Universitat Oberta de Catalunya

uoc.edu

Implementación distribuida maleable del método Laplace

José Reina León

Máster en Ingeniería Informática

Computación de Altas Prestaciones

Sergio Iserte Agut

Josep Jorba Esteve

Enero 2024

FICHA DEL TRABAJO FINAL

Título del trabajo:	Implementación distribuida maleable del método Laplace
Nombre del autor:	José Reina León
Nombre del consultor:	Sergio Iserte Agut
Nombre del PRA:	Josep Jorba Esteve
Fecha de entrega (mm/aaaa):	01/2024
Titulación:	Máster en Ingeniería Informática
Área del Trabajo Final:	Computación de Altas Prestaciones
Idioma del trabajo:	Castellano
Palabras clave:	MPI, Recursos dinámicos, DMR

Resumen del trabajo

La computación de alto rendimiento es una herramienta clave en multitud de áreas donde la capacidad de cómputo y alta capacidad de almacenamiento son necesarias. Esto es llevado a cabo, en la mayoría de casos, por clústeres de ordenadores trabajando coordinados para aumentar la capacidad de cálculo individual. Una vez una tarea es asignada a ciertos nodos de computación sólo cabe esperar hasta su finalización para liberar los recursos que le han sido asignados.

La maleabilidad se presenta como la capacidad de adaptación a un número variable de nodos de computación durante la ejecución de una tarea concreta. Esto permite una mejora en el rendimiento general del clúster permitiendo aumentar la cantidad de trabajos finalizados por unidad de tiempo. A su vez, es capaz de permitir que trabajos prioritarios puedan ejecutarse en caso de que no existan recursos en el momento que es incluido en el gestor de trabajos.

En este proyecto se va a implementar la maleabilidad de una aplicación real mediante el uso de la API DMR. Así mismo, se ejecutará sobre un clúster HPC real y se realizarán las pruebas y análisis correspondiente con el fin de demostrar esa mejora de rendimiento que podemos alcanzar gracias a la maleabilidad de las aplicaciones.

Abstract

High-performance computing (HPC) is a key tool in many areas where computing power and high storage capacity are needed. This is usually carried out by clusters of computers working together to increase individual computing power. Once a task is assigned to certain computing nodes, the only thing left to do is wait until it is finished to release the resources that have been assigned to it.

Malleability is presented as the ability to adapt to a variable number of computing nodes during the execution of a specific task. This allows for an improvement in the overall performance of the cluster, allowing for an increase in the number of jobs completed per unit of time. It is also able to allow priority jobs to be executed even if there are no resources available at the time they are included in the job manager.

In this project, we will implement the malleability of a real application using the DMR API. It will also be executed on a real HPC cluster and the corresponding tests and analysis will be carried out in order to demonstrate the performance improvement that we can achieve thanks to the malleability of applications.

Tabla de contenidos

Índice de figuras

Índice de tablas

1. Introducción

1.1 Contexto y justificación del trabajo

En la actualidad, la computación de altas prestaciones, en adelante HPC, se ha convertido en una herramienta indispensable para la resolución de problemas complejos en multitud de áreas. Áreas como la biología, las matemáticas y la ingeniería entre muchas otras se benefician de este tipo de recursos computacionales.

La HPC hace uso de supercomputadores para llevar a cabo la ejecución de las tareas. Estos supercomputadores están construidos con una arquitectura que posibilita la colaboración de miles de ordenadores, permitiendo así un acceso coordinado y eficiente a estos recursos para abordar problemas complejos.

Debido a la complejidad y la gran demanda de este tipo de recursos es indispensable que las diferentes tareas que deben procesarse en los supercomputadores sean organizadas y priorizadas en función de criterios como la disponibilidad de recursos en el supercomputador y su propia prioridad en la utilización de dichos recursos computacionales. Esta tarea es llevada a cabo por lo que se conoce como gestor de trabajos [7].

Sin embargo, esta asignación de recursos en HPC resulta ser un problema complejo. Una vez lanzado un trabajo a los nodos de cálculo, solo se puede optar por dos opciones para volver a recuperar los recursos asignados a cada una de las diferentes tareas en ejecución: esperar a su finalización o finalizarlo prematuramente. Esta situación provoca que en determinados casos, algunos nodos puedan estar desocupados, por la imposibilidad de ajustar las tareas en ejecución al número de nodos disponibles y las tareas pendientes de ser lanzadas necesiten más nodos que los que están libres en un determinado momento. Además, también puede producirse que en un determinado momento, trabajos de baja prioridad estén utilizando un gran número de nodos, lo que puede impedir que otros trabajos más prioritarios puedan ejecutarse.

Así tenemos que la problemática de la asignación de recursos en HPC abarca varios aspectos, entre ellos la inflexibilidad y la eficiencia:

- La asignación de recursos es inflexible, ya que una vez que se han asignado a un determinado trabajo, no es posible modificarlos sin finalizar dicho trabajo. Esto puede limitar la capacidad de adaptación a cambios en las necesidades computacionales.
- La eficiencia también se ve comprometida, ya que es posible que trabajos de baja prioridad utilicen un gran número de recursos, lo que a su vez puede impedir que otros trabajos más prioritarios que sean agregados posteriormente al gestor de trabajos se ejecuten de manera oportuna.

Esta falta de eficiencia en la asignación de recursos puede tener un impacto significativo en el rendimiento general del sistema HPC y en la satisfacción de las demandas de procesamiento más críticas.

La biblioteca DMR [9] surge como una solución a los retos inherentes a la asignación de recursos en entornos de HPC. Con su implementación, los programas adquieren la capacidad de ajustarse dinámicamente a una cantidad variable de nodos, permitiendo la contracción en presencia de trabajos más prioritarios o la expansión cuando los recursos vuelven a estar disponibles. Esta flexibilidad mejora significativamente la eficiencia en la asignación de recursos, proporcionando adaptabilidad a las cambiantes demandas computacionales incrementando la ocupación de recursos y por lo tanto incrementando la cantidad de trabajos finalizados por unidad de tiempo.

DMR utiliza la biblioteca MPI (*Message Passing Interface*) para facilitar la comunicación entre nodos de cómputo. Esto posibilita el trabajo en paralelo, permitiendo una colaboración eficiente de todos los nodos para obtener resultados finales de manera más rápida y eficaz.

DMR se destaca en el panorama de asignación de recursos en HPC gracias a sus ventajas distintivas. Su flexibilidad para ajustarse dinámicamente, su escalabilidad en grandes sistemas HPC y su simplicidad de uso e integración en programas existentes la posicionan como una herramienta clave para optimizar el rendimiento y la eficiencia en estos entornos computacionales avanzados.

1.2 Objetivos

El objetivo de este trabajo es adaptar una aplicación que se ejecuta sobre varios nodos utilizando la biblioteca MPI para el intercambio de información entre procesos para el uso en conjunto con la biblioteca DMR. De este modo, la aplicación tendrá la capacidad de adaptarse a un número variable de nodos de computación, tanto expandiéndose como contrayéndose, con el fin de adaptarse a los recursos que le sean asignados en cada momento por el gestor de trabajos. Además, con el fin de utilizar toda la capacidad de cálculo de los nodos, se utilizará la biblioteca openMP para implementar esta funcionalidad.

Para llevar a cabo la ejecución se emplea un sistema HPC real. En este caso concreto dispondremos del clúster Minotauro instalado en el *Barcelona Supercomputing Center* (BSC).

Para cumplir con los objetivos aquí planteados es necesario alcanzar las siguientes metas:

- Profundizar en las bibliotecas MPI sobre las que se desarrolla la biblioteca DMR para la obtención de aplicaciones maleables. Así como el uso de la API DMR para llevar a cabo las modificaciones de la aplicación inicial. También será necesario volver a repasar aspectos de la biblioteca openMP para implementar la capacidad de multiprocesamiento de la aplicación en cada uno de los nodos.
- Familiarizarse con el sistema HPC para trabajar y llevar a cabo las ejecuciones necesarias.
- Seleccionar y modificar la aplicación inicial para su comportamiento maleable mediante la API DMR.
- Realizar las oportunas pruebas de rendimiento para obtener los resultados y analizar el comportamiento de la aplicación desarrollada.
- Se publicará el código fuente en un repositorio público para el acceso a cualquier interesado. En nuestro caso se ha utilizado la plataforma GitHub [12].

En los siguiente apartados se irán desarrollando cada uno de los diferentes puntos que hemos destacado con mayor profundidad.

1.3 Enfoque y método seguido

La adaptación de una aplicación al comportamiento maleable en un sistema HPC implica una serie de pasos estructurados para garantizar una integración correcta. En primer lugar, se debe iniciar con la familiarización del usuario con el sistema HPC específico, como en este caso es el clúster Minotauro del BSC. Este proceso implica conocer en detalle los recursos disponibles en el sistema y comprender los procedimientos necesarios para acceder a estos y ejecutar trabajos de manera efectiva.

Posteriormente, es fundamental profundizar en el conocimiento de las bibliotecas MPI y DMR. La biblioteca DMR se construye sobre la base de las bibliotecas MPI, por lo que es esencial comprender su funcionamiento para aprovechar las capacidades de DMR. Además, familiarizarse con la API DMR se vuelve necesario, ya que esta proporciona las funciones clave para adaptar una aplicación al comportamiento maleable. Por último, es necesario repasar aspectos sobre la biblioteca openMP para utilizar toda la potencia de cálculo disponible en cada nodo. De este modo, no solo se implementará una aplicación con el fin de demostrar su maleabilidad a través de la biblioteca DMR sino que además se obtendrá una mayor eficiencia durante la ejecución.

Con esta base establecida, se procederá a la modificación de la aplicación inicialmente seleccionada. Esta fase implica la incorporación de las funciones de la API DMR para gestionar el comportamiento maleable de la aplicación, así como la adaptación para garantizar un funcionamiento correcto con un número variable de nodos de cómputo.

Una vez realizadas las modificaciones, la aplicación se ejecuta en el sistema HPC. Los resultados de estas ejecuciones son analizados minuciosamente para evaluar el comportamiento maleable de la aplicación en diferentes escenarios. Este análisis proporciona información sobre la capacidad de la aplicación para adaptarse a cambios en el número de nodos de computación

Los resultados obtenidos de las ejecuciones demuestran que la aplicación modificada es capaz de adaptarse de manera efectiva a un número variable de nodos de computación, validando así el éxito del proceso de adaptación al comportamiento maleable en el entorno HPC.

Por último, el código de la aplicación modificada será publicado en un repositorio de acceso público para poder ser consultado y utilizado por cualquier interesado en el uso de la maleabilidad en problemas reales.

1.4 Planificación del Trabajo

La consecución exitosa del objetivo final del proyecto requiere una planificación cuidadosa de las actividades involucradas. En términos resumidos, estas actividades pueden ser agrupadas de la siguiente manera:

1. **Familiarización con el sistema HPC**: Iniciar el proceso con la comprensión detallada del sistema HPC seleccionado, como es el clúster Minotauro del BSC. Esto implica conocer los recursos disponibles y familiarizarse con los procedimientos para acceder a ellos y ejecutar trabajos.

2. **Profundización en las bibliotecas MPI, openMP y DMR**: Se profundizará en la comprensión de las bibliotecas MPI y DMR, siendo fundamental para la adaptación exitosa de la aplicación al comportamiento maleable del sistema HPC. También se repasarán aspectos sobre el uso de la biblioteca openMP para el procesamiento en paralelo.

3. **Modificación de la aplicación inicial:** Realizar ajustes a la aplicación original seleccionada, incorporando las funciones de la API DMR para gestionar su comportamiento maleable y adaptándola para un funcionamiento óptimo con un número variable de nodos de cómputo.

4. **Obtención de resultados y análisis**: Ejecutar la aplicación modificada en el sistema HPC y llevar a cabo un análisis exhaustivo de los resultados obtenidos. Este análisis proporcionará información valiosa sobre la capacidad de la aplicación para adaptarse a diferentes configuraciones de nodos.

5. **Publicación del código desarrollado**: Publicar el código obtenido en un repositorio público para

poder ser accedido por cualquier interesado.

Una planificación realista y ordenada de estas actividades es esencial para asegurar un progreso fluido y eficiente hacia la consecución de los objetivos del trabajo. En la tabla 1 se muestra el diagrama de *Gantt* correspondiente, donde se detallan las diferentes tareas a realizar así como cada una de las entregas correspondientes.

Se presenta la distribución de las tareas en cada uno de los grupos de actividades y cómo estas se alinean con las distintas entregas que componen el seguimiento del proyecto:

- En la primera entrega se ha realizado la familiarización con el sistema HPC, así como se han repasado los distintos conceptos de MPI y openMP que se van a utilizar en la modificación de la aplicación seleccionada. También se ha seleccionado la aplicación en cuestión.
- En la segunda entrega se han adquirido los conocimientos necesarios sobre la API DMR y se han realizado las modificaciones necesarias de la aplicación para poder aplicar con mayor facilidad la biblioteca DMR.
- En la tercera entrega se han desarrollado las modificaciones y ajustes necesarios en la aplicación seleccionada. Todo ello ha sido añadido a su parte correspondiente en el presente documento.
- En la cuarta y última entrega se han desarrollado las diferentes pruebas sobre el sistema real, así como el análisis y evaluación de los resultados obtenidos. Se ha acabado de integrar estos resultados y análisis al documento. El código se ha publicado en repositorio de acceso público. Finalmente, se ha finalizado el trabajo de redacción final de la memoria.

Diagrama de Gantt

Tareas	25/09	09/10	23/10	06/11	20/11	04/12	18/12	01/01	09/01
PAC 1									
Planificación del proyecto									
Determinar objetivos y tareas									
Búsqueda de información									
Profundizar y/o repasar MPI y openMP									
Documentación 1ª entrega									
PAC 2									
Determinar herramientas a utilizar									
Seleccionar aplicación a modificar									
Familiarizarse con el entorno HPC									
Iniciarse en la biblioteca DMR									
Configuración entorno DMR									
Documentación 2ª entrega									
PAC 3									
Factorización de la aplicación									
Implementar funcionalidad DMR									
Implementar funcionalidad openMP									
Documentación 3ª entrega									
PAC 4									
Ejecución de la aplicación maleable									
Análisis de los resultados obtenidos									
Publicación del código obtenido									
Conclusiones									
Documentación 4ª entrega									

Tabla 1: Diagrama de Gantt.

Las distintas etapas se definen de manera secuencial. De este modo, hasta no finalizar cada una de las etapas previas no se avanza hacia la siguiente meta. En algunas ocasiones pueden observarse que algunas tareas se realizan simultáneamente, esto es debido a que se tratan de tareas complementarias en su realización.

Además de las distintas etapas y fases debemos indicar que el proyecto ha sido tutorado a través de videoconferencias cada 15 días. En estas tutorías se ha intercambiado tanto información acerca de las diferentes tecnologías utilizadas, como recursos a consultar y resolución de los problemas que han ido apareciendo a lo largo de su desarrollo. Siempre que ha sido necesario se ha recurrido a la compartición de archivos y pantalla para poder seguir adecuadamente el desarrollo de las tutorías. También se ha utilizado el correo electrónico con el mismo fin.

1.5 Breve sumario de productos obtenidos

En este proyecto el principal producto obtenido es una versión MPI+OpenMP maleable para el cálculo de Laplace utilizando DMR. La aplicación hace uso además de la biblioteca MPI para el intercambio de mensajes entre los diferentes nodos, siendo además la base sobre la que está implementada la biblioteca DMR. Igualmente se utiliza la biblioteca openMP para la computación en paralelo por parte de todos los microprocesadores disponibles en cada nodo.

1.6 Breve descripción de los otros capítulos de la memoria

En el resto de capítulos se profundizará en cada una de las diferentes etapas que se compone el proyecto, de manera que sea posible tanto la reproducibilidad del mismo como la adaptación de las técnicas utilizadas a otros ejemplos en las que se desee aplicar las técnicas aquí explicadas y desarrolladas.

2. Desarrollo del proyecto

2.1 Herramientas y entorno de desarrollo

Para el desarrollo del presente trabajo se ha podido contar con el clúster Minotauro del BSC. El acceso a este clúster ha facilitado en gran medida el trabajo ya que cuenta con todo el software necesario para el desarrollo del mismo. A pesar de que el clúster no cuenta con un hardware actualizado ha cumplido con su función adecuadamente, ya que no se ha necesitado la máxima potencia de cálculo sino la mejor plataforma posible tanto para el desarrollo como para la ejecución y pruebas del código.

Minotauro es un clúster formado por 39 nodos, basado en servidores Bullx R421-E4, con Linux como sistema operativo y llevándose a cabo la comunicación sobre enlaces Ethernet. La configuración de cada uno de los nodos que lo forman es la siguiente:

- 2 procesadores Intel Xeon E5-2630 v3 (Haswell) de 8 núcleos, (cada núcleo a 2.4 GHz, y con un caché de 20 MB L3).
- 2 tarjetas K80 NVIDIA GPU.
- 128 GB de memoria principal, distribuida en 8 DIMMs de 16 GB -- DDR4 @ 2133 MHz - ECC SDRAM –.
- Rendimiento pico: 226.98 TFlops (K80) + 23.96 TFlops (Haswell) = 250.94 Tflops.
- 120 GB SSD (Disco de Estado Sólido) como almacenamiento local.
- 1 PCIe 3.0 x8 8GT/s, Mellanox ConnectX®-3 FDR 56 Gbit.
- 4 puertos Gigabit Ethernet.

La distribución de Linux instalada es concretamente una *Red Hat Enterprise Linux Server release* 6.7 (Santiago).

A nivel local, el entorno de desarrollo está compuesto por una computadora portátil con sistema operativo *Debian* 12 (*bookworm*). Básicamente, cualquier equipo hubiese desempeñado su función siempre que contase con la posibilidad de conexión segura mediante el protocolo *ssh*.

Para desarrollar nuestro trabajo se utiliza tanto el hardware como el software instalado en el clúster Minotauro. Así, la estación de trabajo empleada utiliza los recursos remotamente a través de una conexión *ssh*.

2.2 Descripción DMR

En primer lugar, vamos a mostrar y explicar brevemente el funcionamiento de la biblioteca DMR. En el siguiente gráfico podemos observar los diferentes elementos involucrados en su funcionamiento.

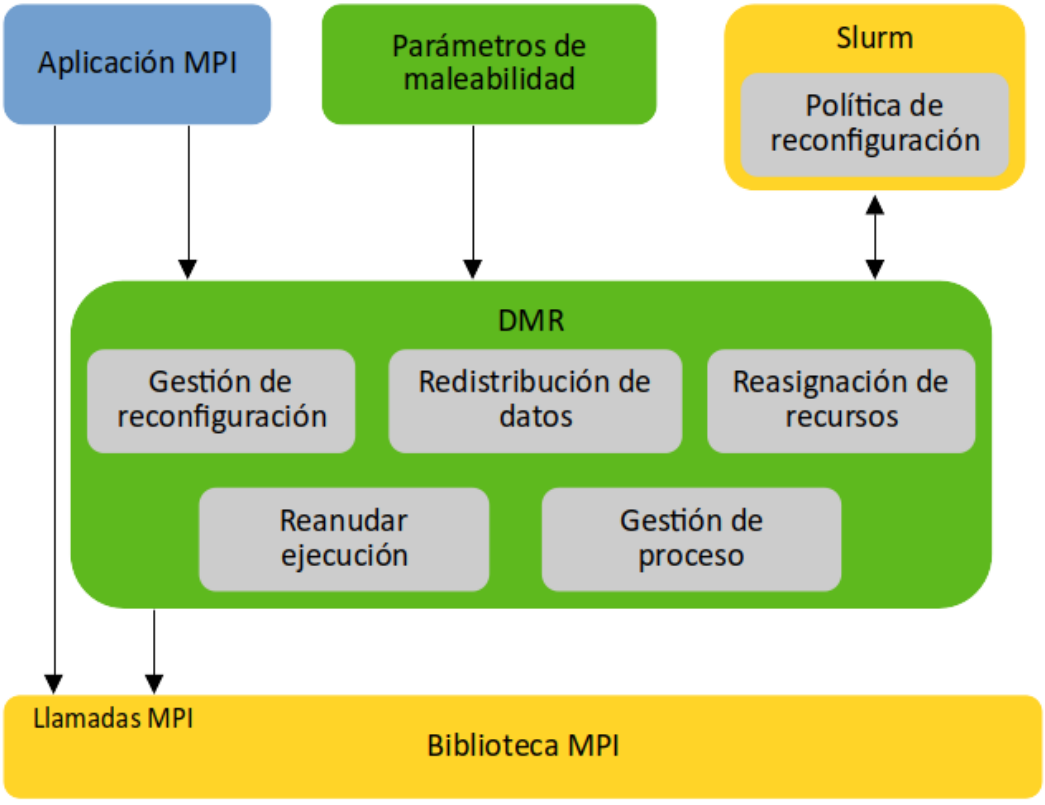

Figura 1: Esquema API DMR.

La aplicación desarrollada hace uso de las llamadas a la biblioteca MPI para su funcionamiento por defecto y debe incluir además las necesarias a la biblioteca DMR para poder comportarse de manera

maleable. La maleabilidad es manejada en tiempo de ejecución a través del gestor de trabajos *Slurm*, que a través de DMR gestiona los recursos asignados a la aplicación maleable en ejecución haciendo uso de la redistribución de datos.

El funcionamiento de la aplicación maleable mediante la biblioteca DMR puede descomponerse en una serie de pasos. Como ejemplo, se utiliza una aplicación que utiliza dos procesos iniciales y se reconfigura en cuatro procesos. En la figura 2 se muestran los dos procesos iniciales que pertenecen al grupo *Comm* 1 y cada uno de ellos posee sus propios datos.

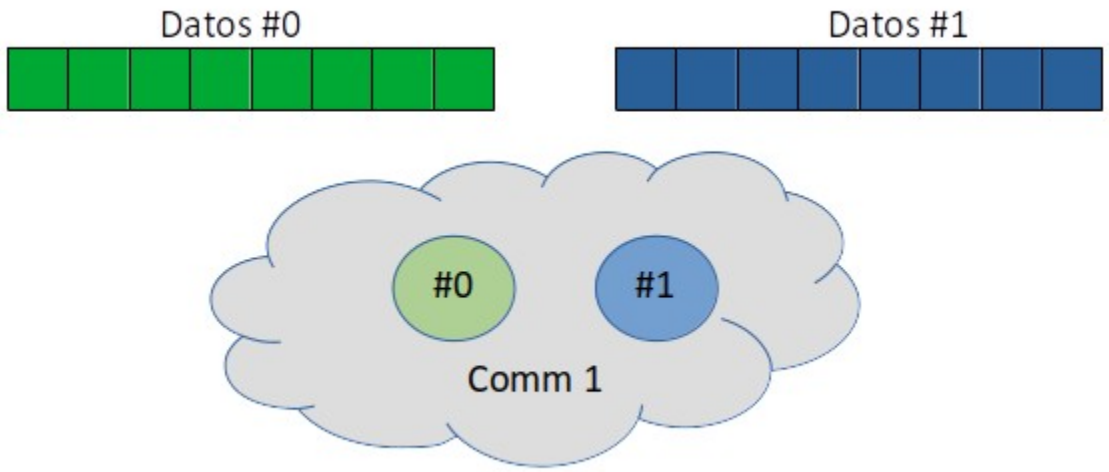

Figura 2: Procesos iniciales en DMR.

Cuando se lleva a cabo la reconfiguración el primer paso es crear un nuevo grupo, que en el ejemplo mostrado es llamado *Comm* 2, que contiene los nuevos procesos. En la siguiente figura podemos ver ambos grupos con sus respectivos procesos y como los pertenecientes al nuevo grupo todavía no tienen sus datos.

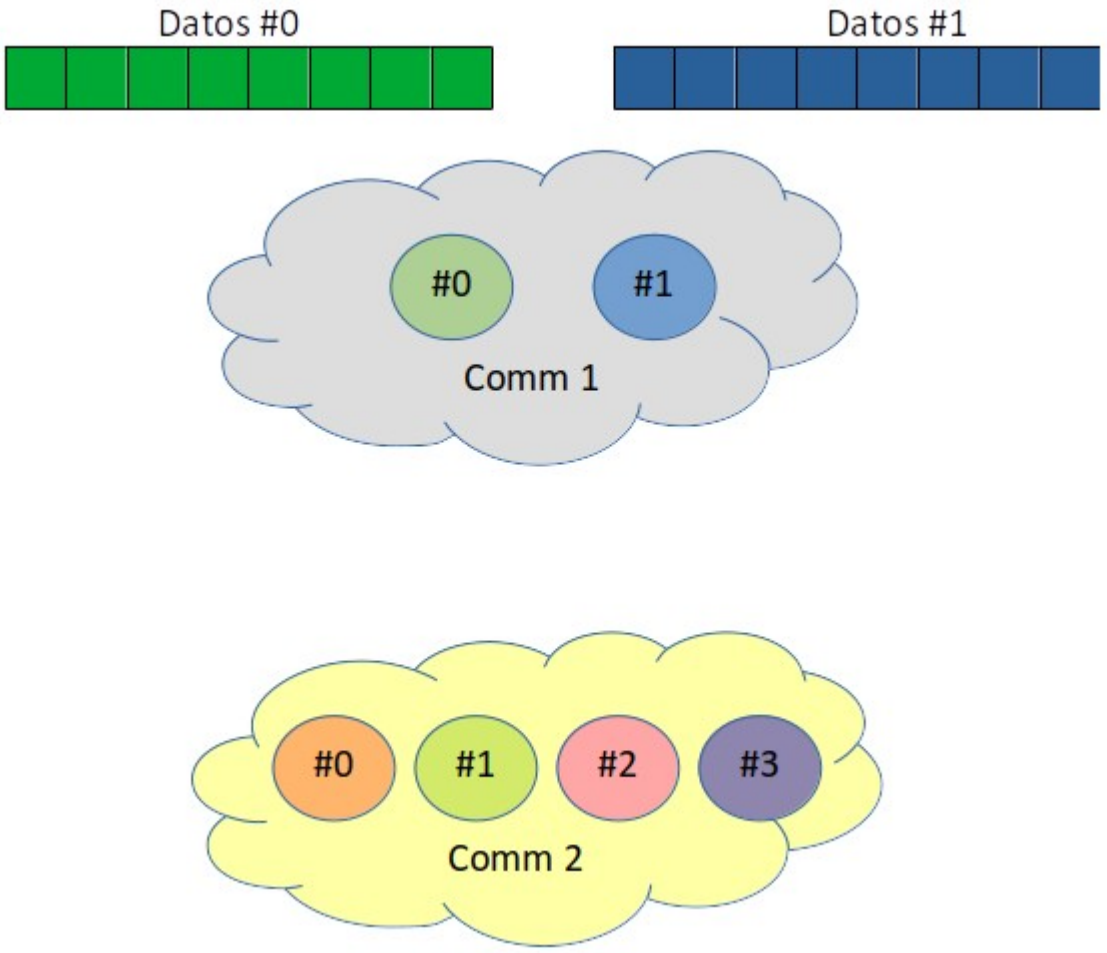

Figura 3: Generación de nuevo grupo y procesos en DMR.

El siguiente paso es preparar los datos existentes para redistribuirlos entre los nuevos procesos. En la figura 4 podemos observar cómo cada proceso ha dividido los datos en 2 dado que este es el factor de expansión. Cada proceso existente debe redistribuir sus datos a 2 nuevos procesos. En caso de que el factor de expansión fuese otra cantidad los datos deberían redistribuirse proporcionalmente a ese valor.

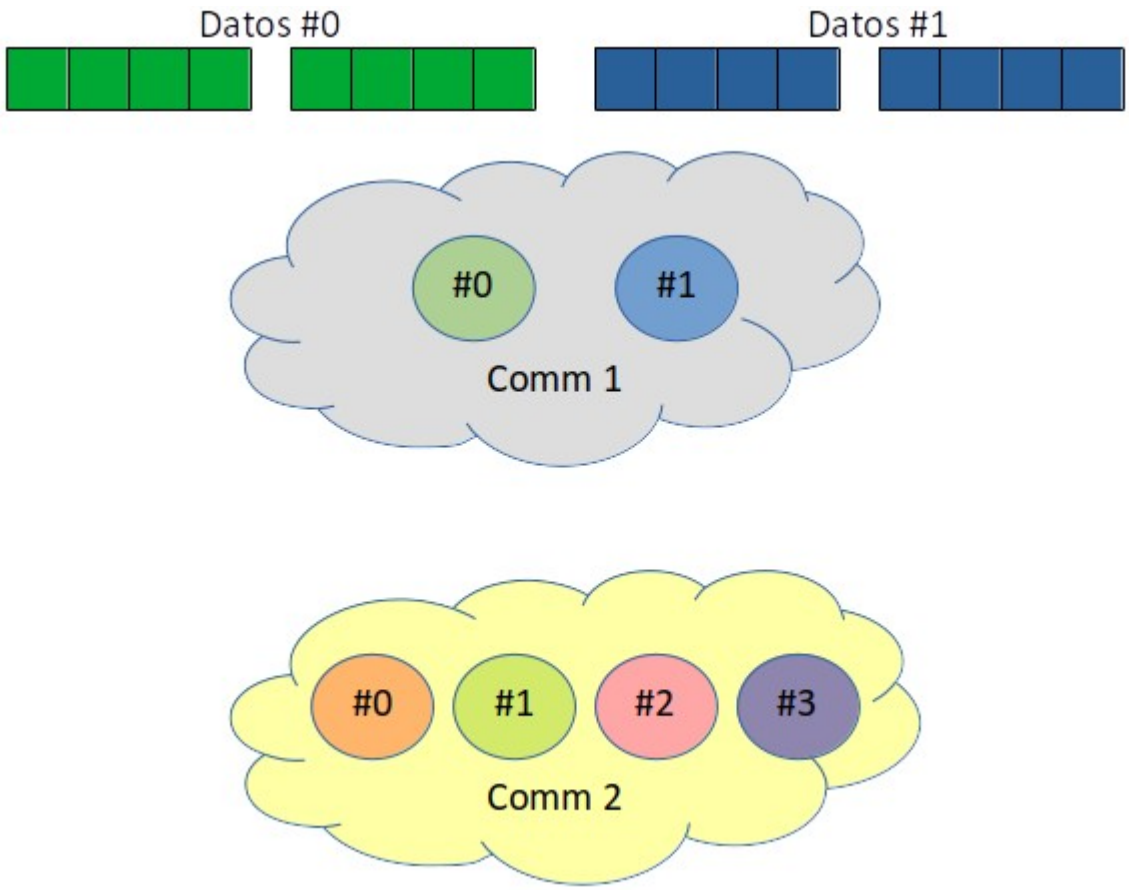

Figura 4: Preparación de datos para su distribución en DMR.

En el siguiente paso los procesos existentes envían los datos a los procesos del nuevo grupo. Como puede verse en la siguiente figura esto se lleva a cabo de manera ordenada y no cabe lugar al envío de datos hacia los nuevos procesos de manera aleatoria. Esto también nos permite mantener una coherencia pudiendo prever cómo se reestructura el sistema cuando se efectúan las expansiones y contracciones en número de procesos.

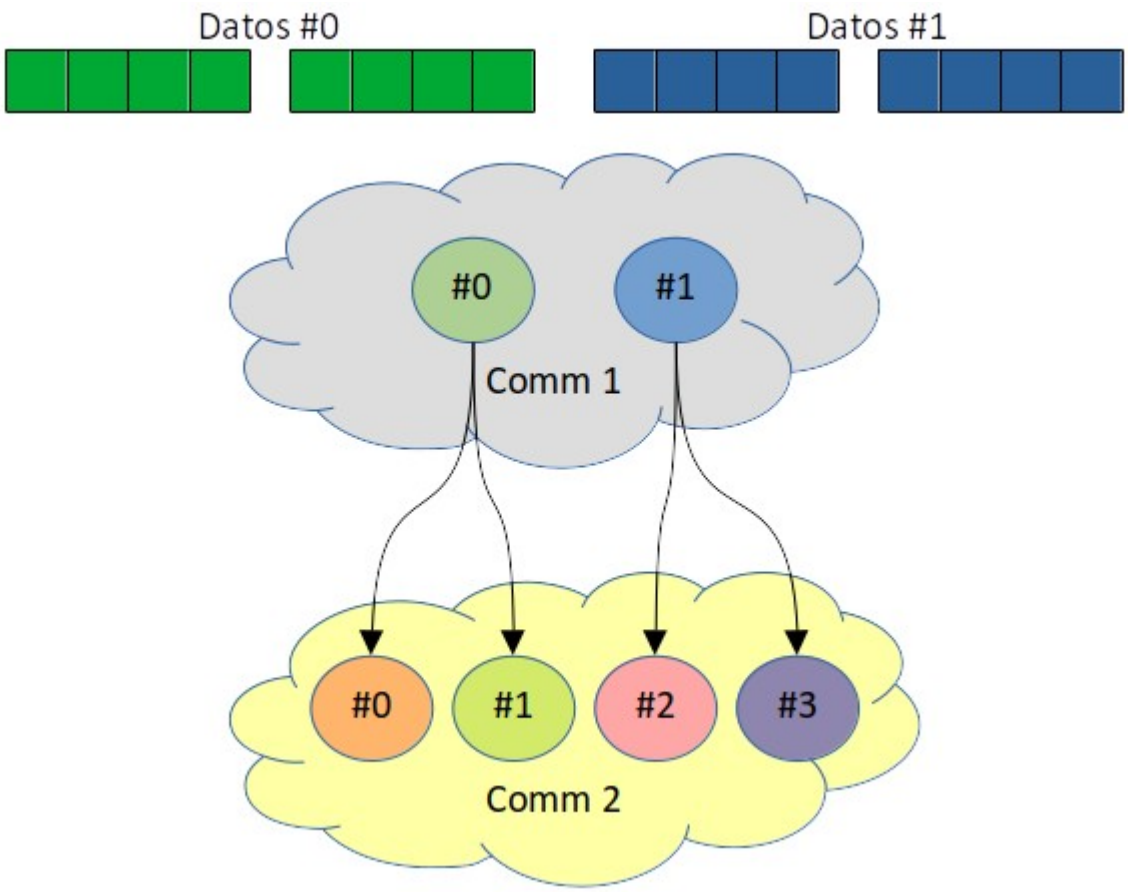

Figura 5: Redistribución de datos en DMR.

Una vez recibidos los datos por los nuevos procesos el sistema puede eliminar el grupo original. Tendríamos el sistema como mostramos en la figura 6, listo para poder continuar la ejecución.

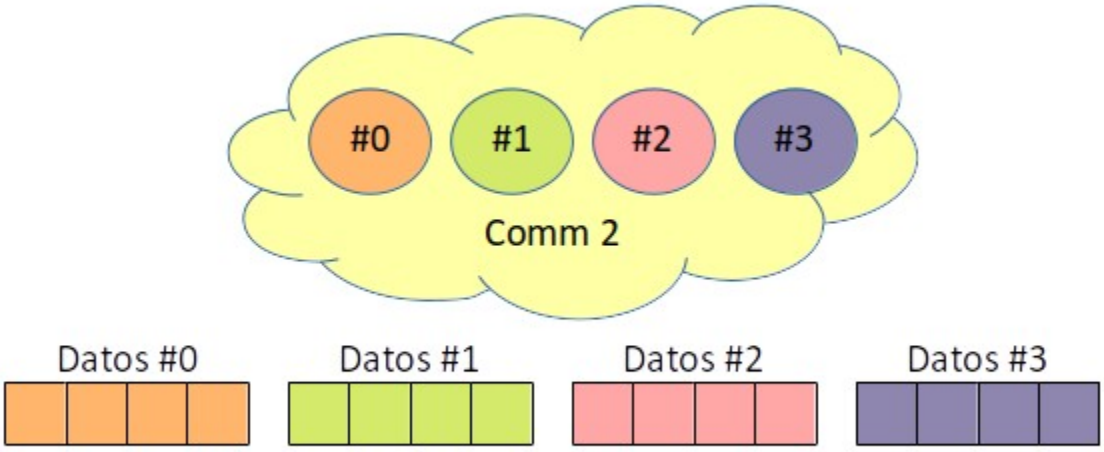

Figura 6: Reanudación de la ejecución en DMR.

Para iniciar el desarrollo de nuestra aplicación maleable es necesario disponer de determinadas bibliotecas además de un conjunto de archivos para poder realizar tanto el desarrollo como su ejecución. Lo más apropiado es seguir los pasos indicados en el tutorial DMR para el clúster Minotauro que se encuentra en[3].

Una vez finalizado el pequeño tutorial dispondremos de los archivos necesarios para realizar un test de ejecución y comprobar que el sistema funciona correctamente. Mostramos el listado de archivos que encontramos en nuestro sistema de archivos y que deberán ser modificados adecuadamente para nuestro caso. La ruta relativa de la carpeta en cuestión es *../dmr-mt/test-dmr-mt*.

```
launch.sh  Makefile  mt_submission.sbatch  mywrapper.sh.base
```

Como primer paso se ha clonado el directorio *test-dmr-mt* y renombrado para nuestro proyecto. A continuación detallaremos cada uno de estos archivos y los cambios efectuados con el fin de adaptarlo a nuestras necesidades.

- ***launch.sh***: Se trata de un guión para *Slurm* el cual contiene las instrucciones para ejecutar el archivo ejecutable con el número de nodos especificados. Lo deberemos modificar para utilizar

nuestro archivo ejecutable. Además, se ha modificado debido a problemas de comunicación entre los procesos en diferentes nodos del modo en que mostramos más adelante.

- ***Makefile***: Contiene las reglas para realizar correctamente la compilación teniendo en cuenta donde se encuentran los archivos necesarios para llevar a cabo la tarea correctamente. Se debe de modificar de tal modo que compile nuestro código fuente y no el archivo utilizado para realizar el test, *sleepOf.c*.
- ***mt_submission.sbatch***: Este es un guión para *Slurm*. En la figura 7 que se muestra a continuación podemos ver de manera esquemática las tareas realizadas. En primera instancia, se reservan los nodos especificados. Seguidamente, se inicia en uno de los nodos una nueva instancia anidada de *Slurm* con las modificaciones para poder utilizar la maleabilidad en el sistema. Por último, se ejecuta el guión *launch.sh* para lanzar nuestro archivo, inicialmente en el número de nodos especificados y aleatoriamente se modifica el número de nodos durante su ejecución. De este modo es posible visualizar en el archivo de salida como van variando su número durante el transcurso de la ejecución. En este caso no es necesario modificar ninguna línea para llevar a cabo nuestra ejecución.
- ***mywrapper.sh.base***: Este guión es lanzado por *mt_submission.sbatch*. Se encarga de iniciar adecuadamente el nuevo gestor de trabajos modificado. No es necesaria ninguna modificación para la ejecución de nuestro archivo.

Como hemos comentado anteriormente, se ha modificado la línea donde contiene la función *mpirun* para que se ejecute nuestra aplicación y para que se comuniquen los nodos correctamente del siguiente modo:

```
mpirun -iface eth0 -n $SLURM_JOB_NUM_NODES -hosts $NODELIST ./jacobi
```

El parámetro *-iface eth0* indica explícitamente que la interfaz de comunicación a utilizar es eth0. Además, puede observarse como se ha indicado al final del comando nuestra aplicación *jacobi* para ser ejecutada.

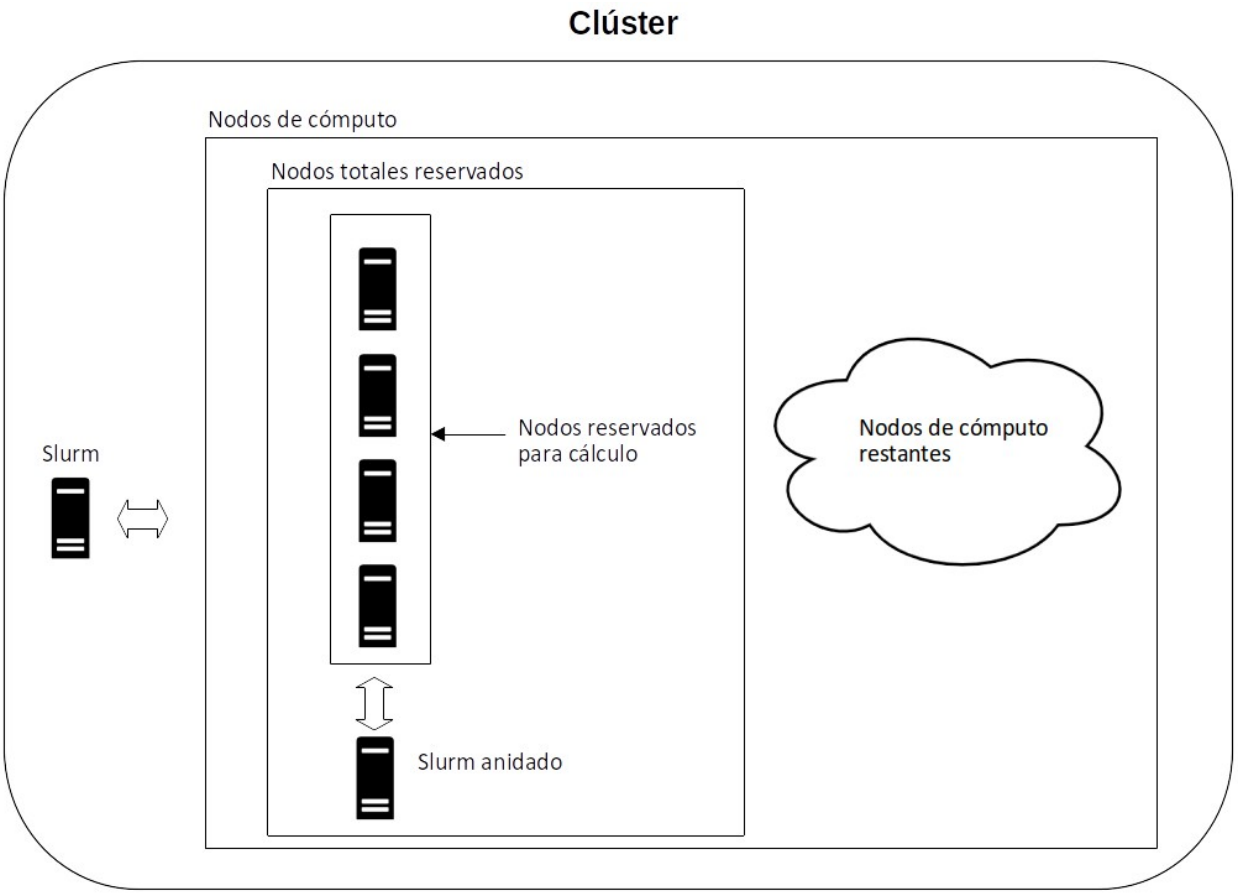

Figura 7: Esquema ejecución.

El archivo *.bash_login* habrá quedado del siguiente modo:

```
# module list for using dmr api

module purge
module load autoconf automake gcc/8.1.0 cmake/3.9.6
export PATH=/apps/dmr/libtool/bin:$PATH
export PATH=/apps/dmr/mpich/bin:$PATH
export LD_LIBRARY_PATH=/apps/dmr/mpich/lib:$LD_LIBRARY_PATH
export ACLOCAL_PATH=/usr/share/aclocal

# variable list for using dmr installation

export DMR_PATH=/gpfs/projects/bsc85/bsc85539/dmr-mt
export LD_LIBRARY_PATH=$DMR_PATH/slurm-install/lib:$LD_LIBRARY_PATH
export LD_LIBRARY_PATH=$DMR_PATH:$LD_LIBRARY_PATH
export SLURM_ROOT=$DMR_PATH/slurm-install
```

Para poder utilizar la API DMR deberemos hacer algunos ajustes a la implementación inicial del código. Se deberá crear una función donde se ejecutarán los cálculos en cada una de las iteraciones que se realicen. También se debe disponer un contador para las iteraciones que se van realizando para poder decidir con qué periodicidad será ejecutada la reconfiguración de recursos de nuestra aplicación. Por último, deberemos realizar una llamada a la función encargada de liberar los recursos. Con lo comentado, el código en C debe de refactorizarse como se muestra a continuación:

```
void main() {
  init();
  while (t < TIMESTEPS) {
    compute();
    t++;
    ------- SYNCHRONIZATION POINT -------
  }
  free();
}
```

Ya realizada la refactorización de la aplicación se realiza la implementación para la utilización de la biblioteca DMR. A continuación se muestra el código base y se explica brevemente cada una de las llamadas a las funciones de la API.

```
void main() {
  DMR_INIT(init(), recv_ex(), recv_sh());
  DMR_inhibit_time(); // (seconds), optional
  DMR_inhibit_iter(); // (iterations), optional
  DMR_Set_parameters(16, 256, 64); // (min, max, sweet spot), optional
  while (DMR_it < TIMESTEPS) {
    compute();
    DMR_it++;
    ------- SYNCHRONIZATION POINT -------
    DMR_RECONFIGURATION(send_ex(), send_sh()) ;
  }
  DMR_FINALIZE(free());
}
```

La función *DMR_INIT(init(), recv_ex(), recv_sh())* deberá contener la inicialización de las estructuras de datos de nuestra aplicación, en nuestro caso concreto se inicializan los vectores con sus valores iniciales. Además, contendrá las llamadas a las funciones encargadas de enviar los datos en caso de llevarse a cabo una reasignación de recursos, tanto sea para expandirse como para contraerse en el número de

nodos asignados.

La función *DMR_inhibit_time()* es opcional y es utilizada para especificar cada que intervalo de tiempo no se va a realizar ningún intento de reconfiguración de la aplicación. La función *DMR_inhibit_iter()* tiene el mismo objetivo, la diferencia en este caso es que en lugar de indicar un cierto intervalo de tiempo indica un número determinado de iteraciones. La función solo contiene un único argumento de tipo entero, que indica cada cuantos ciclos realizados por el iterador DMR_it es ejecutada la llamada para realizar una posible reasignación de recursos.

La función *DMR_Set_parameters(1, 4, 0, 2)* es utilizada para configurar el comportamiento de maleabilidad en nuestra aplicación. Su primer argumento indica la cantidad mínima de procesos que debe utilizar y seguidamente se indica el número máximo. El tercer valor indica el número preferente de procesos a utilizar, en el caso mostrado 0 indica no tener ningún tipo de preferencia. Por último, se indica el factor numérico en el que el número de procesos podrá variar, el indicado es 2, con lo cual el número de procesos pueden ser 1, 2, 4 y así sucesivamente teniendo el valor máximo posible según los nodos de cómputo existentes en el clúster.

El bucle *while* que se encuentra a continuación debe contener como condición de salida un número de iteraciones máximo que es controlado por la variable *DMR_it* y especificada en el encabezamiento del código a través de la cláusula #define normalmente utilizada en el lenguaje C.

La primera función que encontramos dentro del bucle es la función *compute*, que contendrá los cálculos de la aplicación, los argumentos que incluirá serán los necesarios para realizar su tarea y dependerá de cada caso. Seguidamente se incrementará el contador *DMR_it* con el fin de llevar el adecuado control en el número de iteraciones realizadas.

La última función del bucle es la utilizada para llevar a cabo la reconfiguración en el número de procesos. *DMR_RECONFIGURATION(send_ex(), send_sh())* contiene además las funciones encargadas con el envío de los datos de los procesos en ejecución hacia los nuevos procesos que se crearán en el caso de llevarse a cabo un cambio en el número de recursos empleados.

2.3 Problema utilizado para resolver mediante la biblioteca MPI

El problema a resolver emplea la ecuación de Laplace que es resuelta mediante el método Jacobi. Ambos métodos están relacionados con la resolución de ecuaciones matemáticas.

La ecuación de Laplace es una ecuación diferencial parcial de segundo orden que aparece en diversas áreas de la física y las matemáticas. Describe el comportamiento de campos escalares, como la temperatura o el potencial eléctrico, en regiones donde no hay fuentes ni sumideros. La ecuación lleva el nombre de Pierre-Simon Laplace, un matemático francés. Resolver la ecuación de Laplace implica encontrar una función que satisfaga la ecuación y las condiciones de contorno dadas.

El método de Jacobi es un método iterativo utilizado para resolver sistemas de ecuaciones lineales. También se puede aplicar para resolver la ecuación de Laplace, que es una ecuación diferencial parcial que describe la distribución en estado estacionario de la temperatura o potencial en una región. El método de Jacobi implica actualizar los valores de las incógnitas en un sistema de ecuaciones en función del promedio de sus valores vecinos. Es un método simple y ampliamente utilizado para resolver sistemas lineales y puede utilizarse como un método para resolver la ecuación de Laplace.

El problema se plantea como se muestra en la figura 8. Se parte de una placa de metal cuya temperatura, en nuestro caso es 0. A esta placa se le colocan dos fuentes de calor, una a su derecha y otra en su parte inferior. Ambas fuentes de calor tienen un valor incremental, parten de 0 en su inicio y proporcionalmente alcanzan una temperatura de 100^{o} en el extremo contrario.

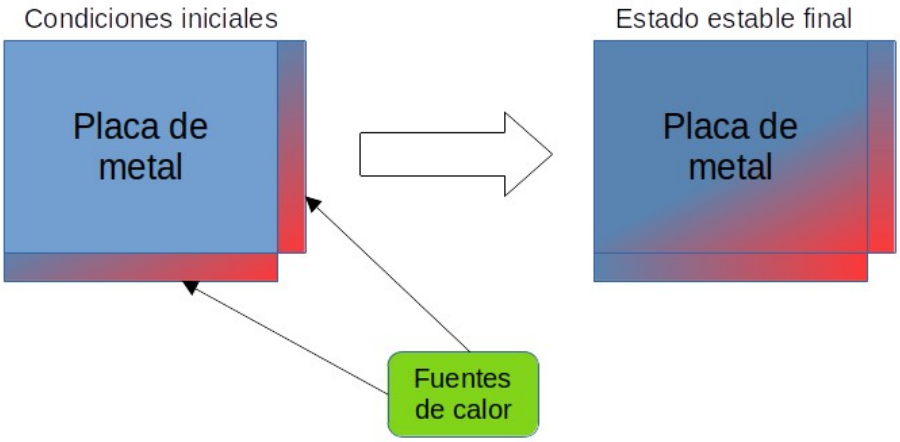

Figura 8: Esquema de placas metálicas en su estado inicial y final.

El código empleado en lenguaje C parte de la solución realizada en serie, que puede consultarse en las transparencias [1]. Partiendo de este código se modifica para ser resuelto por varios procesos mediante la biblioteca MPI [2]. Este código realiza tanto la descomposición de dominio de datos como la comunicación entre procesadores.

El código moldea la placa metálica como una matriz de un número determinado de puntos. La matriz puede ser de tamaño *MxN*, siendo ambos valores 1000 en el ejemplo por utilizado. La ecuación de Laplace establece que cada punto de la rejilla que se ha formado es el promedio de sus puntos vecinos. El método de iteración de Jacobi calcula repetidamente nuevos valores en cada punto de la rejilla en función del promedio de sus puntos vecinos hasta que se logra la convergencia, es decir, se comprueba la diferencia entre los valores de una iteración a la siguiente, y el proceso continúa hasta que la diferencia se vuelve lo suficientemente pequeña como para considerarla tolerable.

El programa puede describirse en los siguiente pasos que se describen a continuación:

1. **Inicialización**: El programa inicializa la rejilla y establece las condiciones iniciales para la distribución de temperatura.
2. **Iteración**: A continuación se entra en un bucle que calcula de manera iterativa nuevos valores para cada punto de la rejilla en función del promedio de sus puntos vecinos. Este proceso continúa hasta que se logra la convergencia o se alcanza el número máximo de iteraciones. Los pasos dentro del bucle son los siguientes:

a. **Promedio y copia**: En cada iteración, el programa realiza un promedio copiando los valores de temperatura actuales en una matriz temporal.
b. **Sincronización**: El programa sincroniza la fase de cálculo, donde se calculan los nuevos valores de temperatura, y la fase de comunicación, donde se intercambian datos entre procesadores.
c. **Comunicación**: En la fase de comunicación, el programa envía y recibe datos entre procesadores, específicamente en las celdas fantasma, que son las regiones fronterizas del subdominio de cada procesador. La función *MPI_Reduce* es la encargada de llevar a cabo el cálculo del valor máximo del diferencial de temperatura calculado individualmente por cada procesos. Una vez calculado este valor máximo es distribuido entre todos los procesos a través de una llamada a la función *MPI_Bcast* que realiza el envío del nuevo valor a todos los procesos incluido el proceso raíz que realiza el envío.
d. **Comprobación de convergencia**: El programa verifica la convergencia calculando el cambio máximo de temperatura entre las iteraciones. Si el cambio está por debajo de un umbral específico, el programa considera que la solución ha convergido.

3. **Finalización**: El programa termina cuando se logra la convergencia o se alcanza el número máximo de iteraciones.

En el código paralelo, la comunicación de datos se maneja mediante el uso de paso de mensajes. Cada procesador tiene su propia subrejilla y solo comunica los valores fronterizos con sus procesadores vecinos. Este enfoque reduce el uso de memoria y minimiza la cantidad de comunicación requerida. Los valores fronterizos se denominan comúnmente "celdas fantasma" y son necesarios para una distribución adecuada de datos entre los procesadores. En la siguiente figura se muestra como se han repartido los diferentes datos entre los procesos y el intercambio de datos de las celdas fantasmas necesarias para realizar los cálculos. El ejemplo está realizado sobre 4 procesos a modo de ejemplo.

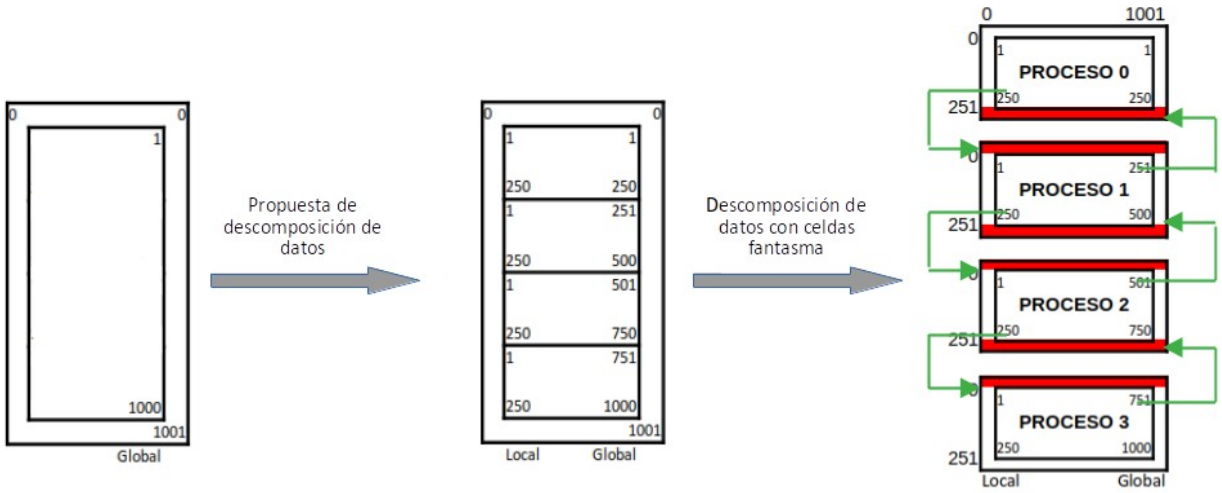

Figura 9: Esquema de distribución de datos entre procesos.

Los datos son particionados de manera vertical como se puede observar. Se realiza de este modo ya que en el lenguaje de programación C esta división proporciona una continuidad en la estructura de datos. Por contraposición tenemos que en el lenguaje Fortran esta división debería realizarse horizontalmente, porque la continuidad de datos en este lenguaje difiere en este sentido al lenguaje C.

Durante la fase de comunicación, cada procesador envía y recibe datos en las zonas fantasma, que son las regiones fronterizas del subdominio de cada procesador.

La convergencia de la solución se puede determinar calculando la diferencia entre los valores de la solución en cada punto de la rejilla de una iteración a la siguiente. En el contexto proporcionado, la solución converge en 3372 iteraciones[1]. Esto implica que el programa calcula de manera iterativa nuevos valores para cada punto de la rejilla hasta que la diferencia entre las iteraciones se vuelve lo suficientemente pequeña como para considerarla aceptable.

Para determinar la convergencia, el programa calcula el cambio máximo en la temperatura entre iteraciones. Si este cambio está por debajo de un umbral específico, el programa considera que la solución ha convergido. El valor específico del umbral no se menciona en el contexto proporcionado, pero generalmente se establece en función de la precisión deseada de la solución.

Es importante tener en cuenta que los criterios de convergencia pueden variar según el problema específico que se está resolviendo y el nivel de precisión deseado. En este caso, la convergencia se determina en función del cambio máximo en la temperatura entre iteraciones, lo que indica que la solución ha alcanzado un estado estable en el que las iteraciones adicionales no afectan significativamente a los resultados.

2.4 Detalles de las modificaciones realizadas sobre el código original

Como primer paso para adaptar el código de Jacobi para la implementación con la biblioteca DMR se han transformado las matrices en vectores. Esta operación se ha efectuado recalculando los índices del siguiente modo:

array[i][j] = vector[i(COLUMNAS+2)+j]*, siendo COLUMNAS el valor máximo del segundo índice, al que además se le añade las dos columnas extras para almacenar las celdas fantasma.

Se detalla el fragmento de código donde se realizan las declaraciones de las funciones necesarias para la implementación del código siguiendo el esquema marcado por las bibliotecas DMR para que la integración resulte más sencilla.

```
// helper routines
void initialize_data(double **data, double **data_old, int world_size, int world_rank, int size);
void track_progress(int iter, int rows, double *data);
double compute(double *data, double *data_old, int world_size, int world_rank, int size, int *iterator);
void finalize(double *data, double *data_old);
```

Como ejemplo de la modificación de las matrices a vectores podemos mostrar el siguiente fragmento de la función *compute* donde se realizan los cálculos para cada uno de los puntos en función de los valores vecinos. El código queda del siguiente modo:

```
for(i=1; i<= size; i++)
{
   for(j=1; j <= COLUMNS; j++)
```

```
  {
    data[i*(COLUMNS+2)+ j]= 0.25 * (data_old[(i+1) * (COLUMNS+2) + j] +
                          data_old[(i-1)*(COLUMNS+2) + j] +
                          data_old[i*(COLUMNS+2) + j+1] +
                          data_old[i*(COLUMNS+2) + (j-1)]);
  }
}
```

En el siguiente fragmento de código, se muestran las declaraciones de las funciones utilizadas para el envío y recepción de datos al aplicar tanto la expansión como la contracción en el número de nodos. Estas funciones son necesarias para enviar los datos de los procesos en ejecución hacia los procesos recién creados en el sistema. Los nuevos procesos variarán en número respecto a los existentes en una determinada potencia que es configurable. A su vez, las funciones de recepción son las encargadas de recoger los datos enviados y almacenarlos en las estructuras de datos correspondientes.

```
// dmr function definitions
void send_shrink(double *data_old,  int size, int iterator);
void recv_shrink(double **data, double **data_old, int *size, int *iterator);
void send_expand(double *data_old,  int size, int iterator);
void recv_expand(double **data, double **data_old, int *size, int *iterator)
```

A continuación se mostrará y comentará el código de cada una de ellas para una mejor comprensión del funcionamiento de la biblioteca DMR.

Es preciso indicar, que tras las ejecuciones inicialmente realizadas, se comprobó que el contador de iteraciones *DMR_it* no contenía el valor real de los ciclos realizados. Para poder asegurar que el número de ciclos son realmente correctos se ha decidido incluir el incremento del contador dentro de la función de cómputo. Además, se ha comprobado la necesidad de enviar el iterador cada vez que el sistema cambia de número de nodos, de otro modo su valor se perdería y adquiriendo un valor aleatorio.

```
void send_shrink(double *data_old,  int size, int iterator)
{
  int world_rank, comm_size, intercomm_size, factor, dst;
  MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);                 // get rank number
  MPI_Comm_size(MPI_COMM_WORLD, &comm_size);                  // get size of the group
  MPI_Comm_remote_size(DMR_INTERCOMM, &intercomm_size);  // get size of the new group
  factor = comm_size / intercomm_size;                        // calculate factor of resize
  dst = world_rank / factor;
```

```
  // synchronous and blocking sending to guarantee the sending and receiving of data
  MPI_Ssend(data_old, (size+2) * (COLUMNS+2), MPI_DOUBLE, dst, 0, DMR_INTERCOMM);
  if (world_rank % factor == 0)
  {
    MPI_Ssend(&iterator, 1, MPI_INT, dst, 0, DMR_INTERCOMM);
  }
}
```

La función mostrada se encarga de enviar los datos actuales a los nuevos procesos en el caso de una contracción en su número. Dado que el número de procesos va a ser menor es necesario que varios procesos envíen datos hacia un mismo proceso de los que se van a crear. El nuevo destino viene determinado por la división entera del proceso actual entre el factor de contracción que se ha calculado previamente. Además, se envía el iterador una única vez a cada uno de los nuevos procesos del programa.

Como función complementaria a la función mostrada tenemos la siguiente mostrada. Esta se encarga de recibir los datos para el nuevo número de procesos:

```
void recv_shrink(double **data, double **data_old, int *size, int *iterator)
{
  int world_rank, comm_size, parent_size, src, factor, iniPart, i, j, number_amount, data_size;
  MPI_Status status;
  double *buffer_data;
  MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &comm_size);
  MPI_Comm_remote_size(DMR_INTERCOMM, &parent_size);
  factor = parent_size / comm_size;
  (*size) = 0;
  for (i = 0; i < factor; i++)
  {
    src = world_rank * factor + i;

    MPI_Probe(src, MPI_ANY_TAG, DMR_INTERCOMM, &status);
    MPI_Get_count(&status, MPI_DOUBLE, &number_amount);

    if (i==0)
    {
      iniPart = 0;
      data_size = number_amount * (parent_size + 1);          // estimated maximum size required
      (*data) =     (double *)malloc( data_size * sizeof(double));
      (*data_old) = (double *)malloc( data_size * sizeof(double));
      buffer_data =     (double *) malloc (number_amount * 2 * sizeof(double));
```

```
    }

    MPI_Recv(buffer_data, number_amount, MPI_DOUBLE, src, 0, DMR_INTERCOMM,
MPI_STATUS_IGNORE);

    if (i==0)
    {
      MPI_Recv(iterator, 1, MPI_INT, src, 0, DMR_INTERCOMM, MPI_STATUS_IGNORE);
    }

    for (j = 0; j < number_amount; j++)
    {
      (*data_old)[j+iniPart] = buffer_data[j];
    }

    iniPart += (number_amount-2*(COLUMNS+2));
    (*size) += (number_amount/(COLUMNS+2) -2);
  }
}
```

En esta función, cada uno de los procesos debe ejecutar la función MPI para recibir datos equivalente al factor de contracción. La primera vez que se ejecuta el bucle de recepción debemos estimar aproximadamente cuantos datos contendrán los vectores que almacenan la información de la placa de metal. Esto se realiza incrementando en una unidad el número de porciones de datos necesarios para almacenar cada una de las iteraciones necesarias para recibir todos los datos. Esto es debido a que la última recepción de datos almacenará las líneas restantes de la división entera en el caso en el que el número de columnas no sea posible dividirlo entre el número de procesos de manera que el resto de su división sea cero.

La función acaba calculando el desplazamiento dentro de los vectores de datos para almacenar a partir de esa posición los nuevos datos que se reciban al iterar el bucle. Además, se suma el número de filas reales a las ya recibidas. En ambos cálculos se tiene en cuenta las filas fantasmas utilizadas y almacenadas en las estructuras de datos.

A continuación mostramos la pareja de funciones implicadas para el caso que el sistema se expanda a un número mayor de nodos de cómputo.

Como en el caso anterior, la primera función se encarga del envío de datos a los nuevos procesos, pero

en esta ocasión se envía la información a más de un proceso.

```
void send_expand(double *data_old,  int size, int iterator)
{
   int world_rank, comm_size, intercomm_size, factor, dst, iniPart;
   MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
   MPI_Comm_size(MPI_COMM_WORLD, &comm_size);
   MPI_Comm_remote_size(DMR_INTERCOMM, &intercomm_size);

   factor = intercomm_size / comm_size;
   size = GLOBAL_ROWS / intercomm_size;

   for (int i = 0; i < factor; i++)
   {
      dst = world_rank * factor + i;
      iniPart = size * (COLUMNS+2) * i;
      if (dst == intercomm_size - 1 )
      {
         size += GLOBAL_ROWS % intercomm_size;
      }
      // synchronous and blocking sending to guarantee the sending and receiving of data
      MPI_Ssend(data_old + iniPart, (size+2)*(COLUMNS+2), MPI_DOUBLE, dst, 0, DMR_INTERCOMM);
      MPI_Ssend(&iterator, 1, MPI_INT, dst, 0, DMR_INTERCOMM);

   }
}
```

Esta función es la encargada de enviar a los nuevos procesos la parte equivalente indicada por el factor de expansión de datos a los nuevos procesos. En este caso es necesario tan solo enviar parte de los datos. Por este motivo es necesario calcular el desplazamiento dentro de la estructura donde los datos son almacenados para enviarlos, siempre teniendo en cuenta las filas fantasmas que deben existir para poder realizar los cálculos adecuadamente. Al igual que en el caso de la contracción se envía el iterador a cada nuevo proceso creado.

La función complementaria al envío es la siguiente:

```
void recv_expand(double **data, double **data_old,  int *size, int *iterator)
{
   int world_rank, comm_size, parent_size, src, factor, number_amount;
   MPI_Status status;
   MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
   MPI_Comm_size(MPI_COMM_WORLD, &comm_size);
```

```
  MPI_Comm_remote_size(DMR_INTERCOMM, &parent_size);
  factor = comm_size / parent_size;
  src = world_rank / factor;

  MPI_Probe(src, MPI_ANY_TAG, DMR_INTERCOMM, &status);
  MPI_Get_count(&status, MPI_DOUBLE, &number_amount);

  (*data) =     (double *)malloc( number_amount * sizeof(double));
  (*data_old) = (double *)malloc( number_amount * sizeof(double));

  (*size) = (number_amount / (COLUMNS + 2)) -2;     // new number of rows without ghost cells

  MPI_Recv((*data_old), number_amount, MPI_DOUBLE, src, 0, DMR_INTERCOMM,
              MPI_STATUS_IGNORE);
  MPI_Recv(iterator, 1, MPI_INT, src, 0, DMR_INTERCOMM, MPI_STATUS_IGNORE);
}
```

La recepción de los datos por la función de recepción se realiza de manera directa. A través de las funciones MPI para recibir la información de los datos a recibir se preparan las nuevas estructuras de datos. Los datos son almacenados seguidamente al igual que el valor del iterador. Como en todas las funciones, se hace necesario contemplar las filas fantasmas para los cálculos de tamaño, como puede comprobarse en el código de cada una de las funciones.

Por último, el ejemplo utilizado utiliza un tamaño de placas modelado por 1000 filas y 1000 columnas. Con este tamaño la solución alcanza la convergencia tras 3372. En nuestro caso se ha modificado el tamaño de los datos aumentandolo para que nuestro programa se ejecute por un tiempo mayor. Así, con la solución serie podemos calcular en cuántas iteraciones se debe de alcanzar el valor de convergencia y de este modo se podrá comprobar que las modificaciones introducidas en nuestro código son correctas al alcanzar la convergencia en el mismo número de iteraciones.

Además del uso de MPI para comunicarse los procesos entre los diferentes nodos de cómputo se ha adaptado el código para el uso del multiproceso mediante las librerías openMP. En caso de no utilizarlas, solo se estaría aprovechando parte de la capacidad de cómputo de cada uno de los nodos. A través de esta librería es posible utilizar más de un procesador de los disponibles en cada uno de los nodos. Aunque la cantidad de CPU por nodo es de 16, será preciso realizar algunos análisis con el fin de determinar el número apropiado de hilos por nodo. Bajo determinadas circunstancias, no siempre se obtiene un mejor rendimiento al utilizar toda la capacidad de cálculo, ya que entra en juego la

disponibilidad de datos en memoria. Así, sobrepasando cierta cantidad puede ser necesario una espera mayor al producirse una mayor cantidad de movimientos entre la memoria del sistema y la memoria interna de las CPUs.

Mediante pruebas sobre el código en su versión serie se han ido añadiendo las instrucciones en los diferentes bucles que es donde podemos utilizar el multihilo y así se ha podido determinar cuales de ellas aumentan el rendimiento del programa. Se ha determinado que en dos de los bucles el multihilo sí proporciona un mayor rendimiento de la aplicación.

El primer fragmento donde se ha utilizado el multihilo es en la función *compute*, en su bucle de cálculo. Podemos ver esta modificación a continuación:

```
// main calculation: average of my four neighbors
#pragma omp parallel for private(j)
for(i=1; i<= size; i++)
{
  for(j=1; j <= COLUMNS; j++)
    {
       data[i*(COLUMNS+2)+ j] = 0.25 * (data_old[(i+1) * (COLUMNS+2) + j] +
                                        data_old[(i-1)*(COLUMNS+2) + j] +
                                        data_old[i*(COLUMNS+2) + j+1] +
                                        data_old[i*(COLUMNS+2) + (j-1)]);
    }
}
```

El otro fragmento de código donde se ha empleado esta técnica también pertenece a la función *compute*. En esta ocasión se emplea en el bucle encargado para encontrar la diferencia de temperatura mayor. Se muestra el fragmento modificado:

```
#pragma omp parallel for reduction(max:dt) private(j,incr)
for(i=1; i <= size; i++)
{
  for(j=1; j <= COLUMNS; j++)
  {
    // using ternary operators improves the execution time a little
    incr = data[i*(COLUMNS+2)+ j] - data_old[i*(COLUMNS+2)+ j] ;
    incr = (incr > 0) ? incr : incr * (-1);
    dt = (incr > dt) ? incr : dt;
    data_old[i*(COLUMNS+2)+ j]  = data[i*(COLUMNS+2)+ j] ;
```

```
  }
}
```

En este último fragmento también podemos ver una última modificación introducida en el código original. La función encargada de devolver el mayor de dos números dados se ha modificado de modo que se utilizan operadores ternarios disponibles en C. De este modo la aplicación mejora su rendimiento al no realizar llamadas a ninguna función durante la ejecución del bucle.

3 Pruebas y análisis realizados

3.1 Análisis de escalabilidad de multihilo openMP

Se ha realizado un análisis de escalabilidad para comprobar cómo se comporta la aplicación en un nodo variando el número de hilos empleados. El cambio en el número de hilos por openMP se realiza fácilmente cambiando el valor de la variable que lo controla a través del guión que utilizamos para lanzar los trabajos, *launch.sh.* Sólo es necesario añadir la siguiente línea y variar el número de hilos.

```
export OMP_NUM_THREADS=4
```

Los resultados de las ejecuciones con diferentes números de hilos los mostramos en la siguiente tabla:

Número de hilos	Tiempo de ejecución (s)
1	1985
2	756
4	481
8	507
16	681

Tabla 2: Tiempos de ejecución con diferentes números de hilos.

Se ha confeccionado la siguiente gráfica donde se puede apreciar de manera visual los resultados obtenidos en la tabla anterior:

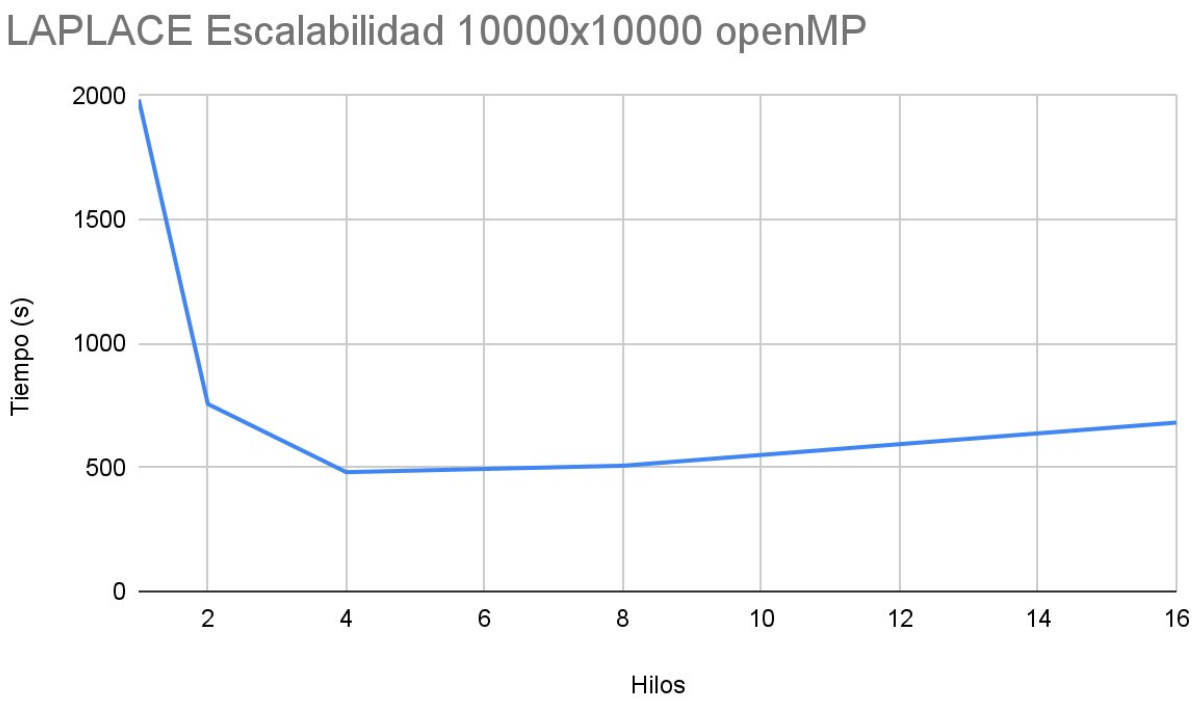

Figura 10: Gráfico con las diferentes configuraciones en multihilo de la aplicación.

Con los datos obtenidos de las ejecuciones en multihilo se puede observar que aumentar la cantidad de estos en más de 4 no comporta una mejora de rendimiento sino que puede verse como el tiempo de ejecución aumenta ligeramente a medida que superamos el número óptimo de cuatro hilos. Por lo tanto, en el análisis de escalabilidad que ejecutaremos para comprobar el comportamiento con diferentes nodos lo más aproximado es realizarlo utilizando 4 hilos.

Este comportamiento es debido tanto a la naturaleza del problema como por las características del sistema.

3.2 Análisis de escalabilidad por nodos de ejecución con MPI

Se ha realizado un análisis de escalabilidad para comprobar cómo disminuyen los tiempos de ejecución al ejecutar nuestro desarrollo sobre un número de nodos determinado. Se han escogido potencias de 2 para realizar las pruebas. Para poder apreciar mejor la mejora de rendimiento el problema se ha incrementado de tamaño pasando ahora a tener 10.000 filas y 10.000 columnas.

Con los datos de ejecución se ha calculado el valor de la eficiencia en paralelo del siguiente modo:

$$Eficiencia(N) = \frac{T(1)}{P \cdot T(N)}$$

Se han obtenido los siguientes tiempos de ejecución con sus correspondientes valores de eficiencia:

Nodos	Tiempo de ejecución (s)	Eficiencia
1	634	1
2	350	0.91
4	191	0.83
8	113	0.70
16	77	0.51

Tabla 3: Cálculos de eficiencia según número de hilos.

Con los datos obtenidos se ha creado el siguiente gráfico donde se aprecia mejor la escalabilidad del sistema:

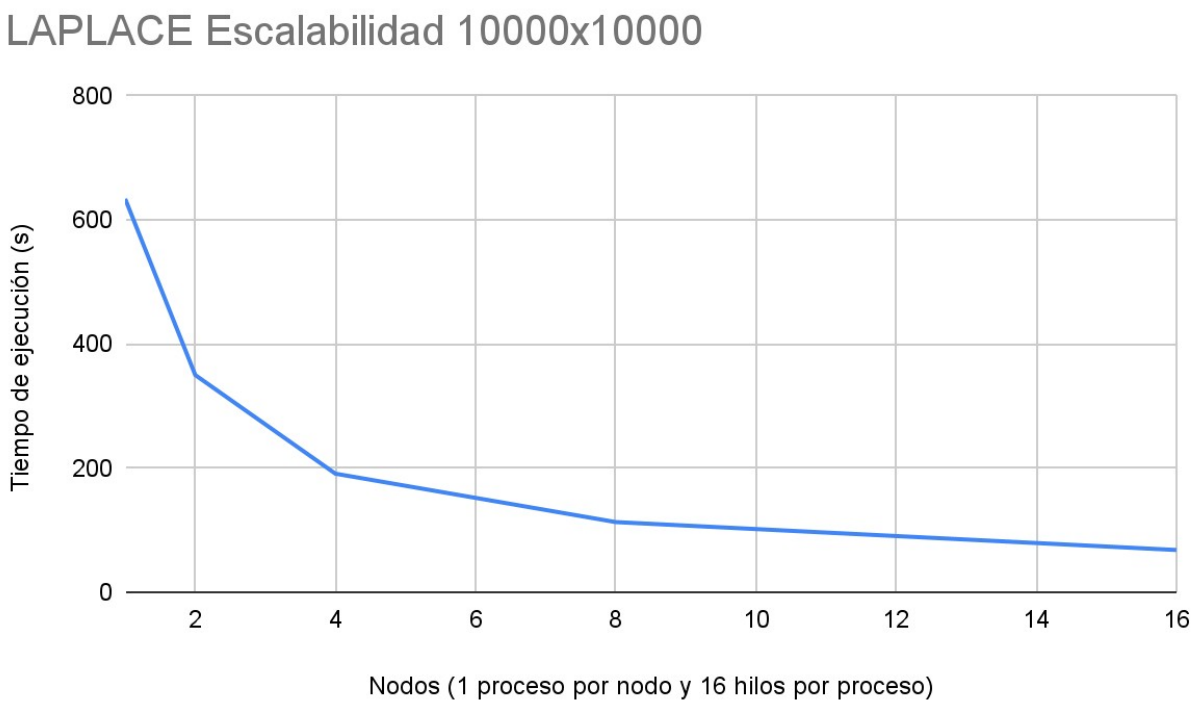

Figura 11: Gráfico de resultados con diferentes números de nodos.

3.3 Análisis teórico de eficiencia de trabajos finalizados por unidad de tiempo

Para el presente análisis teórico partimos de una configuración de 16 nodos. De este modo no existirá ningún nodo ocioso se use o no la maleabilidad en la aplicación. Los trabajos lanzados son de matrices cuadradas de tamaño 10.000 y se lanzan un total de 10 trabajos simultáneos.

En el caso de no utilizar maleabilidad, cada uno de los trabajos lanzados ocupan la totalidad de los 16 nodos de cómputo, con lo que no será posible iniciar un nuevo trabajo hasta haber finalizado el anterior. Los trabajos quedarían distribuidos del siguiente modo:

N1	N2	N3	N4	N5	N6	N7	N8	N9	N10	N11	N12	N13	N14	N15	N16	T (s)
T1	T1	T1	T1	T1	T1	T1	T1	T1	T1	T1	T1	T1	T1	T1	T1	77
T2	T2	T2	T2	T2	T2	T2	T2	T2	T2	T2	T2	T2	T2	T2	T2	77
T3	T3	T3	T3	T3	T3	T3	T3	T3	T3	T3	T3	T3	T3	T3	T3	77
T4	T4	T4	T4	T4	T4	T4	T4	T4	T4	T4	T4	T4	T4	T4	T4	77
T5	T5	T5	T5	T5	T5	T5	T5	T5	T5	T5	T5	T5	T5	T5	T5	77
T6	T6	T6	T6	T6	T6	T6	T6	T6	T6	T6	T6	T6	T6	T6	T6	77
T7	T7	T7	T7	T7	T7	T7	T7	T7	T7	T7	T7	T7	T7	T7	T7	77
T8	T8	T8	T8	T8	T8	T8	T8	T8	T8	T8	T8	T8	T8	T8	T8	77
T9	T9	T9	T9	T9	T9	T9	T9	T9	T9	T9	T9	T9	T9	T9	T9	77
T10	T10	T10	T10	T10	T10	T10	T10	T10	T10	T10	T10	T10	T10	T10	T10	77

Tabla 4: Cola de trabajos sin maleablidad.

En este primer caso podemos apreciar rápidamente que todos los trabajos en la cola tendrán una duración estimada de 770 segundos (tiempos calculados en el apartado anterior), ya que cada uno de ellos tiene una duración aproximada de 77 segundos.

La siguiente tabla mostraría la ejecución con maleabilidad con una cantidad preferente de nodos de computación de 4 nodos. Así, el sistema acomodaría los trabajos existentes en la cola ocupando 4 nodos cada uno de ellos hasta ocupar la totalidad de los nodos de computación. Una vez finalizados los 8 primeros trabajos de este modo podría computar los dos trabajos restantes ocupando 8 nodos de computación cada uno de ellos, con lo que tendríamos un tiempo total de cómputo para los 10 trabajos de 495 segundos (tiempos extraídos del anterior apartado según la cantidad de nodos utilizados).

N1	N2	N3	N4	N5	N6	N7	N8	N9	N10	N11	N12	N13	N14	N16	N16	T (s)
T1	T1	T1	T1	T2	T2	T2	T2	T3	T3	T3	T3	T4	T4	T4	T4	191
T5	T5	T5	T5	T6	T6	T6	T6	T7	T7	T7	T7	T8	T8	T8	T8	191
T9	T9	T9	T9	T9	T9	T9	T9	T10	T10	T10	T10	T10	T10	T10	T10	113

Tabla 5: Cola de trabajos con maleabilidad.

Por lo tanto podemos calcular con ambos valores de tiempos la mejora en la eficiencia de trabajos finalizados por unidad de tiempo.

$$Mejora(\%) = \frac{T_{con\,maleabilidad}}{T_{\text{sin}\,maleabilidad}} \times 100 = 65\,\%$$

3.4 Análisis de resultados en el clúster Minotauro

Las pruebas reales sobre el clúster del BSC Minotauro han consistido en dos tipos distintos de lanzamientos de trabajos. Para ambos casos se han reservado 21 nodos de los disponibles en el clúster, en uno de ellos se instala el gestor de trabajos *Slurm* modificado para gestionar la maleabilidad de las aplicaciones y el resto de nodos se utilizan como nodos de cómputo.

Cada uno de los experimentos ha consistido en enviar 10 trabajos, todos ellos idénticos, y separando los envíos al gestor de trabajos en 5 segundos. Igualmente en todos los casos, tanto sin maleabilidad como con esta activada, se han enviado cada uno de ellos para que se ejecute sobre dos nodos inicialmente. En el caso de no utilizar maleabilidad se ha comprobado como el número de nodos utilizados por cada trabajo no varía. Cuando se han repetido los lanzamientos con maleabilidad, se ha podido comprobar que el envío de cada trabajo se realizaba sobre 2 nodos pero cambiaba en número expandiéndose a más nodos cuando habían disponibles.

Los tiempos se han medido desde el guión de lanzamiento de trabajos. El archivo modificado ha sido *mt_submission.sbatch*. Se muestra el fragmento modificado para realizar la toma de tiempos iniciales y finales junto con el cálculo para mostrar el tiempo total. Además, puede observarse cómo se han añadido las pausas entre trabajos de 5 segundos. Los primeros 30 segundos iniciales son utilizados para poder tener el tiempo suficiente para acceder al nodo donde se ejecuta el gestor de trabajos anidado y poder visualizar y grabar la evolución de los trabajos en los nodos utilizados para computación.

Los lanzamientos con maleabilidad se han configurado en la función DMR para ello con los siguientes parámetros de configuración:

```
DMR_Set_parameters(2, 16, 0, 2);
```

De los parámetros mostrados se puede ver como el número de nodos mínimo a utilizar por la aplicación es de dos nodos y el número máximo ha sido de 16 nodos. No se ha utilizado el tercer parámetro que se emplea para indicar el número preferente de nodos. El cuarto parámetro indica que el número de nodos debe cambiar en potencias de 2.

```
sleep 30

#Save time value into a variable
start_time=$(date +%s)

#job #1
$SLURM_BIN/sbatch -Jdmr_jacobi -N2 launch.sh &
sleep 5
#job #2
$SLURM_BIN/sbatch -Jdmr_jacobi -N2 launch.sh &
sleep 5
#job #3
$SLURM_BIN/sbatch -Jdmr_jacobi -N2 launch.sh &
sleep 5
#job #4
$SLURM_BIN/sbatch -Jdmr_jacobi -N2 launch.sh &
sleep 5
#job #5
$SLURM_BIN/sbatch -Jdmr_jacobi -N2 launch.sh &
sleep 5
#job #6
$SLURM_BIN/sbatch -Jdmr_jacobi -N2 launch.sh &
sleep 5
#job #7
$SLURM_BIN/sbatch -Jdmr_jacobi -N2 launch.sh &
sleep 5
#job #8
$SLURM_BIN/sbatch -Jdmr_jacobi -N2 launch.sh &
sleep 5
#job #9
$SLURM_BIN/sbatch -Jdmr_jacobi -N2 launch.sh &
sleep 5
#job #10
```

```
$SLURM_BIN/sbatch -Jdmr_jacobi -N2 launch.sh &

$SLURM_BIN/sinfo

aux=$( $SLURM_BIN/squeue | wc -l );
while [ $aux -gt 1 ]; do
        aux=$( $SLURM_BIN/squeue | wc -l );
        echo "$aux jobs remaining...";
        sleep 10;
done

# Calculate elapsed time
end_time=$(date +%s)
elapsed_time=$((end_time - start_time))

# Print execution time
echo "Elapsed time: $elapsed_time seconds"
```

Cada serie de lanzamientos se ha repetido tres veces para tener una mejor referencia de tiempos, en la siguiente tabla se muestran los resultados de los experimentos así como los tiempos medios calculados:

Configuración	Experimento 1	Experimento 2	Experimento 3	Tiempo medio
Sin maleabilidad	446	446	456	449
Con maleabilidad	425	416	406	415

Tabla 6: Tiempos medios con y sin maleabilidad.

Además, se ha confeccionado el siguiente gráfico con los tiempos medios calculados donde puede verse de manera visual la mejora de rendimiento.

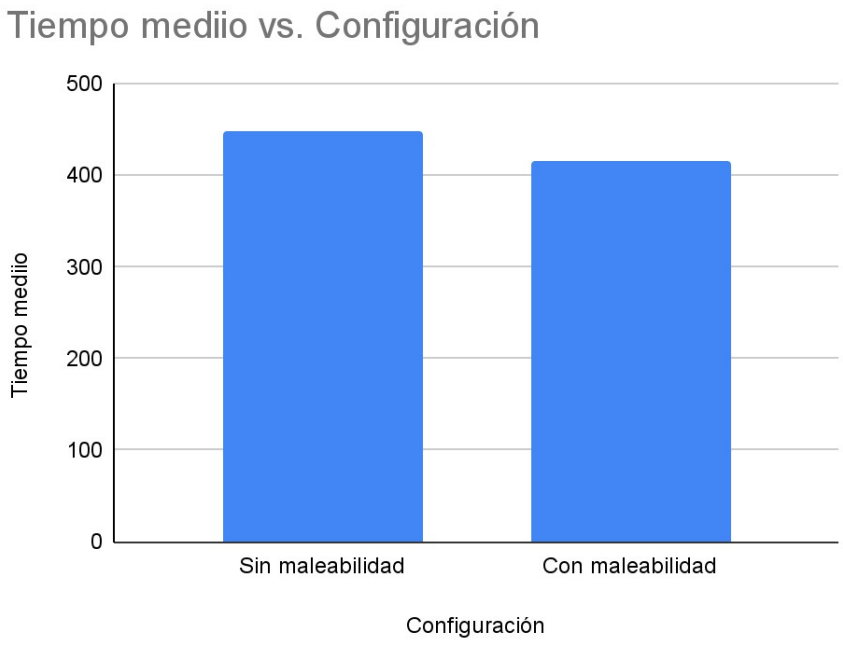

Figura 12: Gráfico de tiempos medios de las ejecuciones de las colas sin y con maleabilidad.

Podemos calcular el porcentaje de la mejora obtenida al utilizar maleabilidad, la fórmula utilizada es la siguiente:

$$Mejora(\%) = \frac{T_{\text{sin } maleabilidad} - T_{con\,maleabilidad}}{T_{con\,maleabilidad}} \times 100 = 8\%$$

Como se puede ver por el resultado obtenido, se ha obtenido una mejora de tiempos del 8% al utilizar maleabilidad. Estos resultados reflejan que el uso de maleabilidad en aplicaciones HPC permite un mejor aprovechamiento de la infraestructura, aumentando la cantidad de trabajos finalizados por unidad de tiempo.

4. Conclusiones y trabajos futuros

4.1 Conclusiones

La aplicación de la maleabilidad en las aplicaciones que se ejecutan en entornos HPC es una herramienta que posibilita un mejor aprovechamiento de las infraestructuras de cálculo como ha podido comprobarse.

Como se ha indicado anteriormente, sería interesante implementar a las bibliotecas la funcionalidad necesaria para ocultar la mayor parte de los detalles tanto en la contracción como en la expansión de los nodos de computación en los que se ejecuta.

Como futuros trabajos a realizar sería interesante realizar la puesta en producción de un clúster HPC con una versión de *Slurm* con capacidad de maleabilidad. De este modo sería posible realizar ejecuciones directamente con el gestor de trabajos modificado, en lugar del escenario anidado con el que se ha realizado el presente proyecto.

A nivel personal, el presente trabajo me ha permitido profundizar en el uso de las bibliotecas ampliamente utilizadas en entornos HPC para el uso eficiente de estas, como son las bibliotecas openMPI y openMP.

Igualmente, me ha permitido familiarizarme en el uso de un supercomputador para el desarrollo de aplicaciones científicas como una herramienta más de uso habitual.

4.2 Futuras mejoras

Como se ha podido comprobar, la implementación de nuestro problema utilizando las bibliotecas DMR ha pasado por aproximarse lo máximo posible a la implementación inicial planteada. A partir de ahí las

mayores problemáticas que se han encontrado han sido en las funcionalidades para la contracción y expansión en los nodos de computación.

Otro de los puntos que podrían mejorarse en la biblioteca DMR es la elección del número de nodos en el momento de lanzar los trabajos. Tal y como está implementado, el número de nodos iniciales es a elección del programador. Este hecho provoca que cuando existe un número de nodos que por la configuración de la maleabilidad de la aplicación podría ejecutarse pero el número de nodos especificado en la instrucción *mpirun* es superior al número de nodos libres, el trabajo no pueda lanzarse cuando si podría ejecutarse con normalidad. Esto podría solventarse si el gestor de trabajos pudiese elegir el número de nodos iniciales siempre que la configuración de la maleabilidad de la aplicación lo permitiese, de este modo podría lanzarse sin problemas si existiese un número de nodos libres en el clúster de computación.

Para abordar la implementación con mayores facilidades, sería deseable implementar algunas funcionalidades para ocultar los detalles en la implementación y realizarlo a través de plantillas o métodos de mayor abstracción para facilitar el uso de la biblioteca.

5. Bibliografía

[1] https://www.psc.edu/wp-content/uploads/2022/05/MPI-Laplace-Exercise-Intro.pdf

[2] http://rcs.bu.edu/examples/mpi/basic/laplace_mpi.c

[3] https://siserte.atlassian.net/wiki/spaces/DMR/pages/11010058/Minotauro

[4] https://www.archer2.ac.uk/training/materials/

[5] https://www.bsc.es/

[6] https://www.openmp.org/

[7] **Iserte, S.; Catalán, S.; Carratalá, R.; López, S.** (2021). *Construya su propio superordenador con Raspberry Pi.* Barcelona: Marcombo.

[8] **S. Iserte and K. Rojek**, "A Study of the Effect of Process Malleability in the Energy Efficiency on GPU-based Clusters", Journal of Supercomputing (76), pp. 255–274, Oct. 2020. ISSN: 0920-8542. https://doi.org/10.1007/s11227-019-03034-x

[9] **S. Iserte, R. Mayo, E. S. Quintana-Ortí, and A. J. Peña**, "DMRlib: Easy-coding and Efficient Resource Management for Job Malleability", IEEE Transactions on Computers (70), pp. 1443 - 1457, Sep. 2020. ISSN: 0018-9340. https://doi.org/10.1109/TC.2020.3022933

[10] **S. Iserte, H. Martínez, S. Barrachina, M. Castillo, R. Mayo, and A. J. Peña**, "Dynamic Reconfiguration of Non-iterative Scientific Applications: A Case Study with HPG-aligner", International Journal of High Performance Computing Application (33), pp. 1-10, Aug. 2018. ISSN: 1094-3420. https://doi.org/10.1177/1094342018802347

[11] **S. Iserte, R. Mayo, E. S. Quintana-Ortí, V. Beltran, and A. J. Peña,** "DMR API: Improving Cluster Productivity by Turning Applications into Malleable", Parallel Computing (78), pp. 54-66, July 2018. ISSN: 0167-8191. https://doi.org/10.1016/j.parco.2018.07.006

[12] https://github.com/jreinaleon/jacobi-dmr-example

6. Agradecimientos

Debo agradecer al Barcelona *Supercomputing Center* por haberme permitido hacer uso del clúster Minotauro para desarrollar el presente TFM. Gracias a ello, he podido familiarizarme en el uso de un sistema HPC real y me ha facilitado parte del trabajo al poner a disposición tanto la infraestructura como el software necesario.

Igualmente, he de agradecer a todo el equipo de soporte del BSC por la ayuda prestada cuando ha sido necesaria, solucionando los problemas que han ido surgiendo y respondiendo a mis peticiones.

Finalmente, agradecer a mi tutor de proyecto Sergio Iserte por la ayuda prestada durante la realización del proyecto y el seguimiento realizado mediante nuestras reuniones virtuales quincenales y los correos electrónicos intercambiados.