El camino desde la maleabilidad MPI hasta las cargas de trabajos adaptativas

Sergio Iserte[1], Rafael Mayo[1], Enrique S. Quintana-Ortí[1],
Vicenç Beltran[2] y Antonio J. Peña[2]

Resumen— **En este artículo hacemos un repaso de las técnicas de reconfiguración dinámica que se han desarrollado en los últimos años. Estas técnicas se utilizan sobre aplicaciones flexibles, es decir, que están diseñadas e implementadas para permitir variar el número de procesos que utilizan durante su ejecución. Cuando nos encontramos en un sistema distribuido, migrar la ejecución de unos procesos a otros implica transferir datos entre ellos. Esta transferencia se puede llevar a cabo bien con mecanismos de Checkpoint/Restart, en los cuales los procesos guardan y cargan los datos en/de memoria y/o disco; o bien con redistribución dinámica, donde los propios procesos se comunican para transferir los datos. Así pues, un conjunto de aplicaciones flexibles se convierte en una carga de trabajos adaptativa que, bien administrada, conlleva una mejor utilización de los recursos, a la vez que un incremento en la productividad del sistema.**

Palabras clave— **Estado del arte; Maleabilidad MPI; Carga de trabajos adaptativa;**

I. Introducción

En instalaciones de computación de altas prestaciones (HPC, del inglés *High Perfomance Computing*), las aplicaciones se ejecutan en computadoras compartidas donde cientos o miles de aplicaciones compiten por los mismos recursos. Las aplicaciones son enviadas al sistema en forma de trabajos, los cuales conforman la carga de trabajo del sistema. Así pues, adaptar la carga de trabajo a los recursos disponibles puede mejorar considerablemente la utilización de recursos y la productividad global del sistema.

Adaptar la carga de trabajo en tiempo de ejecución, no solo beneficia a los administradores de sistemas, si no también a los usuarios finales. Mientras que los administradores buscan aumentar el ratio de productividad a la vez que aprovechar al máximo los recursos disponibles, los usuarios son los beneficiarios directos de la adaptación reescalable ya que contarán con restricciones menos estrictas en los recursos a la hora de enviar sus trabajos. Aunque esta situación pueda significar una ejecución más lenta del trabajo, el *tiempo total* de realización (*tiempo de espera* más *tiempo de ejecución*) compensará este retraso.

Sin embargo, para poder adaptar la carga al estado del sistema, los trabajos deberán mostrar su predisposición a "cambiar su tamaño", es decir, a ser reconfigurados dinámicamente. A priori, las aplicaciones SPMD (en inglés, *Single Program Multiple Data*) iterativas se presentan como las más adecuadas a ser reconfiguradas, ya que cada iteración representa un importante punto de sincronismo en la ejecución.

La gran mayoría de las aplicaciones científicas que son ejecutadas en clústers HPC utilizan el sistema de paso de mensajes MPI (del inglés *Message Passing Interface*) [1]. Además, la *maleabilidad MPI* permite que un trabajo pueda modificar su número de procesos durante su ejecución. Esta funcionalidad implica que los procesos iniciales mueran para dejar paso a los nuevos procesos creados, no sin antes haber transferido todos los datos, para que se puede continuar con la ejecución del trabajo.

Aparte de la *maleabilidad MPI*, que puede expandir o contraer un trabajo favoreciendo a una mejor utilización de recursos, también encontramos la *migración MPI*. Aunque también pueda ser utilizada para reducir la fragmentación en el uso de recursos, con la migración se pueden implementar sistemas de resiliencia de errores.

Así pues, estamos hablando de un tema de verdadero interés, en el qué se están dedicando muchos esfuerzos. Este artículo prentende reunir todos estos esfuerzos para ver en que estado nos encontramos.

El resto del artículo está estructurado del siguiente modo: En la Sección II, se describe como clasificar el tipo de trabajos. En la Sección III, se analizan las distintas soluciones que se han publicado en cuanto a la reconfiguración. Finalmente, la Sección IV presenta las conclusiones sobre la reconfiguración y establece cuáles podrían ser las futuras líneas de trabajo.

II. Clasificación de aplicaciones

En 1996, D. Feitelson y L.Rudolph [2] presentan una clasificación de trabajos paralelos (aplicaciones) basada en *quién* y *cuándo* se especifica el número de procesos utilizados. En esta clasificación se establecen 4 tipos de trabajos (ver Tabla I) descritos a continuación:

- **Trabajos rígidos:** trabajos que necesitan un determinado número de procesos.
- **Trabajos moldeables:** trabajos que pueden ser ejecutados utilizando un rango de procesos. A diferencia de los *rígidos*, estos trabajos admiten distintos números de procesos.
- **Trabajos maleables:** trabajos que pueden continuar su ejecución con diferente número de procesos.
- **Trabajos evolutivos:** trabajos que cambian el número de procesos durante su ejecución. A diferencia de los *maleables*, estos trabajos tienen definido en su propio código el lugar dónde re-

[1]Universitat Jaume I (UJI) - Castelló de la plana, e-mails: {siserte, mayo, quintana}@uji.es.
[2]Barcelona Supercomputing Center (BSC), e-mails: {vbeltran, antonio.pena}@bsc.es.

TABLA I
CLASIFICACIÓN DE TRABAJOS BASADA EN CÓMO SE ESPECIFICA EL NÚMERO DE PROCESOS [2].

Quién decide	Cuándo se decide...	
	...al lanzar	...en tiempo de ejecución
Usuario	Rígido	Evolutivo
Sistema	Moldeable	Maleable

configurar el número de procesos. Estos trabajos suelen contar con distintas fases de procesamiento en las cuales se necesita un determinado tipo de recurso y/o una cantidad concreta de éstos.

Esta clasificación la podemos condensar en dos categorías, dependiendo de la capacidad de los trabajos para ser reconfigurados (en términos del número de procesos en ejecución) una vez se han lanzado, es decir, en tiempo de ejecución. Así pues, denominaremos trabajos **fijos** a los trabajos cuyo tamaño permanece inalteble durante su ejecución (aplicaciones *rígidas* y *moldeables*). Por el contrario, serán trabajos **flexibles** los que puedan ser redimensionados (en términos de número de procesos) *al vuelo*, permitiendo distintas cantidades de procesos en diferentes partes de la ejecución (aplicaciones *maleables* y *evolutivas*).

III. Metodologías de reconfiguración dinámica

La reconfiguración de trabajos en tiempo de ejecución es un tema que que lleva años generando investigación. En esta sección se hará un repaso de las metodologías de reconfiguración más actuales.

Los primeros pasos hacia la reconfiguración de trabajos se basaron en simulaciones y estudios teóricos como los llevados a cabo en [3], [4], [5]. Aunque también se realizaron experimentos en [6], [7], éstos se basaron únicamente en sistemas de memoria compartida y reconfiguración intranodo.

En [8], por primera vez se presentó un análisis experimental de las cargas de trabajos adaptativas para sistemas distribuidos. En este trabajo se diseñaron tres políticas de reconfiguración de trabajos, implementadas en un sistema Intel Paragon [9], donde se procesaron varias cargas de trabajos para comprobar sus resultados. De estas tres políticas, la primera no reconfigura trabajos *al vuelo* ya que no utiliza métodos de interrupción de trabajos, mientras que las otras dos sí lo hacen. A continuación, se explican las tres políticas implementadas:

- **Planificación adaptativa**: método dirigido a trabajos moldeables, en el que dos trabajos iguales pueden tener asignados un número distinto de recursos, dependiendo del estado del sistema. En esta política el planificador calcula un "objetivo" de procesadores libres siguiendo la ecuación:

$$\text{máx}(1, \frac{\textit{Procesadores en el sistema}}{\textit{Trabajos en espera}}) \quad (1)$$

  Hasta que el número de procesadores libres no llegue al "objetivo", el planificador no ejecuta ningún trabajo. Al cumplirse el "objetivo", el planificador asigna la cantidad "objetivo" a los trabajos que puedan utilizar estos recursos para iniciar su ejecución. Una vez no se puedan planificar más trabajos por falta de recursos, el "objetivo" se recalcula. Cuantos más trabajos encolados hayan, menor será el "objetivo" y más trabajos podrán ser ejecutados (con menos recursos).
- ***Equipartitioning***: esta política trata de igualar la asignación de recursos a todos los trabajos. Así pues, cuando un nuevo trabajo llega a la cola, todos los trabajos en ejecución son interrumpidos para calcular la nueva cantidad de recursos a asignar. Esta cantidad se obtiene al aplicar la ecuación:

$$\text{máx}(1, \frac{\textit{Procesadores en el sistema}}{\textit{Trabajos listos}}) \quad (2)$$

  donde *Trabajos listos* es la suma de trabajos interrumpidos más los recién llegados a la cola. Aunque puede darse el caso que no todos los trabajos puedan ser ejecutados, obligatoriamente los trabajos interrumpidos serán reanudados. Las principales desventajas de esta política son: las interrupciones suponen un alto coste temporal; y los retrasos en la sincronización debido a que la ejecución de cada trabajo puede llegar al punto de sincronización en un momento distinto.
- ***Folding***: para evitar las desventajas del *equipartitioning* se desarrolló *folding*, que a lo sumo interrumpiría sólo un trabajo. Así pues, cuando un nuevo trabajo llega a la cola, el sistema comprueba si hay recursos libres. En caso afirmativo, se los asigna todos al nuevo trabajo, sin interrumpir ningún trabajo. Si no hay suficientes recursos, el sistema interrumpe el trabajo con más recursos asignados y le reasigna la mitad, para asignar la otra mitad al nuevo trabajo (siempre cuando sea posible). Por el contrario, si hay recursos libres y no hay trabajos en espera, éstos le serán reasignados al trabajo con menos recursos en el sistema, denominándose esta operación como *unfolding*.

Esta primera publicación sobre cargas adaptativas concluye que las mejoras en el tiempo global vienen dadas, principalmente, por la reducción en el tiempo de espera de los trabajos (aunque en ciertos casos el tiempo de ejecución individual aumenta). Los mejores tiempos de procesado de cargas de trabajos se obtienen con la política de *folding*, ya que permite ejecutar trabajos encolados sin tener inicialmente recursos libres y ofrece menor sobrecoste que el resto de políticas de interrupción.

La solución presentada, a pesar de ser muy rudimentaria ya que el usuario debería encargarse de todo el proceso de reconfiguración y redistribución de datos, abrió el camino al estudio de nuevas metodologías para la reconfiguración de trabajos y adap-

TABLA II
API PCM EXTENDIDA [10]

Servicio	Función
Inicialización	MPI_PCM_Init
Finalización	PCM_Exit, PCM_Finalize
Estado del entorno	PCM_Process_Status, PCM_Comm_rank, PCM_Status, PCM_Merge_datacnts
Inicialización	MPI_PCM_Init
Reconfiguración	PCM_Reconfigure, PCM_Split, PCM_Merge, PCM_Split_Collective, PCM_Merge_Collective
Checkpointing	PCM_Load, PCM_Store

tabilidad de cargas de trabajos. A continuación, se presentan varios de los esfuerzos más destacables hechos en este campo:

A. Checkpoint/Restart

Los mecanismos de *Checkpoint/Restart* guardan el estado de una aplicación para reanudar su ejecución más adelante. Esta técnica tradicionalmente se ha utilizado para evitar la pérdida total de los resultados obtenidos durante la ejecución de una aplicación en caso de que se produjera algún fallo.

Sin embargo, el *Checkpoint/Restart* también se ha utilizado para la reconfiguración de trabajos, tanto a nivel de migración, como de redimensionamiento del número de procesos. El método consiste en guardar el estado de la ejecución, para reanudarla con un número de procesos distinto (o reanudar los procesos en otros *hosts*).

En esta sección se presenta una extensión de la API (del inglés, *Application Programming Interface*) PCM (*Process Checkpointing and Migration*) y el *framework* AMPI (*Adaptative MPI)*, basados en mecanismos *Checkpoint/Restart*.

A.1 Extensión de la API PCM

En [10], los autores exploran cómo utilizar la maleabilidad de las aplicaciones junto con *Checkpoint/Restart* para automatizar la reconfiguración. Para ello, extienden la biblioteca MPI PCM con nuevas funciones que el usuario puede utilizar para reconfigurar la aplicación durante su ejecución (ver Tabla II).

Tal y como se muestra en el artículo, la adopción de este nuevo conjunto de funciones convierte el código original en uno mucho más complejo, por lo que se puede considerar poco usable.

A.2 AMPI

El uso directo de MPI para gestionar migraciones requiere un esfuerzo considerable por parte del programador, que tiene que controlar la transferencia de datos entre comunicadores MPI. Es por eso que los autores en [11], en lugar de utilizar bibliotecas MPI, basan su solución en un nuevo paradigma de programación: CHARM++. El artículo presenta un nuevo *framework* (llamado AMPI) que se ejecuta sobre CHARM++. Con esta nueva *capa*, se aprovecha el paradigma de objetos (chares) y los mecanismos de sincronización que ofrece CHARM++ para equilibrar la carga dinámicamente.

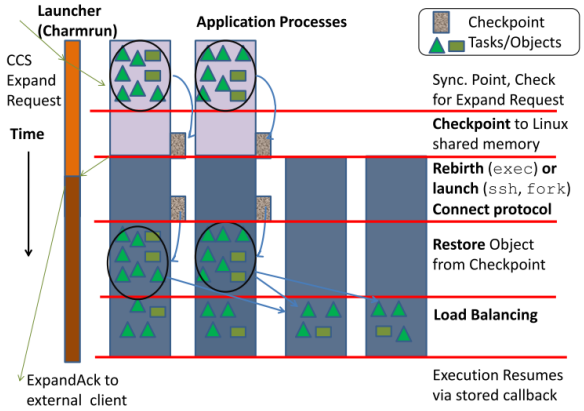

Fig. 1. Esquema de una expansión utilizando AMPI [11].

La Figura 1 ejemplifica un caso en el que el equilibrado de carga implica una expansión. La figura muestra que, cuando llega la orden de expandir y la aplicación llega a un punto de sincronización, los datos son almacenados en memoria. Los procesos iniciales terminan para dar paso a los nuevos procesos, que inicialmente restauran los datos desde la memoria y se encargaran de redistribuirlos entre el nuevo grupo de procesos.

Además, se han añadido dos nuevas estrategias de rebalanceo con las que reconfigurar el sistema para que se adapte a las exigencias:

- **RefineLB**: mueve la carga de los procesadores más cargados a otros que lo estén menos.
- **GreedyLB**: asigna los objetos computacionalmente más pesados a los procesadores con menos carga.

B. Redistribución de datos dinámica

Las soluciones implementadas con mecanismos de *Checkpoint/Restart* son penalizadas en su rendimiento por el hecho de tener que guardar/cargar el estado de la ejecución en/desde memoria. De hecho, en [12], los autores comparan las dos metodologías, *Checkpoint/Restart* y redistribución dinámica de datos, implementadas con las bibliotecas MPI SCR [13] y ULFM [14], respectivamente. En ese trabajo, concluyen que la redistribución dinámica es mucho más rápida que la redistribución utilizando *Checkpoint/-Restart*.

Así pues, en esta sección veremos las herramientas EasyGrid AMS (del ingles, *Application Management System*) y FLEX-MPI.

B.1 EasyGrid AMS

En el trabajo [15] se presenta una solución basada en el biblioteca EasyGrid AMS para ajustar automáticamente el tamaño de un trabajo. Para ello, proponen modificar el código original añadiendo:

- Puntos de reconfiguración para determinar el momento de redimensionamiento del trabajo.
- Mecanismos de reconfiguración para calcular el nuevo grado de paralelismo de una aplicación y redistribuir los datos. Esta decisión es tomada

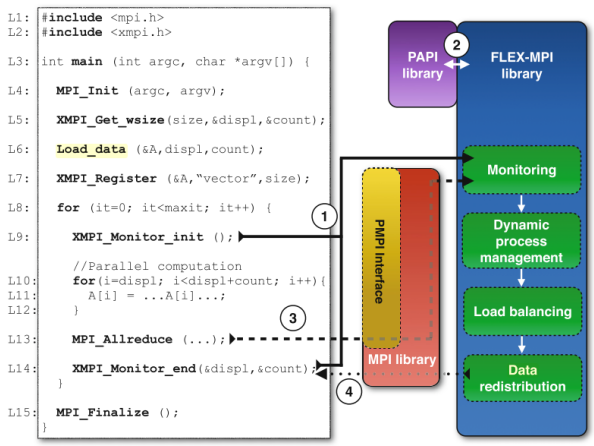

Fig. 2.Estructura y llamadas de un código enlazado con la biblioteca FLEX-MPI [16].

teniendo en cuenta la información de la ejecución que se ha ido recolectando en iteraciones previas.
- Puntos de invocación del mecanismo de reconfiguración durante la ejecución de la aplicación.

B.2 FLEX-MPI

FLEX-MPI [16] es una extensión de MPI que integra 3 funcionalidades: monitorización, equilibrado de carga y redistribución de datos. Para la reconfiguración de aplicaciones se monitoriza el rendimiento de una aplicación mediante contadores hardware y la interfaz de *profiling* de MPI. Con su enfoque, los autores consiguen reducir significativamente el tiempo de ejecución de las aplicaciones. La Figura 2 muestra un ejemplo de una aplicación SPMD utilizando la biblioteca FLEX-MPI. La aplicación necesariamente debe seguir estos pasos:

- Obtener la información sobre los procesos y el entorno actual (línea 5).
- Registrar las estructuras de datos gestionadas por el *runtime* (línea 7).
- Iniciar la monitorización de rendimiento de la aplicación (línea 9).
- Ejecutar el rebalanceo, si es necesario (línea 14).

C. Gestores de recursos adaptativos

Hasta el momento hemos visto bibliotecas y *runtimes* capaces de reconfigurar aplicaciones. Ahora describiremos el estado del arte de herramientas capaces de adaptar la carga de trabajo a las necesidades del sistema.

Para poder adaptar dinámicamente la carga de trabajos necesitaremos principalmente dos herramientas: (i) un gestor de recursos (RMS, del inglés *Resource Manager System*) capaz de modificar la asignación de recursos de los trabajos; y (ii) un *runtime* paralelo para reescalar la aplicación cuando sea necesario.

En esta sección se analizan la integración del *runtime* AMPI en el RMS Torque/Maui, el *framework* ReSHAPE y la API DMR (del inglés, *Dynamic Management of Resources*) que surge de la necesidad de disponer de una solución modular de redistribución de datos dinámica extensible a otros sistemas.

C.1 CHARM++ Torque/Maui

El primer gestor de recursos adaptativo es presentado en [17]. En él, se utilizan aplicaciones CHARM++, maleables por definición gracias a la biblioteca AMPI explicada anteriormente. Los autores presentan una extensión del planificador de clústeres Torque/Maui para que trabaje en cooperación con CHARM++. Esta extensión permite expandir y contraer un trabajo gracias a las siguientes características desarrolladas:

- Extensión del comando para enviar trabajos a la cola, `qsub`, de modo que admita trabajos maleables.
- Funcionalidad para redimensionar un conjunto de recursos.
- Funcionalidad para asignar y liberar nodos.
- Un mecanismo de comunicación entre el gestor y el *runtime*.

C.2 ReSHAPE

ReSHAPE [18] es un *framework* que gestiona desde la maleabilidad de aplicaciones iterativas, hasta la adaptabilidad de la carga de trabajos para incrementar la productividad del sistema. En la arquitectura de ReSHAPE (Figura 3) se diferencian los 3 módulos que lo conforman:

- El monitor de la aplicación que monitoriza cada uno de los trabajos.
- La biblioteca de redimensionamiento para redistribuir los datos en cada reconfiguración.
- El planificador que toma decisiones de reconfiguración con la información que recibe de cada aplicación en cada iteración.

ReSHAPE ha demostrado en varias ocasiones mejorar la productividad en termino de trabajos ejecutados por unidad de tiempo en cargas de trabajos formadas por aplicaciones [19], [20], [21].

Aunque ReSHAPE sea una solución completa de adaptabilidad de cargas de trabajos, también es cerrada ya que necesita que todos los trabajos enviados a la cola hayan sido compilados con sus bibliotecas.

Las dos soluciones presentadas cuentan con características deseables, como la redistribución de datos dinámica (ReSHAPE) y la compatibilidad con herramientas extendidas en el mundo del HPC (CHARM++ con Torque/Maui). Desgraciadamente esas características no las encontramos juntas, siendo ésta la principal desventaja de cada una de las soluciones.

C.3 API DMR

Por un lado, la API DMR [22] utiliza el RMS Slurm Workload Manager para le gestión óptima de recursos en el clúster. Slurm es uno de los gestores de recursos más utilizados en HPC a nivel mundial, es código abierto, portable y altamente escalable [24], [25].

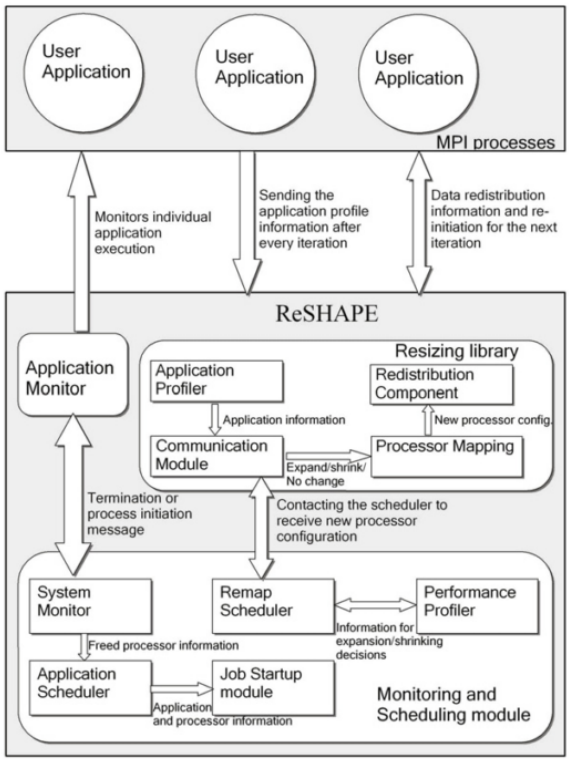

Fig. 3. Arquitectura de ReSHAPE [18].

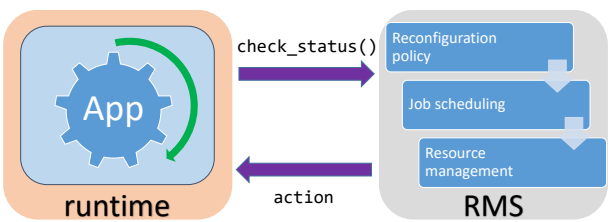

Fig. 4. Esquema de la relación entre los módulos de DMR.

Por el otro, DMR utiliza el *runtime* de OmpSs (Nanos++) para gestionar la reconfiguración de procesos. Además, hemos aprovechado la recientemente incorporada semántica de migración al modelo de programación OmpSs [26] para facilitar la maleabilidad y la redistribución de datos. Nanox++ ha sido extendido, para que pueda establecer una conexión directa con el RMS para así determinar cuándo y cómo llevar a cabo la reconfiguración del trabajo.

La Figura 4 describe la relación entre los dos entes. Tal y como se muestra, el *runtime* consulta el estado del gestor de recursos y, dependiendo de ese estado, el gestor le corresponderá con una acción al *runtime* para que la lleve a cabo.

En el Código 1 se encuentra la estructura básica de una aplicación MPI iterativa que use DMR. La ejecución se inicia en el `main` (línea 1) y, tras inicializar las estructuras de datos, comienza la etapa de cómputo (línea 5). En cada iteración se llama a la función de la DMR API `dmr_check_status` para evaluar el estado del sistema. Esta evaluación puede dar como resultado una expansión o contracción de procesos. En ambos casos (línea 12), el usuario deberá gestionar la acción planificada y programar el *offload* (línea 13). Para llevar a cabo el *offload*, hay

```
1  void main(void) {
2    ...
3    int t = 0;
4    init(data);
5    compute(data, t);
6    ...
7  }
8
9  void compute(data, t0) {
10   for (t=t0; t<timesteps; t++) {
11     action = dmr_check_status(&handler);
12     if (action) {
13       #pragma omp task inout(data)
             onto(handler, rank)
14       compute(data, t)
15       #pragma omp taskwait
16     } else
17       compute_iter(data);
18   }
19 }
```

Código 1
Pseudo-código de una reconfiguración utilizando OmpSs.

que indicar los datos a transferir (`inout(data)`), el nuevo comunicador (`handler`) y el identificador del proceso destino (`rank`). Con esa información, el *runtime* realizará la redistribución de datos. La línea 14 indica dónde han de comenzar su ejecución los nuevos procesos, en este caso, en la propia función que lo ha invocado pero en una iteración distinta. Finalmente, se sincronizan los procesos iniciales (línea 15) para que puedan terminar y dejar que la ejecución prosiga en el nuevo comunicador. En el caso de que no se deba llevar a cabo ninguna acción (línea 16), se realizará la computación de la iteración actual.

IV. Conclusión

En este artículo se ha hecho un repaso de la evolución de la reconfiguración dinámica de trabajos maleables. Hemos visto distintos mecanismos (bibliotecas, paradigmas de programación) que utilizan diferentes técnicas de redistribución de datos. En cuanto a la reconfiguración, algunos investigadores han optado por enfocar sus soluciones a la mejora del rendimiento mediante la monitorización de la ejecución. Otros han conducido su investigación hacia el incremento de la productividad en términos de trabajos ejecutados por unidad de tiempo.

Uno de los mayores inconvenientes que encontramos a la hora de utilizar estas soluciones es la necesidad de readaptar nuestro código a la herramienta. Por ese motivo, en la actualidad ninguno de estos *frameworks* se ha extendido a un sistema en producción.

Tras el análisis llevado a cabo en este artículo, podemos concluir que, para que la reconfiguración se extenda a nivel de producción, se deberían hacer esfuerzos hacía la usabilidad de las herramientas. Es decir, reconfigurar un trabajo no debería suponer esfuerzo a los programadores de aplicaciones HPC. Además, los *frameworks* de reconfiguración deberían

ser compatibles con diferentes gestores de recursos y/o *runtimes* paralelos, para así facilitar su adopción en distintos sistemas.

Agradecimientos

Este trabajo esta financiado por los fondos MINECO y FEDER (TIN2014-53495-R y TIN2015-65316-P). Antonio J. Peña está cofinanciado por el MINECO bajo el programa Juan de la Cierva (IJCI-2015-23266).

Referencias

[1] Message Passing Interface Forum, "MPI: A message-passing interface standard version 3.1," Tech. Rep., June 2015.

[2] Dror G. Feitelson and Larry Rudolph, "Toward convergence in job schedulers for parallel supercomputers," in *Job Scheduling Strategies for Parallel Processing*, 1996, vol. 1162/1996, pp. 1–26.

[3] Kee-Hyun Park and Lawrence W. Dowdy, "Dynamic partitioning of multiprocessor systems," *International Journal of Parallel Programming*, vol. 18, no. 2, pp. 91–120, apr 1989.

[4] John Zahorjan, Cathy McCann, John Zahorjan, and Cathy McCann, "Processor scheduling in shared memory multiprocessors," in *Proceedings of the 1990 ACM SIGMETRICS conference on Measurement and modeling of computer systems - SIGMETRICS '90*, New York, New York, USA, 1990, vol. 18, pp. 214–225, ACM Press.

[5] Cathy McCann, Raj Vaswani, and John Zahorjan, "A dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors," *ACM Transactions on Computer Systems*, vol. 11, no. 2, pp. 146–178, may 1993.

[6] A. Tucker, A. Gupta, A. Tucker, and A. Gupta, "Process control and scheduling issues for multiprogrammed shared-memory multiprocessors," in *Proceedings of the twelfth ACM symposium on Operating systems principles - SOSP '89*, New York, New York, USA, 1989, vol. 23, pp. 159–166, ACM Press.

[7] Anoop Gupta, Andrew Tucker, Shigeru Urushibara, Anoop Gupta, Andrew Tucker, and Shigeru Urushibara, "The impact of operating system scheduling policies and synchronization methods of performance of parallel applications," in *Proceedings of the 1991 ACM SIGMETRICS conference on Measurement and modeling of computer systems - SIGMETRICS '91*, New York, New York, USA, 1991, vol. 19, pp. 120–132, ACM Press.

[8] Jitendra Padhye and Lawrence Dowdy, "Dynamic versus adaptive processor allocation policies for message passing parallel computers: An empirical comparison," in *Job Scheduling Strategies for Parallel Processing (IPPS)*, 1996, pp. 224–243.

[9] Rüdiger Esser and Renate Knecht, *Intel Paragon XP/S - Architecture and Software Environment*, pp. 121–141, Springer Berlin Heidelberg, Berlin, Heidelberg, 1993.

[10] K. El Maghraoui, Travis J. Desell, Boleslaw K. Szymanski, and Carlos A. Varela, "Dynamic malleability in iterative MPI applications," in *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, May 2007, pp. 591–598.

[11] Abhishek Gupta, Bilge Acun, Osman Sarood, and Laxmikant V Kalé, "Towards realizing the potential of malleable jobs," in *21st International Conference on High Performance Computing (HiPC)*, 2014.

[12] Pierre Lemarinier, Khalid Hasanov, Srikumar Venugopal, and Kostas Katrinis, "Architecting malleable MPI applications for priority-driven adaptive scheduling," in *Proceedings of the 23rd European MPI Users' Group Meeting (EuroMPI)*, 2016, pp. 74–81.

[13] Adam Moody, Greg Bronevetsky, Kathryn Mohror, and Bronis R. de Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC10)*, Nov. 2010.

[14] W. Bland, A. Bouteiller, T. Herault, G. Bosilca, and J. Dongarra, "Post-failure recovery of MPI communication capability: Design and rationale," *International Journal of High Performance Computing Applications*, vol. 27, no. 3, pp. 244–254, June 2013.

[15] Felipe S. Ribeiro, Aline P. Nascimento, Cristina Boeres, Vinod E F Rebello, and Alexandre C. Sena, "Autonomic malleability in iterative MPI applications," in *Symposium on Computer Architecture and High Performance Computing*, 2013, pp. 192–199.

[16] Gonzalo Martín, Maria Cristina Marinescu, David E. Singh, and Jesús Carretero, "FLEX-MPI: an MPI extension for supporting dynamic load balancing on heterogeneous non-dedicated systems," in *Euro-Par Parallel Processing*, Aug. 2013, pp. 138–149.

[17] Suraj Prabhakaran, Marcel Neumann, Sebastian Rinke, Felix Wolf, Abhishek Gupta, and Laxmikant V. Kale, "A batch system with efficient adaptive scheduling for malleable and evolving applications," in *2015 IEEE International Parallel and Distributed Processing Symposium*, May 2015, pp. 429–438.

[18] Rajesh Sudarsan and Calvin J. Ribbens, "ReSHAPE: A framework for dynamic resizing and scheduling of homogeneous applications in a parallel environment," in *International Conference on Parallel Processing*, 2007.

[19] Rajesh Sudarsan, Calvin J. Ribbens, and Diana Farkas, "Dynamic resizing of parallel scientific simulations: A case study using LAMMPS," in *International Conference on Computational Science (ICCS)*, 2009, pp. 175–184.

[20] R. Sudarsan and C.J. Ribbens, "Scheduling resizable parallel applications," in *International Symposium on Parallel & Distributed Processing*. May 2009, IEEE.

[21] Rajesh Sudarsan and Calvin J. Ribbens, "Combining performance and priority for scheduling resizable parallel applications," *Journal of Parallel and Distributed Computing*, vol. 87, pp. 55–66, 2016.

[22] Sergio Iserte, Antonio J. Peña, Rafael Mayo, Enrique S. Quintana-Ortí, and Vicenç Beltran, "Dynamic Management of Resource Allocation for OmpSs Jobs," in *Proceedings of the First PhD Symposium on Sustainable Ultrascale Computing Systems (NESUS PhD 2016).*, Jesus Carretero, Javier Garcia Blas, and Dana Petcu, Eds., Timisoara, Romania, 2016, pp. 55–58.

[23] Andy B Yoo, Morris A Jette, and Mark Grondona, "SLURM: Simple Linux utility for resource management," in *9th International Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, 2003, pp. 44–60.

[24] "Slurm Workload Manager," http://slurm.schedmd.com.

[25] "The Top500 List," https://www.top500.org.

[26] Florentino Sainz, Jorge Bellon, Vicenc Beltran, and Jesus Labarta, "Collective offload for heterogeneous clusters," in *22nd International Conference on High Performance Computing (HiPC)*, 2015.