Aplicación sintética para el estudio de maleabilidad en computación de altas prestaciones

Iker Martín-Álvarez[1], José I. Aliaga[1], Maribel Castillo[1], Sergio Iserte[2] y Rafael Mayo[1]

Resumen— **Hoy en día, la mejora del rendimiento en los grandes clusters de ordenadores recomienda el desarrollo de aplicaciones maleables. Así, durante la ejecución de estas aplicaciones en un trabajo, el sistema de gestión de recursos (RMS) puede modificar su asignación de recursos, con el fin de aumentar el rendimiento global. Existen diferentes alternativas para completar los distintos pasos en los que se descompone la reasignación de recursos. Para encontrar las mejores alternativas, este trabajo introduce una aplicación sintética iterativa maleable MPI capaz de modificar, en tiempo de ejecución, el número de procesos MPI en función de varios parámetros. La aplicación incluye un módulo de rendimiento para medir el coste de las etapas dentro de los pasos, desde la gestión de los procesos hasta la redistribución de los datos. De este modo, el análisis de diferentes escenarios permitirá concluir cómo debe realizarse la redistribución de datos en diferentes circunstancias.**

Palabras clave— **MPI, Maleabilidad, Aplicaciones iterativas, Programación híbrida.**

I. Introducción

LOS sistemas de computación de alto rendimiento (HPC) son tradicionalmente compartidos por usuarios que ejecutan sus aplicaciones en forma de trabajos. Los sistemas de gestión de recursos (RMS) operan transversalmente a través de herramientas que se encargan de controlar los recursos disponibles y cómo se asignan.

La maleabilidad permite a las aplicaciones modificar los recursos computacionales asignados en tiempo de ejecución. En los superordenadores es habitual encontrar trabajos paralelos distribuidos que suelen desarrollarse utilizando la interfaz de paso de mensajes (MPI) como herramienta de desarrollo estándar de facto. En particular, en este trabajo, consideramos la maleabilidad como la capacidad de un trabajo paralelo distribuido de cambiar su tamaño, en términos de procesos MPI, y modificar los recursos computacionales inicialmente asignados al trabajo en cualquier punto de la ejecución tantas veces como sea necesario.

La implantación de la maleabilidad debe apoyarse en sus beneficios, que pueden aparecer de dos formas diferentes. Desde el punto de vista de cada aplicación individual, el beneficio puede obtenerse en el aumento de su rendimiento particular, al expandir el trabajo. Desde el punto de vista del sistema global, este beneficio repercute en el aumento del rendimiento global del sistema en cuanto a productividad y trabajos finalizados por segundo.

Redimensionar un trabajo implica modificar el número de procesos MPI que lo están ejecutando y redistribuir los datos entre ellos para que la ejecución continue. Evidentemente, la operación de redistribución es la que introducirá más sobrecarga, generando un impacto negativo en ambos, tanto en el rendimiento de la aplicación como en el del sistema [1], [2], [3].

Hay dos enfoques diferentes para realizar esta redistribución de datos, de forma similar al proceso de migración de memoria vivo de máquinas virtuales [4]. El primer enfoque, redistribución síncrona (SR), implica que los procesos padres detienen su ejecución mientras se lleva a cabo esta operación. El segundo enfoque, redistribución asíncrona (AR), implica que los datos se redistribuyen parcialmente desde los procesos padres a los hijos como una tarea en segundo plano, sin detener la ejecución de los padres. A continuación, los procesos padres se detienen y se procede a la redistribución del resto de los datos en una comunicación síncrona.

Para ilustrar ambos enfoques, la Figura 1 muestra un ejemplo sencillo sobre cómo la maleabilidad puede afectar al tiempo de ejecución de una aplicación iterativa cuando se amplía el número de procesos de 2 a 4. Desde la parte superior a la inferior de la figura, la primera barra (sin maleabilidad) representa una aplicación iterativa ejecutando siete iteraciones. Las dos siguientes representan versiones maleables de la aplicación paralela utilizando las dos técnicas propuestas para realizar la distribución de datos. La segunda barra (síncrona maleable) emplea SR, mientras que la última (asíncrona maleable) utiliza AR. Ambas versiones necesitan definir un punto de redimensionamiento en la aplicación en el que se debe detener la ejecución para iniciar el método de reconfiguración. En este momento, se crean nuevos procesos y se redistribuyen los datos de los procesos anteriores a los nuevos. La creación de los nuevos procesos obliga a detener la ejecución. Sin embargo, la redistribución de datos puede realizarse como una tarea en segundo plano mientras se siguen ejecutando las iteraciones en los procesos padre. Cuando la redistribución finaliza, ya sea para AR o SR, los hijos toman el control y continúan su ejecución.

Este trabajo presenta una herramienta parametrizable para emular el comportamiento de una aplica-

[1]Dpto. de Ingeniería y Ciencia de los Computadores, Universitat Jaume I de Castelló, e-mails: martini@uji.es, aliaga@uji.es, castillo@uji.es, mayo@uji.es

[2]Personal investigador postdoctoral (Programa Generalitat Valenciana) Dep. d'Enginyeria Mecànica i Construcció, e-mail: siserte@uji.es

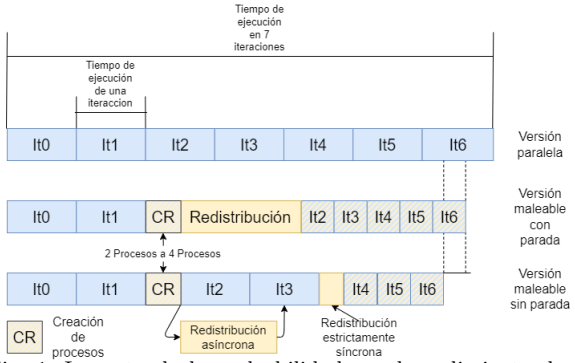

Fig. 1: Impacto de la maleabilidad en el rendimiento de la aplicación, utilizando comunicaciones síncronas o asíncronas.

ción científica con unas determinadas características. La aplicación sintética permite analizar los diferentes escenarios/situaciones y ayudará a estudiar el comportamiento de esta aplicación si fuese maleable y su rendimiento. Esta herramienta, también puede ser útil para configurar cargas de trabajo sintéticas y analizar el impacto de la maleabilidad en el rendimiento del sistema.

En la siguiente sección se presentan los fundamentos de la maleabilidad, y los esfuerzos previos en este tema. A continuación, se presenta las principales características de la aplicación sintética maleable, mostrando como las diferentes etapas en la que se descompone su funcionamiento. Seguidamente, se muestran las resultados obtenidos al utilizar la aplicación sintética en diferentes escenarios. Finalmente, el análisis de los resultados dan lugar a unos conclusiones finales.

II. Fundamentos

Los primeros pasos en maleabilidad se presentan en [5] para sistemas de memoria compartida, donde los trabajos activos se expulsaban para redistribuir la carga de trabajo de los procesadores. Para sistemas distribuidos [6], [7], [8], la primera solución se realizó con mecanismos *Checkpoint/Restart* [9], que escriben el estado del trabajo en un archivo de punto de control que posteriormente se utiliza para reiniciar el trabajo con un número diferente de procesos.

En [10], se utiliza el entorno EasyGrid AMS para ejecutar trabajos MPI en grids de clusters informáticos, proponiendo estrategias para redimensionar automáticamente esos trabajos a través de una serie de funciones que sirven para determinar los puntos de reconfiguración, calcular el nuevo grado de paralelismo y aplicar la reconfiguración.

Hay algunos entornos que presentan la interacción entre aplicaciones maleables y administradores de recursos. Uno de los más estudiados es ReSHAPE [11], que integra técnicas de reconfiguración de trabajos en la planificación de recursos, pero el entorno requiere que todas las aplicaciones sean maleables [12]. En [2] se presenta una estrategia similar a EasyGrid AMS y ReSHAPE: Flex-MPI [13]. Esta biblioteca integra cuatro nuevas funcionalidades: monitorización, equilibrio de carga, redistribución de datos y un modelo de predicción computacional. En este caso, la maleabilidad se centra únicamente en el análisis de las prestaciones de los trabajos.

La maleabilidad también se ha implementado usando modelos de programación basados en tareas como CHARM++[14], que virtualiza procesadores y su paradigma se basa en objetos migrables llamados *chares*. También incluye un subsistema de reconfiguración de procesos basado en *Checkpoint/Restart* de forma que, tras guardar el estado de una aplicación, los objetos *chares* pueden redistribuirse entre el nuevo número de procesadores. La operación es compatible con Adaptative MPI (AMPI) [3], que es una implementación de MPI en el sistema de ejecución Charm++, donde se implementan los puntos de control en la memoria.

En [15], DMRlib, se presenta un modelo de programación basado en tareas y OmpSs. DMRlib permite a los usuarios determinar la cantidad de recursos con los que una aplicación puede comenzar a ejecutarse y, posteriormente, modificarlos durante su ejecución. Esta variación tiene en cuenta, por un lado, que la aplicación solicita los recursos y, por otro, que el gestor de recursos decide con cuántos recursos se puede ejecutar la aplicación en el siguiente periodo. Por lo tanto, considera tanto el rendimiento de la aplicación como del sistema global.

También existen infraestructuras que permiten la ejecución de aplicaciones maleables como Invasive MPI[16], que también es una extensión de las APIs de MPICH y Slurm. La integración de ambos sistemas en uno permite a Slurm crear o destruir procesos mientras administra la asignación de recursos. Para ello, se modifica la implementación de MPICH y Slurm en [17], [18] y, a su vez, se agregan nuevas funcionalidades para habilitar la maleabilidad. Además, permite que los procesos verifiquen periódicamente si Slurm ha iniciado una reconfiguración.

En [19] se puede consultar un estado del arte extendido de la maleabilidad en el que se muestran además otras soluciones no mostradas en esta sección.

Sin embargo, ninguno de estos trabajos previos ha tenido en cuenta la comunicación asíncrona para la redistribución de datos o creación de procesos, lo que podría ofrecer un mejor rendimiento [20] tanto a la aplicación como al sistema.

III. Aplicación Sintética MPI Maleable

Esta sección presenta los parámetros de configuración, los componentes internos y la funcionalidad de nuestra herramienta. Aunque el tipo de aplicación productor-consumidor ha sido ampliamente estudiado [16], [21], históricamente el objetivo de la maleabilidad se ha aplicado sobre aplicaciones iterativas [22], [23], [24]. Es por eso que hemos enfocado nuestro trabajo en la emulación del comportamiento de aquellas aplicaciones MPI iterativas que puedan cambiar de tamaño (reducir o expandir) sobre la marcha. No obstante, el uso adecuado de los parámetros de configuración permitirá emular el comportamiento de casi cualquier tipo de aplicación MPI.

La Figura 2 muestra un diagrama de flujo de nuestra herramienta, en el que aparecen seis módulos principales: (i) *Inicialización*, (ii) *Cálculo*, (iii) *Comunicación*, (iv) *Redimensionar*, (v) *Monitorización* y (vi) *Finalización*. Listing 1 muestra un ejemplo de un código con todos los módulos enumerados anteriormente diferenciados, respectivamente, por los colores: Inicialización en verde, redimensionamiento en púrpura, computación/comunicación en naranja y finalización en rojo. En este sentido, las líneas 3-13 inicializan los grupos de aplicaciones y procesos; las líneas 14-22 representan un cálculo iterativo; el redimensionado tiene lugar desde la línea 24 a la 28; y en la línea 29 se almacenan los datos.

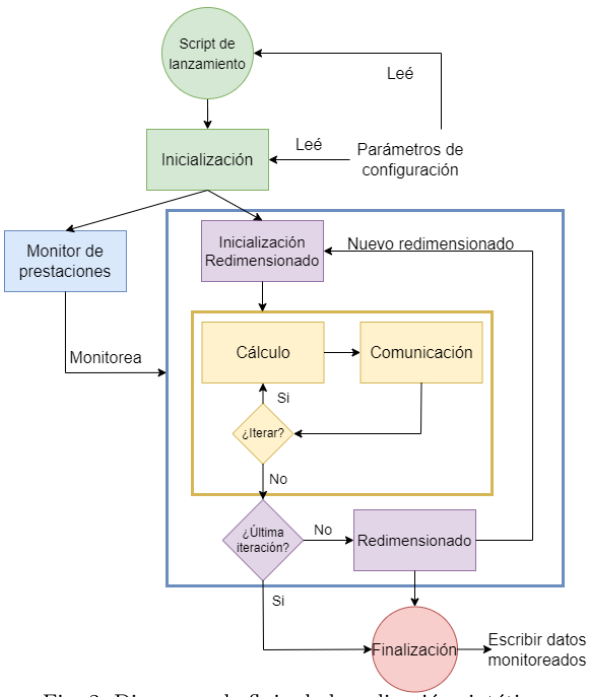

Fig. 2: Diagrama de flujo de la aplicación sintética.

Es importante destacar, que durante un redimensionado, los procesos originales se identifican como "padres", mientras que el nuevo conjunto de procesos se consideran "hijos".

Las siguientes subsecciones explican la funcionalidad de estos módulos y los parámetros de configuración que los definen.

A. Inicialización

Este módulo se encarga de leer/distribuir los parámetros almacenados en el archivo de configuración que definen el comportamiento de la aplicación. Si se llega por primera vez al módulo, el proceso 0 lee los parámetros, y los retransmite al resto de procesos MPI. Si no es la primera vez, significa que se ha realizado un redimensionamiento, y el proceso padre 0 envía los parámetros a todos los procesos hijos, que los reciben en las líneas 10-12.

Además, este módulo también se encarga de calcular algunos parámetros (no directos), cuyos valores dependen de algunos de los definidos en el archivo de configuración y del sistema sobre el que se realiza la ejecución. Uno de los parámetros no directos más importantes es `op` en el Listing 1. Este valor define el número de veces que se debe ejecutar una operación particular en el módulo de cómputo para alcanzar el tiempo máximo de ejecución en una iteración, definido en el parámetro T_{it}.

Todos los parámetros no directos, excepto `op`, se calculan solo una vez y se envían de padres a hijos en cada cambio de tamaño. Por su parte, `op` se calcula cada vez porque depende de diferentes parámetros.

B. Cálculo

La herramienta presentada permite elegir dos tipos diferentes de cómputo que corresponden a algunos de los comportamientos más comunes en las aplicaciones de HPC:

- Aplicaciones limitadas por el cómputo: Simula un tipo de aplicación donde la utilización de la CPU es el factor principal del coste computacional. En este caso, se ejecuta el método Monte Carlo para estimar π, en el que no se requiere acceso a la memoria de datos.
- Aplicaciones limitadas por el acceso a memoria: Simula un tipo de aplicación donde los accesos a memoria representan el factor principal del coste computacional. La opción elegida ha sido calculad un producto matriz-vector donde la matriz está almacenada por columnas.

En un futuro próximo, se pretende simular aplicaciones limitadas por la E/S.

El parámetro P_t en el archivo de configuración, selecciona uno de estos dos procedimientos de cálculo.

C. Comunicación

Para definir el impacto de las comunicaciones en la simulación, se deben establecer dos parámetros en el archivo de configuración: el volumen de datos asociados a la comunicación (C_b) y el tipo de comunicación (C_t). Para el segundo de los parámetros, las alternativas son las siguientes:

- Punto a punto: Esta comunicación se implementa utilizando la operación `MPI_Sendrecv`. Suponiendo que todos los procesos están conectados en una topología de anillo, cada proceso envía datos al siguiente proceso y recibe información del anterior.
- Colectiva uno a todos: Esta es una comunicación que involucra a todos los procesos del grupo, pero solo uno envía datos. Se realiza con una llamada a la operación `MPI_Bcast`.
- Colectiva todos a todos: Es una comunicación que involucra a todos los procesos del grupo, todos ellos enviando y recibiendo información. Se implementa utilizando la operación `MPI_Alltoall`.
- Reducción: Esta es una comunicación que involucra a todos los procesos del grupo, pero solo uno recibe datos, y en la que se ejecuta algún cálculo sobre los datos. Se implementa con la operación `MPI_Reduction`.

```
int main() {
  // El primer grupo inicializa la aplicación
  if(init) {
    if (rank == 0)
        read_config();
    broadcast_config();  // Desde P_0 al resto
    calculate_non_direct_parameters();

  } else {
    broadcast_config();  // Recibir del padre P_0
    calculate_non_direct_parameters();
    redistribute_data(); // Recibir de los padres
  }
  // Fase de Cálculo
  for(int iter=0; iter < Iters; iter++) {
    // Calcula hasta consumir T_it segundos
    for(int i=0; i < op; i++) {
      compute(N, Pt);
    }
    // Comunica Cb bytes (opcional)
    communications(Cb, Ct);
  }
  // Reconfigurar si no es el último resize
  if (!last_resize) {
      create_child_processes();
      broadcast_config();  // Desde P_0 a los hijos
      redistribute_data(); // Enviar a los hijos
  }
  store_performance_data();
}
```

Listing 1: Esqueleto básico de la aplicación sintética.

Todas las comunicaciones se realizan después de que se hayan completado los cálculos, y cada proceso envía C_b bytes divididos entre el número de procesos en el grupo, a excepción de la operación del colectivo uno a todos, en la que un mismo proceso envía C_b bytes que son recibidos por el resto de procesos.

D. Redimensionado

Este módulo permite redimensionar la aplicación posibilitando la maleabilidad. En Listing 1 se puede observar como se realiza una reconfiguración después de ejecutar `Iters` iteraciones.

El proceso de cambio de tamaño se realiza en tres pasos:

- Se comunica con el RMS para mostrar la disponibilidad de la aplicación a ser redimensionada.
- Se crea un nuevo conjunto/grupo de procesos MPI.
- Se redistribuyen los datos de los procesos padre a los procesos hijo.

El primer paso se describe en [1] y consiste en comunicarse con el RMS para verificar el estado del sistema, y decidir si se deben asignar más recursos a un trabajo en ejecución (expandir) o si algunos de sus recursos deben desasignarse (reducir). En la versión actual de la herramienta se emula este comportamiento, incluyendo el parámetro `Procs` en el archivo de configuración, que indica el nuevo número de procesos de la aplicación.

En el segundo paso, los procesos se crean calculando el número de procesos que se necesitan para utilizar adecuadamente los recursos disponibles y cómo deben distribuirse. A continuación se ejecuta `MPI_Comm_spawn`, con un argumento `MPI_Info` para indicar el mapeado físico de los nuevos procesos. Este argumento es necesario para evitar que la API genere los procesos en los mismos nodos que donde están sus padres.

Para analizar el impacto de la asignación de procesos a núcleos del sistema se han considerado dos estrategias extremas:

- Spread: En esta estrategia, se considera que el RMS intenta asignar el mínimo número de nucleos en todos los nodos disponibles.
- Compact: En ese caso, el RMS intenta reducir el número de nodos en la asignación de los núcleos.

El parámetro `Dist` en la configuración determina qué estrategia se utiliza en la simulación. En el futuro, se considerarán otras estrategias intermedias para emular de un modo más aproximado el comportamiento real de un RMS.

En el tercer paso, los datos se redistribuyen de padres a hijos. Los datos en cada grupo de procesos se distribuyen linealmente entre la memoria de todos los procesos involucrados en el cómputo.

La redistribución se puede realizar con comunicaciones síncronas, con comunicaciones asíncronas o con una combinación de ambas. En el archivo de configuración aparecen dos parámetros para definir esta tarea, `SDR` y `ADR`, que indican cuántos bytes se envían en cada modo. Si se deben realizar ambos tipos de comunicación, primero se inicia la comunicación asíncrona, y cuando finaliza, los padres se detienen para iniciar la síncrona. Este orden es importante para las aplicaciones reales porque primero los datos constantes se envían de forma asíncrona y, mientras, los padres continúan ejecutando la aplicación. A continuación, cuando estos han sido recibidos, se inicia la comunicación síncrona donde se envían datos variables. No es posible realizarlo al revés ya que los datos variables se modifican después de cada iteración.

Cada uno de los diferentes tipos de redistribución se explica en detalle a continuación.

Redistribución síncrona

Esta operación se puede implementar mediante una única llamada a `MPI_Alltoallv`, que permite especificar cómo enviar datos de padres a hijos.

Redistribución asíncrona

Permite realizar la redistribución de datos como una tarea en segundo plano. De esta forma, la aplicación continúa ejecutando los procedimientos de cómputo mientras se redimensiona.

Hay cuatro implementaciones diferentes posibles, tres basadas en primitivas asíncronas MPI y un enfoque basado en pthreads. Los basados en primitivas asíncronas MPI son los siguientes:

- Punto a punto: Redistribuye datos utilizando funciones punto a punto asíncronas. La redistribución se considera completada para los padres cuando *MPI_Test* indica que se han finalizado todas las funciones *MPI_Isend*.
- Colectivas todos a todos (en Padres): Redistribuye los datos utilizando la función

`MPI_Ialltoallv`. La redistribución se considera completadada para los padres cuando *MPI_Test* indica que se ha finalizado la función para todos los padres.
- Colectiva todos a todos (en Hijos): Similar al anterior, pero la comunicación se completa cuando *MPI_Test* indica que se ha finalizado la función para todos los hijos. Este paso necesita una comunicación adicional para notificar a los padres que los hijos han recibido todos los datos.

El cuarto caso se basa en *pthreads*, que requiere iniciar MPI mediante la función *MPI_Init_thread* con el valor *MPI_THREAD_MULTIPLE* para permitir llamadas simultáneas a MPI desde múltiples hilos. Al iniciar una redistribución con este método, se lanza un nuevo hilo en cada padre que se encarga de la redistribución de datos utilizando la función `MPI_Alltoallv`.

E. Monitorización

Este módulo captura tiempos de ejecución para diferentes etapas de la aplicación, y los almacena cuando finaliza la simulación. La medición utiliza la rutina *MPI_Wtime*, considerando el análisis de tiempo de los siguientes procedimientos:

- T_c: Tiempo por iteración mientras se realiza la redistribución asíncrona de datos. Este tiempo es igual a $T_{it} * \omega$, donde ω es la sobrecarga debido a la superposición de la iteración con la redistribución de datos.
- T_{spawn}: Tiempo requerido para generar los nuevos procesos.
- T_{SR}: Tiempo utilizado para la redistribución síncrona de datos.
- T_{AR}: Tiempo empleado para la redistribución asíncrona de datos.
- T_{start}: Tiempo empleado para la redistribución inicial de datos.
- T_{total}: Tiempo total desde el inicio hasta el final de la aplicación.

Para cada iteración ejecutada, también se almacena cuántas veces se ha ejecutado el procedimiento de cómputo y si se ha producido un redimensionado en segundo plano.

F. Finalización

Este módulo se encarga de completar la ejecución de un grupo de procesos, y se ejecuta tras cada redimensionado. Para ello el grupo de procesos almacenan sus métricas en un fichero externo para su posterior análisis. A continuación, los padres finalizan su ejecución y los hijos continúan ejecutando el número de iteraciones que se ha definido en el archivo de configuración para este nuevo grupo de procesos. Cuando se completa la ejecución del último grupo de procesos, la aplicación finaliza.

G. Configuración

Las características de los diferentes trabajos de un centro de datos científico a simular, así como los parámetros para un posible redimensionamiento, se definen en un archivo de configuración. Este archivo debe crearse antes de ejecutar la aplicación y será leído por el proceso 0 del primer grupo de procesos (ver Listing 1).

Este archivo involucra dos tipos de parámetros: Los que permiten emular un tipo de aplicación, y los que indican cómo redimensionar el trabajo.

Los parámetros dedicados a emular el comportamiento de una aplicación son los siguientes:

- `Pt`: Tipo de aplicación a ejecutar, ya sea estimación de π u operación matriz-vector.
- `N`: Tamaño de la matriz cuadrada relacionada con la operación matriz-vector o el número de cuadrados en la estimación π.
- `Cb`: Total de bytes comunicados entre procesos en cada iteración.
- `Ct`: Tipo de comunicación a utilizar en cada iteración. Es uno de los cuatro descritos en la subsección III-C.
- T_{it}: Tiempo base de una iteración en la aplicación simulada en un núcleo (secuencial). Cada iteración se realiza ejecutando el procedimiento de cálculo seleccionado varias veces hasta alcanzar T_{it}.

Los siguientes parámetros están dedicados al ajuste del cambio de tamaño:

- `R`: Número de redimensionamientos a realizar. Un valor igual a cero indica que no se realizará ninguno.
- `SDR`: Número de bytes a redistribuir de forma síncrona en cada etapa de redistribución.
- `ADR`: Número de bytes a redistribuir de forma asíncrona en cada etapa de redistribución.
- `AT`: Tipo de redistribución asíncrona, ya sea utilizando primitivas asíncronas MPI o *pthreads*.

`SDR` y `ADR` son valores independientes, y ambos pueden tener un valor diferente a cero, lo que indica que puede haber una redistribución síncrona y/o asíncrona. Esto es lo que sucede en aplicaciones reales, ya que algunos datos se modifican en cada iteración y, por lo tanto, no todos los datos se pueden enviar de forma asíncrona, ya que esos datos dejarían de ser válidos para los hijos. De esta forma, es posible utilizar `SDR` y `ADR` en la misma reconfiguración.

Después de los parámetros anteriores, se definen `R+1` conjuntos de parámetros, que caracterizan la asignación de procesos a núcleos después de cada reconfiguración (reducir, expandir o migrar). Para cada grupo, los parámetros relacionados son los siguientes:

- `Iters`: Total de iteraciones a ejecutar por este grupo de procesos antes de cambiar el tamaño o finalizar la aplicación.
- `Procs`: Número de procesos en este grupo.
- `Dist`: Estrategia aplicada para emular la asignación de nodos, Spread o Compact.
- `FactorS`: Este factor de aceleración emula cómo se ve afectado el rendimiento de la aplicación para este grupo de procesos y tiene que ser esti-

```
[general]
R=1          # Número de cambios de tamaño
N=100000     # Tamaño del problema de cálculo
Pt=0         # Tipo de procedimiento
Cb=10000000 # Bytes enviados por iteración
Ct=1         # Tipo de comunicación por iteración
SDR=0        # Bytes redistrib. síncr.
ADR=1000000000 # Bytes redistrib. asíncr.
AT=3         # Tipo de redistribución asíncrona
T_it=4       # Tiempo por iteración
;end [general]
[resize0]
Iters=1      # Nómero de iteraciones del grupo
Procs=2      # Número de procesos del grupo
FactorS=0.5 # Factor de escalabilidad
Dist=spread # Tipo de asignacion de procesos
;end [resize0]
[resize1]
iters=10
Procesos=10
FactorS=0.1
Dist=spread
;end [resize1]
```

Listing 2: Ejemplo de archivo de configuración

mado previamente por el usuario. Por ejemplo, si la aplicación tuviera una aceleración ideal, se tendría que $FactorS = 1/Procs$. Por lo tanto, este parámetro determina la escalabilidad de la aplicación emulada. Al multiplicar $FactorS * T_{it}$ se obtiene el tiempo por iteración para este grupos de procesos.

Además, es necesario calcular otros parámetros para completar la emulación, siendo los más importantes los siguientes:

- T_{op}: Tiempo para ejecutar una vez una aplicación `Pt` cuyo tamaño es igual a `N`. Esto se calcula después de leer el archivo de configuración.
- `op`: Número de veces que se debe ejecutar la aplicación `Pt` de tamaño `N` para lograr la escalabilidad definida en `FactorS` cuando el coste secuencial es T_{it}. Esto se define como $op = T_{it} * FactorS/T_{op}$.

El Listing 2 muestra un ejemplo de fichero de configuración que realiza una expansión de 2 a 10 procesos con solo comunicaciones asíncronas basadas en *pthreads* y 1 redimensionado.

IV. Resultados experimentales

Este apartado presenta los resultados de las simulaciones realizadas con la aplicación sintética descrita anteriormente. Los experimentos se han ejecutado utilizando dos servidores, cada uno con dos procesadores de 10 núcleos Intel Xeon 4210, y utilizando MPICH 3.4.3. En todos los casos se ha realizado un único redimensionado en cada experimento, haciéndolo desde 2, 10, 20 y 40 procesos a cualquiera de los mismos números y probando dos estrategias de asignación (Spread o Compact).

Para estos resultados, sólo consideramos dos valores de `FactorS`: "1", que indica un tiempo constante e independiente del número de procesos; o un valor variable que indica una aceleración ideal para cada número de procesos (es decir, si se duplica el número de procesos, el tiempo por iteración se reduce a la mitad. Por tanto, $FactorS = 1/Procs$). Además, en todos los casos se realiza una redistribución de datos de 1GB basada en *pthreads* en cada redimensionado, que puede ser totalmente síncrono o totalmente asíncrono. Por último, se realizan once iteraciones de 4 segundos para la estimación de Montecarlo π, con 1MB de comunicación por iteración basada en una operación Colectiva uno-a-todos. Un ejemplo de archivo de configuración para este conjunto de parámetros se muestra en el Listing 2.

Los resultados que se muestran a continuación corresponden a la media de cinco ejecuciones diferentes para reducir la variabilidad. Además, las figuras de este apartado muestran los tiempos de ejecución para diferentes número de procesos padre (NP) e hijo (NC). Dependiendo de qué grupo tenga más procesos, la reconfiguración corresponde a una expansión o a una contracción. Obsérvese que las etiquetas del eje X representan la operación de redimensionado, en la que los procesos padres son el primer número, y el segundo corresponde al número de procesos hijo.

Se han realizado diferentes experimentos para probar la redistribución de datos asíncrona bien utilizando la operación MPI *non-blocking* colectiva de todos a todos (padres) o bien *pthreads*. La opción *pthreads* ha demostrado ser más regular que la operación MPI *non-blocking*. La Figura 3 muestra cómo la mayoría de los casos configurados con la estrategia Compact y un *FactorS* constante, ofrecen un mayor rendimiento. Por ello, en el resto de las pruebas se ha utilizado este mecanismo de comunicación para el caso asíncrono.

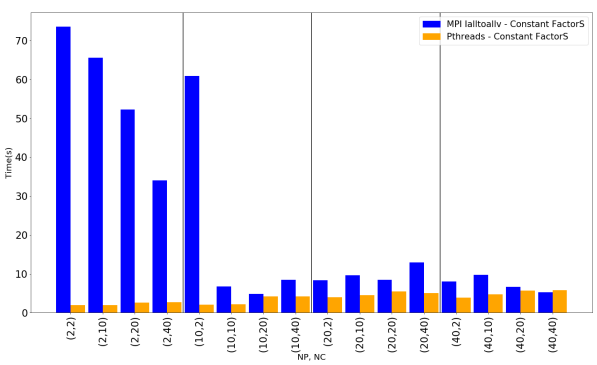

Fig. 3: Tiempo de redistribución asíncrona para una estrategia Compact y *FactorS* constante en función del parámetro *AT*. NP indica el total de procesos padres y NC los procesos hijos.

Las Figuras 4 y 5 comparan el tiempo total de ejecución de una aplicación con un *FactorS* variable (speedup ideal) y comparan las estrategias Spread y Compact. Se muestra para un número diferente de padres e hijos considerando la maleabilidad y utilizando diferentes tipos de comunicaciones para la redistribución de datos.

Se observa que la redistribución asíncrona (AR) siempre ofrece un mejor rendimiento cuando se reduce el número de procesos, mientras que la redistribución síncrona (SR) siempre es mejor cuando se aumenta el número de procesos. Esto ocurre porque cuando se reduce con AR, los procesos padres ejecutan más iteraciones, y estos tienen un tiempo de ejecución menor por iteración que los hijos. Por el

contrario, cuando se redimensiona a más procesos con la estrategia AR, esas iteraciones extra ejecutadas por los padres tienen un tiempo de ejecución mayor que la de los hijos.

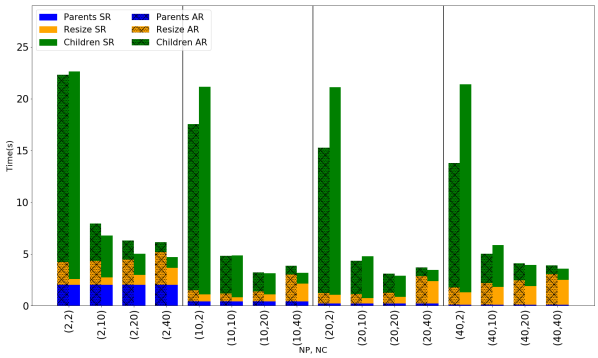

Fig. 4: Tiempo total de ejecución (tiempo de ejecución de los padres (azul), tiempo de redimensionado (naranja) y tiempo de ejecución de los hijos (verde)) para una estrategia Spread y *FactorS* para un speedup ideal. NP indica el total de procesos padres y NC los procesos hijos.

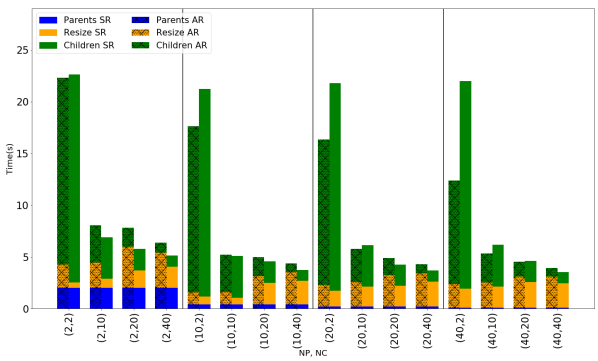

Fig. 5: Tiempo total de ejecución (tiempo de ejecución de los padres (azul), tiempo de redimensionado (naranja) y tiempo de ejecución de los hijos (verde)) para una estrategia Compact y *FactorS* constante. NP indica el total de procesos padres y NC los procesos hijos.

Las Figuras 6 y 7 comparan el tiempo total de ejecución de la misma aplicación que en la prueba anterior pero con un valor constante de *FactorS* e igual a 1 (sin escalabilidad). En estos experimentos, AR ofrece un mejor rendimiento cuando se trata de dos procesos padre. A su vez, se prefiere SR para más procesos padre. El punto de inflexión en la Figura 7 aparece cuando se reduce de 10 padres a 2 hijos, donde AR sigue siendo mejor, mientras que en la Figura 6, AR es mejor hasta la reducción de 20 a 2. Esto ocurre porque AR crea un hilo más por cada padre y al mismo tiempo también se crean los hijos. Por lo tanto, como los nodos sólo tienen 20 núcleos, se produce un problema de *oversubscription* y el rendimiento se reduce. A partir de esos puntos de inflexión los problemas derivados del *oversubscription* crea un mayor sobrecoste que el beneficio de AR.

Las Figuras 8 y 9 comparan el tiempo consumido en el procedimiento de redimensionado, considerando la creación de procesos y la redistribución de datos, con un *FactorS* para un speedup ideal y comparando las estrategias Spread y Compact. Estos datos muestran que la configuración que utiliza comunicaciones síncronas es siempre más rápida, sin embargo, la diferencia es muy pequeña y es importante recordar que la versión asíncrona está realizando tareas de cálculo y comunicación relacionadas con las iteraciones al mismo tiempo.

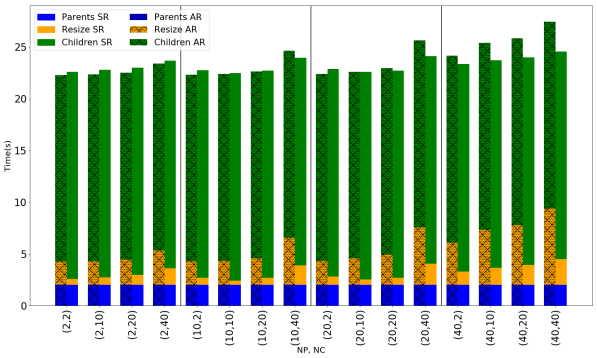

Fig. 6: Tiempo total de ejecución (tiempo de ejecución de los padres (azul), tiempo de redimensionado (naranja) y tiempo de ejecución de los hijos (verde)) para una estrategia Spread y *FactorS* constante. NP indica el total de procesos padres y NC los procesos hijos.

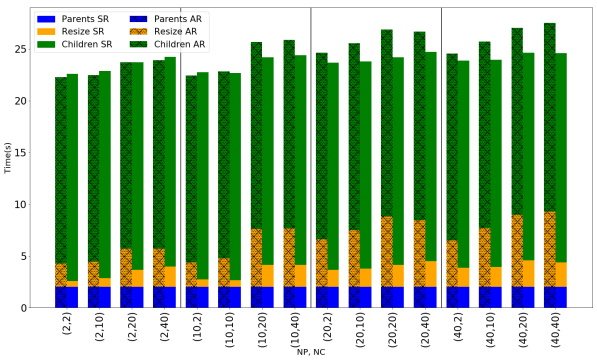

Fig. 7: Tiempo total de ejecución (tiempo de ejecución de los padres (azul), tiempo de redimensionado (naranja) y tiempo de ejecución de los hijos (verde)) para una estrategia Compact y *FactorS* constante. NP indica el total de procesos padres y NC los procesos hijos.

También puede observarse que la creación de procesos (*Spawn*) ocupa una parte importante del tiempo total de redimensionado aún cuando se está redistribuyendo un 1GB de datos.

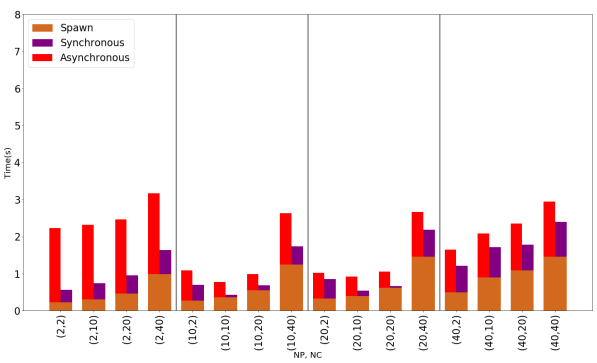

Fig. 8: Tiempo de redimensionado (generación de procesos y redistribución de datos) para una estrategia Spread y *FactorS* para un speedup ideal. NP indica el total de procesos padres y procesos hijos NC.

Las Figuras 10 y 11 muestran los mismos resultados que el caso anterior pero utilizando un *FactorS*. En este caso, los resultados son similares a los mostrados en las Figuras 8 y 9, pero la diferencia entre la configuración SR y AR es aún mayor. Esto

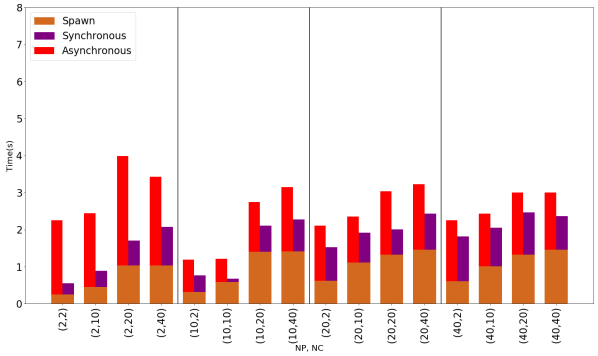

Fig. 9: Tiempo de redimensionado (generación de procesos y redistribución de datos) para una estrategia Compact y *FactorS* para un speedup ideal. NP indica el total de procesos padres y procesos hijos NC.

ocurre porque después de la finalización efectiva de las comunicaciones, AR necesita esperar hasta que la iteración en ejecución finalice completamente, aumentando T_{AR}. Este problema está directamente relacionado con T_c, siendo mayor cuando T_c es también mayor que T_{AR}.

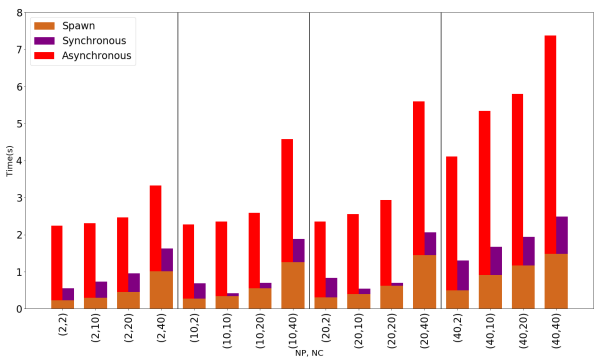

Fig. 10: Tiempo de redimensionado (generación de procesos y redistribución de datos) con la estrategia Spread y *FactorS* constante. NP indica el total de procesos padres y NC los procesos hijos.

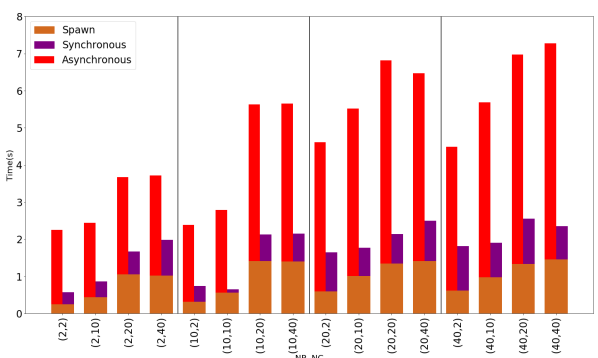

Fig. 11: Tiempo de redimensionado (generación de procesos y redistribución de datos) con la estrategiaa Compact y *FactorS* constante. NP indica el total de procesos padres y NC los procesos hijos.

A partir de estos resultados, los usuarios pueden decidir qué configuración obtendría mejores resultados para su aplicación maleable, o verificar si se mejoraría el tiempo de ejecución efectivo cuando una reasignación de recursos fuese propuesta por el RMS.

V. Conclusiones

Este trabajo presenta una aplicación sintética configurable capaz de modificar, en tiempo de ejecución, el número de procesos MPI en función de diferentes parámetros.

La principal contribución es proporcionar una herramienta fácil de utilizar que permita analizar diferentes escenarios para el desarrollo de aplicaciones maleables, prestando especial atención a la etapa de redistribución de datos, para realizar el redimensionado de una aplicación de estas características. Además, esta herramienta es capaz de recoger métricas de rendimiento que ayudan a elegir la mejor alternativa para completar la reconfiguración.

La primera tarea de extensión de la aplicación sintética será utilizar herramientas de trazas para obtener el modelo de aplicaciones reales, y utilizarlos para configurar la herramienta para emular su comportamiento. A continuación, se crearán cargas de trabajo que permitan analizar el comportamiento de dichas aplicaciones cuando se aplica la maleabilidad. También se pretende utilizar un gestor de recursos consciente de las cargas de trabajo adaptables para confirmar cómo se comportan estas cargas de trabajo en el sistema.

El trabajo futuro se centrará también en ampliar la aplicación permitiendo al usuario indicar un coste computacional y una función que defina su comportamiento, a partir de los cuales se puedan obtener diferentes T_{it} para cada iteración. Además, se estudiará la posibilidad de indicar diferentes valores de SDR y ADR para cada redimensionado en la misma ejecución, puesto que en las aplicaciones reales se pueden manejar diferentes cantidades de datos en distintos momentos de su ejecución. Otra mejora importante será añadir la posibilidad de utilizar otras distribuciones de datos entre los procesos. Por último, la aplicación sintética puede extenderse fácilmente para imitar comportamientos no iterativos, como es el caso del modelo productor-consumidor.

Agradecimientos

El presente trabajo ha sido subvencionado por el proyecto PID2020-113656RB-C21 financiado por MCIN/AEI/10.13039/501100011033 y por el proyecto UJI-B2019-36 financiado por la Universitat Jaume I. El trabajo del investigador S. Iserte ha sido subvencionado por la ayuda postdoctoral APOSTD/2020/026, y el trabajo del investigador I. Martín-Álvarez fue subvencionado por la ayuda predoctoral ACIF/2021/260, ambas ayudas financiadas por el Gobierno Autonómico Valenciano y por la European Social Funds.

Referencias

[1] Sergio Iserte, Rafael Mayo, Enrique S. Quintana-Ortí, Vicenç Beltran, and Antonio J. Peña, "DMR API: Improving Cluster Productivity by Turning Applications into Malleable," *Parallel Computing*, vol. 78, pp. 54–66, oct 2018.

[2] Gonzalo Martín, David E. Singh, Maria-Cristina Marinescu, and Jesús Carretero, "Enhancing the Performance of Malleable MPI Applications by Using Performance-

aware Dynamic Reconfiguration," *Parallel Computing*, vol. 46, pp. 60–77, jul 2015.

[3] Chao Huang, Orion Lawlor, and L. V. Kalé, "Adaptive MPI," in *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2003), LNCS 2958*, College Station, Texas, October 2003, pp. 306–322.

[4] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield, "Live Migration of Virtual Machines," in *Proceedings of the 2nd Conference on Symposium on Networked Systems Design and Implementation - Volume 2*, USA, 2005, NSDI'05, p. 273–286, USENIX Association.

[5] 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*, Dror G. Feitelson and Larry Rudolph, Eds., Berlin, Heidelberg, 1996, pp. 224–243, Springer Berlin Heidelberg.

[6] S. S. Vadhiyar and J. J. Dongarra, "SRS - A Framework for Developing Malleable and Migratable Applications for Distributed Systems," *Parallel Processing Letters*, vol. 2, pp. 291–312, 2002.

[7] 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.

[8] 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.

[9] J. S. Plank, M. Beck, G. Kingsley, and K. Li, "Libckpt: Transparent Checkpointing under Unix," in *Usenix Winter Technical Conference*, January 1995, pp. 213–223.

[10] 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.

[11] Rajesh Sudarsan and Calvin J. Ribbens, "ReSHAPE: a Framework for Dynamic Resizing and Scheduling of Homogeneous Applications in a Parallel Environment," in *Proceedings of the International Conference on Parallel Processing*, 2007.

[12] R. Sudarsan and C.J. Ribbens, "Scheduling Resizable Parallel Applications," in *International Symposium on Parallel & Distributed Processing*. may 2009, IEEE.

[13] 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.

[14] Laxmikant V. Kale and Sanjeev Krishnan, "CHARM++: a Portable Concurrent Object Oriented System Based on C++," *ACM SIGPLAN Notices*, vol. 28, no. 10, pp. 91–108, oct 1993.

[15] S. Iserte, R. Mayo, E.S. Quintana-Orti, and A.J. Pena, "DMRlib: Easy-coding and Efficient Resource Management for Job Malleability," *IEEE Transactions on Computers*, 2020.

[16] Isaías Comprés, Ao Mo-Hellenbrand, Michael Gerndt, and Hans Joachim Bungartz, "Infrastructure and API Extensions for Elastic Execution of MPI Applications," in *ACM International Conference Proceeding Series*, New York, New York, USA, 2016, vol. 25-28-Sept, pp. 82–97, ACM Press.

[17] Mohak Chadha, "Adaptive Resource-Aware Batch Scheduling for HPC systems," M.S. thesis, Technische Universität München, Germany, 2019.

[18] Mohak Chadha, Jophin John, and Michael Gerndt, "Extending SLURM for Dynamic Resource-Aware Adaptive Batch Scheduling," 2021.

[19] Jose I. Aliaga, Maribel Castillo, Sergio Iserte, Iker Martín-Álvarez, and Rafael Mayo, "A survey on malleability solutions for high-performance distributed computing," *Applied Sciences*, vol. 12, no. 10, 2022.

[20] Markus Wittmann, Georg Hager, Thomas Zeiser, and Gerhard Wellein, "Asynchronous MPI for the Masses," 02 2013.

[21] Sergio Iserte, Héctor Martínez, Sergio Barrachina, Maribel Castillo, Rafael Mayo, and Antonio J Peña, "Dynamic Reconfiguration of Noniterative Scientific Applications," *The International Journal of High Performance Computing Applications*, p. 109434201880234, sep 2018.

[22] Rajesh Sudarsan, Calvin J. Ribbens, and Diana Farkas, "Dynamic Resizing of Parallel Scientific Simulations: A Case Study Using LAMMPS," in *Computational Science – ICCS*, 2009, vol. 5544, pp. 175–184.

[23] Sergio Iserte and Krzysztof Rojek, "An Study of the Effect of Process Malleability in the Energy Efficiency on GPU-based Clusters," *The Journal of Supercomputing*, pp. 1–20, oct 2019.

[24] 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*, 2015, pp. 429–438.