPI_INT,0,ALL);
MPI_Bcast(&r,1,MPI_FLOAT,0,ALL);
if ( iProc!=0 )
  puntos=calloc(sizeof(float),2*n);
MPI_Bcast(puntos, 2*n, MPI_FLOAT, 0,
ALL);
Busca_Puntos( ... );
MPI_Gather(&cant, 1, MPI_INT, cants, nProc,
MPI_INT, 0, ALL);
if ( iProc==0 ) {
  CalculaDesplaz(cants,desplas,&cantG);
  indicesG=calloc(sizeof(int),cantG);
}
MPI_Gatherv(indices, cant, MPI_INT, indicesG,
cants, desplas, MPI_INT, 0, ALL);
}
```

El algoritmo 7 muestra la implementación utilizando nuestra biblioteca OMPI. Puede observarse que el programador no necesita preocuparse por el tamaño del buffer de recepción pues su manejo es automático, lo que en muchas ocasiones reduce la cantidad de mensajes. También pueden mandarse fácilmente cualquier combinación de tipos de datos en un solo mensaje. Además el uso de las funciones de comunicación es mucho más fácil, pues proporciona una

interfaz más sencilla que las funciones de MPI.

Algorithm 7 Programa usando OMPI

```
#include <vector>
#include "OMPI.h"
int main(int argc, char **argv) {
int k;
float r;
vector<float>puntos;
vector<int>indices; // Índices de los puntos
int despla;

OMPI_Global::Init(argc,argv);
if ( OMPI_Global::Proc()==0 ) {
  Leer_Datos(k,r,puntos);
  OMPI_Global::write(k);
  OMPI_Global::write(r);
  OMPI_Global::write(puntos);
  }
OMPI_Global::Bcast();
OMPI_Global::read(k);
OMPI_Global::read(r);
OMPI_Global::read(puntos);
Busca_Puntos( ... );
OMPI_Global::write(indices);
despla=OMPI_Global::Gather();
if ( OMPI_Global::Proc()==0 )
  for ( int i=1;i<OMPI_Global::Proc();i++) {
    OMPI_Global::setPosBuf(despla*i);
    OMPI_Global::read(indices);
  }
}
```

V. Conclusiones

En el trabajo exponemos una biblioteca de clases C++ que libera al programador de la tediosa tarea de garantizar que exista suficiente espacio en el buffer de recepción de los mensajes MPI. La biblioteca permite además enviar fácilmente mensajes con datos de distintos tipos, lo que puede conducir a que se reduzca el número de mensajes y, por tanto, ofrecer una mayor eficiencia del programa paralelo. La funcionalidad que se ofrece ha tratado de ser lo más simple posible para facilitar la programación, al precio de disminuir la flexibilidad que ofrece MPI. La biblioteca es portable y fácilmente ampliable para incluir mayores funcionalidades.

La biblioteca OMPI se ha utilizado con éxito en la paralelización de algoritmos de agrupamiento. Los resultados obtenidos en este caso han sido positivos, dado que las prestaciones experimentales se han visto poco afectadas por el uso de los mecanismos de control añadidos por la biblioteca. La misma se encuentra disponible en forma de código libre.

Referencias

[1] The message passing interface (mpi) standard. `http://www-unix.mcs.anl.gov/mpi/`.

[2] R. J. Gil-García, J. M. Badía-Contelles, and A. Pons-Porrata. A parallel algorithm for incremental compact clustering. *Lecture Notes on Computer Sciences*, 2790:310–317, 2003.

[3] R. J. Gil-García and J.M. Badía-Contelles. GLC Parallel Clustering Algorithm. In *Pattern Recognition. Advances and Perspectives. Research on Computing Science CIARP'2002*, pages 383–394, 2002. In Spanish.

[4] M. Snir, S. Otto, and et al. *MPI: The Complete Reference*. The MIT Press, 1998.