Instituto de Computación, Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay

PROYECTO DE GRADO

Planificación de procesos en sistemas heterogéneos utilizando hwloc

Diego Regueira
Abril de 2016

Tutores:
Sergio Nesmachnow
Santiago Iturriaga

Planificación de procesos en sistemas
heterogéneos utilizando hwloc

Diego Regueira

Proyecto de Grado

Instituto de Computación - Facultad de Ingeniería
Universidad de la República

Montevideo, Uruguay, Abril de 2016

Planificación de Procesos en Sistemas Heterogéneos utilizando Hwloc

Resumen

Este trabajo presenta la aplicación de heurísticas de optimización combinatoria para la resolución del problema de la planificación por afinidad en sistemas de computación heterogénea multinúcleo. La planificación por afinidad es una técnica que permite asignar procesos a recursos de cómputo específicos, seleccionados estratégicamente de acuerdo a ciertos criterios de eficiencia. Se aborda la utilización de heurísticas de optimización combinatoria para resolver el problema de planificación, específicamente, de tres heurísticas: Ávida, Búsqueda en Escalada y Búsqueda Local Iterada. Se evalúan los algoritmos implementados, presentando un estudio comparativo entre los mismos y una heurística convencional, haciendo foco sobre las calidad de las soluciones y la eficiencia computacional. Este estudio demuestra que la aplicación de los algoritmos de planificación propuestos permite obtener soluciones precisas en forma eficiente, computando planificaciones de mayor calidad en comparación a las obtenidas mediante una estrategia convencional que no considera afinidades.

Palabras clave: Planificación, Computación Heterogénea, Heurística, Metaheurística, Optimización Combinatoria, HwLoc.

Índice general

1. Introducción **11**

2. Planificación con afinidades en sistemas multinúcleo **14**
2.1. Descripción del problema 14
2.2. Formulación matemática 15
2.3. Técnicas de planificación por afinidad 16
2.3.1. Planificación por afinidad de caché 16
2.3.2. Planificación de bucles paralelos 17
2.3.3. Planificación dinámica autoajustable 21
2.4. Resumen 23

3. Heurísticas de optimización combinatoria **24**
3.1. Optimización combinatoria 24
3.2. Problemas NP-difíciles 25
3.3. Heurística Ávida 26
3.4. Búsqueda en Escalada 26
3.5. Búsqueda Local Iterada 28
3.6. Resumen 29

4. Caracterización de aplicaciones paralelas **30**
4.1. Clasificación de aplicaciones paralelas 30
4.1.1. Topología plana 30
4.1.2. Topología dirigida por la aplicación 32
4.1.3. Topología jerárquica 33
4.2. Resumen 35

5. Caracterización de sistemas multinúcleo **36**
5.1. Hardware Locality 36
5.2. Benchmarks 41
5.3. Resumen 42

6. Heurísticas de optimización para planificación con afinidades 43
6.1. Mecanismos de caracterización 43
6.2. Representación de instancias del problema 44
6.3. Representación de soluciones 45
6.4. Heurística Ávida para la planificación por afinidad 46
6.5. Búsqueda en Escalada para la planificación por afinidad 47
6.6. Búsqueda Local Iterada para la planificación por afinidad . . . 48
6.7. Resumen 49

7. Análisis experimental 50
7.1. Plataforma de ejecución 50
7.2. Instancias de evaluación 51
7.2.1. Arquitecturas de cómputo 51
7.2.2. Aplicaciones paralelas 54
7.2.3. Conjunto de instancias de evaluación 58
7.3. Heurística de línea base 61
7.4. Calibración paramétrica 62
7.5. Resultados numéricos 63
7.5.1. Resultados de los algoritmos de planificación 63
7.5.2. Resultados comparativos 67
7.5.3. Comparación con cotas inferiores 70
7.5.4. Análisis de la eficiencia computacional 73

8. Conclusiones y trabajo futuro 75
8.1. Conclusiones 75
8.2. Trabajo futuro 76

Bibliografía 78

Índice de figuras

1.1. Sistema de cómputo real integrado por dos procesadores de cuatro núcleos cada uno y dos módulos de memoria, distribuidos en dos nodos NUMA. 12

2.1. Distribución de 16 procesadores en 4 grupos de CAFS. 20
2.2. Distribución de iteraciones entre 16 procesadores bajo HAFS. 21

3.1. Comportamiento de la técnica Búsqueda en Escalada en un problema de minimización. 27
3.2. Comportamiento de la técnica Búsqueda Local Iterada en un problema de minimización. 28

4.1. Topología plana en su forma convencional. 31
4.2. Topología plana con comunicación entre esclavos. 31
4.3. Topología plana con sincronización entre esclavos. 32
4.4. Topología dirigida por la aplicación con particionamiento por filas y sincronización de a tres procesos. 32
4.5. Topología dirigida por la aplicación con particionamiento por columnas y sincronización de a tres procesos. 33
4.6. Topología dirigida por la aplicación con sincronización de procesos por nivel. 33
4.7. Topología jerárquica con sincronización entre hermanos. 34
4.8. Topología jerárquica con sincronización por niveles. 34

5.1. Representación interna en HwLoc de una topología de cinco niveles. 39
5.2. Topología obtenida con HwLoc mediante la utilización del comando lstopo. 41

6.1. Ejemplo de planificación de una conjunto de procesos sobre un conjunto de núcleos. 45

7.1. Topología generada con HwLoc de la arquitectura a_1. 52

7.2. Topología generada con HwLoc de la arquitectura a_2. 52
7.3. Topología generada con HwLoc de la arquitectura a_3. 53
7.4. Instancia de 6 procesos y sincronizaciones de a 2 esclavos. . . . 58
7.5. Instancia de 8 procesos y sincronizaciones entre todos los procesos esclavos. 58
7.6. Instancia de 8 procesos y sincronizaciones por nivel. 59
7.7. Instancia de 12 procesos y sincronizaciones por nivel. 59
7.8. Instancia de 12 procesos y sincronizaciones entre procesos de un mismo nivel. 60
7.9. Instancia de 24 procesos y sincronizaciones entre procesos hermanos. 60
7.10. Mejora porcentual obtenida respecto a la heurística de línea base mediante las heurísticas de optimización agrupadas por tipo de instancia. 69
7.11. Margen de mejora potencial calculados sobre los 3 tipos de instancia respecto a las cotas inferiores. 72

Índice de tablas

7.1. Costos de interacción (comunicaciones y sincronizaciones) entre dos procesos para las arquitecturas consideradas en el análisis experimental. 54
7.2. Conjunto de instancias de evaluación. 61
7.3. Costo de las soluciones computadas para instancias de tipo TC. 64
7.4. Costo de las soluciones computadas para instancias de tipo FT. 65
7.5. Costo de las soluciones computadas para instancias de tipo OR. 66
7.6. Mejora porcentual alcanzada por las heurísticas de optimización respecto a la de línea base. 68
7.7. Márgenes de mejora potencial agrupados por arquitectura y cantidad de procesos. 71
7.8. Eficiencia computacional (μs). 73

Índice de algoritmos

1. Algoritmo AFS. 18
2. Algoritmo SADS. 22
3. Algoritmo Ávido. 26
4. Algoritmo de Búsqueda en Escalada. 27
5. Algoritmo de Búsqueda Local Iterada. 28
6. Planificador Ávido para la planificación por afinidad. 46
7. Planificador Búsqueda en Escalada para la planificación por afinidad. 47
8. Planificador Búsqueda Local Iterada para la planificación por afinidad. 48
9. Aplicación de transferencia de calor. 55
10. Aplicación de flujo de trabajo. 56
11. Aplicación de ordenamiento rápido. 57
12. Planificador Round Robin. 62

Capítulo 1

Introducción

Los sistemas de computación heterogénea (en inglés, Heterogeneous Computing o HC) se han convertido en la principal opción a la hora de resolver problemas complejos que surgen en distintas áreas de aplicación [1]. Esta tendencia se ha debido, principalmente, al incremento de la capacidad de cómputo y al gran avance en las tecnologías de interconexión y redes. Los sistemas heterogéneos están constituidos por un conjunto coordinado de recursos de cómputo, interconectados entre sí y con capacidad de cómputo variable. Un aspecto clave al utilizar sistemas de este tipo radica en hallar una estrategia de planificación que permita ejecutar eficientemente un conjunto de procesos. El objetivo de esta planificación, consiste en asignar procesos a recursos de cómputo optimizando ciertos criterios de eficiencia, habitualmente asociados al tiempo total de ejecución o la utilización de recursos. Los problemas de planificación en sistemas de cómputo han sido ampliamente tratados en el área de Investigación Operativa, y se han propuesto diversos métodos para encontrar planificaciones precisas en tiempos razonables [2, 3]. En su formulación clásica, los problemas de este tipo asumen ambientes de cómputo homogéneos. Es en la década de 1990 cuando el problema de planificación en sistemas heterogéneos se vuelve de gran importancia para la comunidad científica, debido a la popularización de este tipo de infraestructura [4]. La planificación de procesos en sistemas heterogéneos es un problema de optimización combinatoria NP-difícil [5]. Por lo tanto, al enfrentarnos a este tipo de problemas, la aplicabilidad de métodos exactos de resolución se encuentra limitada a instancias de tamaño reducido, por el enorme tiempo y consumo de recursos computacionales que demandan la resolución de instancias de tamaño realista. Las técnicas heurísticas y metaheurísticas [6] son la única alternativa viable para resolver este tipo de problemas de planificación, al permitir calcular soluciones eficientes en tiempos de ejecución reducidos, aún para instancias del problema de gran dimensión. La heterogeneidad presente

en este tipo de sistemas, provoca que un proceso ejecute más eficientemente en ciertas unidades de cómputo. Factores como la localidad de datos y las latencias de comunicación entre recursos computacionales establecen una relación de afinidad entre los procesos a ejecutar y las unidades de procesamiento disponibles. Actualmente, los sistemas de computación multinúcleo se organizan bajo arquitecturas de tipo NUMA (en inglés, Non-Uniform Memory Access) donde la memoria compartida se encuentra físicamente distribuida entre los procesadores y los accesos a memoria varían de acuerdo a si son locales (rápidos) o remotos (lentos) [7]. La complejidad creciente que presentan este tipo de sistemas impacta significativamente sobre el tiempo de ejecución, por lo que es de gran importancia contar con planificadores que consideren las afinidades entre procesos y recursos de computo al momento de planificar. En la Figura 1.1 se observa la organización interna de un sistema de cómputo real compuesto por dos nodos NUMA, cada nodo cuenta con una memoria local y un procesador, cada procesador contiene cuatro núcleos y un memoria caché organizada en tres niveles (L1, L2 y L3).

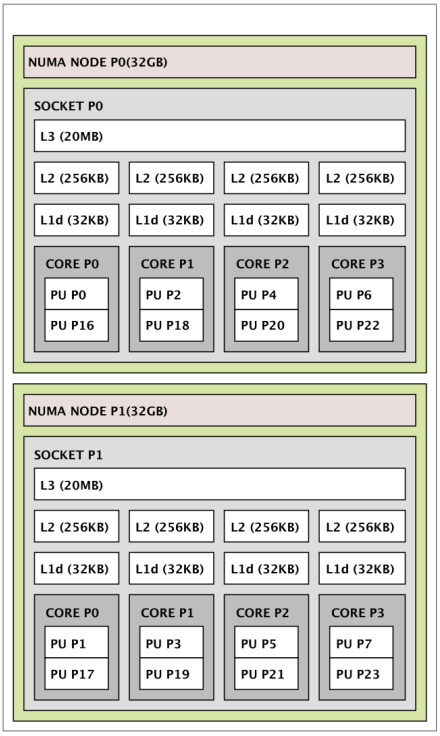

Figura 1.1: Sistema de cómputo real integrado por dos procesadores de cuatro núcleos cada uno y dos módulos de memoria, distribuidos en dos nodos NUMA.

El problema de la planificación de procesos por afinidad ha sido objeto de estudio en numerosos trabajos. Algunos de ellos proponen resolver el problema considerando afinidades de memoria caché locales, mejorando el desempeño mediante la reutilización de datos previamente almacenados en memoria caché [8]. Otros estudios se centran específicamente en la planificación de bucles paralelos [9–12], plantean políticas para el particionamiento y migración de iteraciones, buscando maximizar la reutilización de memoria y balancear la carga de los procesadores. En este trabajo se propone el estudio e implementación de algoritmos para la planificación de procesos por afinidad en sistemas de computación heterogénea multinúcleo, utilizando la herramienta HwLoc. Las principales contribuciones de la investigación reportada en este trabajo incluyen:

- La descripción del problema de planificación por afinidad enfocado en la minimización de los tiempos de comunicación y sincronización entre procesos.
- El diseño e implementación de tres algoritmos heurísticos de optimización combinatoria (Ávida, Búsqueda en Escalada y Búsqueda Local Iterada) para resolver el problema de planificación.
- La realización de un análisis experimental sobre los tres algoritmos de planificación, aplicados sobre un conjunto de instancias reales del problema considerado en este trabajo.

Este proyecto de grado se enmarca en el contexto del proyecto STIC-Amsud de cooperación entre la Universidad de la República (Uruguay), INRIA (Francia) y la Universidad Nacional de San Luis (Argentina). El contenido de este informe ha sido estructurado del modo que se describe a continuación: el Capítulo 2 plantea el problema de planificación por afinidad en sistemas multinúcleo, su formulación matemática y la presentación de diferentes técnicas de planificación por afinidad. En el Capítulo 3 se presenta un conjunto de heurísticas de optimización combinatoria aplicables al problema de planificación. Se describen mecanismos de caracterización para aplicaciones paralelas e infraestructuras de cómputo en los Capítulos 4 y 5, respectivamente. En el Capítulo 6 se proponen tres heurísticas de optimización combinatoria para la resolución del problema de planificación por afinidad en sistemas de cómputo multinúcleo. Un análisis experimental de los algoritmos y la discusión los principales resultados obtenidos se presentan en el Capítulo 7. Por último, las conclusiones sobre el proyecto y las principales lineas de trabajo futuro se presentan en el Capítulo 8.

Capítulo 2

Planificación con afinidades en sistemas multinúcleo

Este capítulo presenta la descripción del problema de planificación por afinidad en arquitecturas multinúcleo, su formulación matemática y un conjunto de algoritmos propuestos en la literatura relacionada para la planificación de procesos que consideran diferentes tipos de afinidad.

2.1. Descripción del problema

Un sistema de computación multinúcleo consiste en un equipo multiprocesador de memoria compartida, donde cada procesador cuenta con uno o varios núcleos capaces de ejecutar procesos de forma paralela [13]. La memoria RAM se encuentra físicamente distribuida entre los procesadores, esto provoca que los accesos a memoria varíen de acuerdo a si son locales (rápidos) o remotos (lentos). Estas arquitecturas pueden ser consideradas como un caso especial de sistema de computación heterogénea ya que diferentes núcleos son capaces de computar una misma tarea a diferentes velocidades de procesamiento. Sobre este tipo de sistemas se pretende ejecutar paralelamente un conjuntos de procesos, los cuales cooperan entre sí a través de la ejecución de operaciones de comunicación y sincronización, con el propósito de resolver un determinado problema. Cada proceso es considerado una unidad atómica de procesamiento, por lo tanto, no puede ser dividido en partes más pequeñas ni tampoco pude ser interrumpido una vez iniciado su procesamiento. El problema de optimización consiste en realizar la asignación de procesos a núcleos (es decir, planificar), minimizando el tiempo utilizado para efectuar las comunicaciones y sincronizaciones entre procesos, considerando ciertas características presentes en los sistema multinúcleo y las aplicacio-

nes paralelas. Estas características están relacionadas con la frecuencia de interacción entre procesos, la localidad de datos y los accesos a recursos específicos (memoria compartida, memoria caché, unidades de procesamiento, etc.), que pueden influir negativamente en el costo temporal de las comunicaciones y sincronizaciones, sobre todo en arquitecturas NUMA, donde el acceso a memoria no es uniforme [7].

2.2. Formulación matemática

La formulación matemática del problema considera los elementos presentados a continuación:

- Un sistema multinúcleo compuesto por un conjunto de núcleos $N = \{n_1, \ldots, n_a\}$.
- Una colección de procesos o tareas $T = \{t_1, \ldots, t_b\}$ a ejecutar.
- Una función que contabiliza las comunicaciones $C : T \times T \to \mathbb{N}^+$, donde $C(t_i, t_j)$ indica la cantidad de comunicaciones entre dos procesos t_i y t_j con $i, j \leqslant b$.
- Una función que contabiliza las sincronizaciones $S : T \times T \to \mathbb{N}^+$, donde $S(t_i, t_j)$ reporta la cantidad sincronizaciones entre un par de procesos t_i y t_j con $i, j \leqslant b$.
- Una función de costos de comunicación $CC : N \times N \to \mathbb{R}^+$, donde $CC(n_i, n_j)$ retorna el tiempo requerido para la comunicación entre dos núcleos n_i y n_j con $i, j \leqslant a$.
- Una función de costos de sincronización $CS : N \times N \to \mathbb{R}^+$, donde $CS(n_i, n_j)$ marca el tiempo requerido para la sincronización entre dos núcleos n_i y n_j con $i, j \leqslant a$.
- Una función de planificación $f : T \to N$, donde el proceso t_j será ejecutado en el núcleo n_i si y sólo si $f(t_j) = n_i$ con $j \leqslant b$ y $i \leqslant a$.

El objetivo consiste en encontrar la función f de planificación que minimice la función de costo z, que contabiliza el tiempo total utilizado para efectuar las operaciones de comunicación y sincronización entre procesos, de acuerdo a la Ecuación 2.1.

$$z = \sum_{t_i \in T} \sum_{t_j \in T} C(t_i, t_j) \times CC(f(t_i), f(t_j)) + S(t_i, t_j) \times CS(f(t_i), f(t_j)) \quad (2.1)$$

2.3. Técnicas de planificación por afinidad

Esta sección presenta un conjunto de técnicas elaboradas en diferentes trabajos, que plantean resolver la planificación de procesos sobre sistemas heterogéneos siguiendo estrategias basadas en diferentes tipos de afinidad.

2.3.1. Planificación por afinidad de caché

En sistemas heterogéneos la migración de procesos reduce la efectividad de la memoria caché, los procesos deben regenerar su estado en memoria caché cada vez que son asignados a un nuevo núcleo o cuando la información generada en una ejecución previa fue desplazada por otro proceso. Estos dos factores ocasionan un incremento en el número de fallos de caché, lo cual aumenta el tiempo total de ejecución. Torrellas et al. [8] plantean una técnica de planificación por afinidad de caché que propone disminuir el tiempo de ejecución mediante el incremento en la tasa de aciertos en memoria caché, procurando que los procesos reutilicen de forma más frecuente los datos previamente almacenados en caché . Partiendo desde un esquema de planificación por prioridades, donde los procesos son ordenados y ejecutados de acuerdo a una prioridad numérica, establecida de forma inversamente proporcional a la utilización de unidades de cómputo (si un proceso hace un alto uso de procesamiento, su prioridad de ejecución será más baja). La técnica propone modificar el esquema de forma que las asignaciones de recursos a procesos se realicen considerando las afinidades de caché existentes en el sistema. Para ello, se plantea aumentar temporalmente las prioridades de aquellos procesos que resultan más atractivos de acuerdo a las afinidades de memoria caché, aplicando los siguientes criterios:

- Adicionar un factor constante a_p (por unidad de procesamiento) a las prioridades de los procesos que alguna vez han sido ejecutados en la unidad para la cual se está eligiendo un proceso. Esto produce una disminución en la migración de procesos entre las unidades de cómputo.
- Sumar otra constante a_t (por tiempo) a las prioridades de los procesos que han sido ejecutados más recientemente en la unidad para la cual se está seleccionado un proceso. Esto disminuye los desplazamiento de datos generados por los procesos en la memoria caché de los procesadores.

Ambos ajustes son aplicados con la finalidad de realizar la planificación en ese instante. Una vez realizada la selección del proceso a ejecutar, las prioridades son restablecidas mediante la remoción de los factores mencionados.

2.3.2. Planificación de bucles paralelos

Los métodos exhibidos a continuación presentan diferentes formas de planificar bucles paralelos sobre arquitecturas NUMA (en inglés, Non-Uniform Memory Access), considerando afinidades de memoria. La idea principal consiste en realizar una partición de las iteraciones en grupos, los cuales serán asignados a los procesadores intentando minimizar la sobrecarga generada por las comunicaciones, las sincronizaciones y el desbalance de cargas.

Planificación por afinidad

La técnica de planificación por afinidad (en inglés, Affinity Scheduling o AFS) diseñada por Markatos et al. [14] consiste en dividir un bucle de tamaño N en grupos de (N/P) iteraciones, donde P representa la cantidad de procesadores. Estos grupos de iteraciones serán asignados a los procesadores ociosos produciendo una mayor utilización de datos preexistentes en la memoria caché y en la memoria local. En caso de ocurrir un desbalance de cargas, se realiza un migración de iteraciones desde el procesador más cargado a uno de los procesadores que se encuentren ociosos en ese momento. Seguidamente se detallan las fases de planificación:

- Inicialización: En esta fase se realiza una partición de las iteraciones a procesar en grupos de (N/P) iteraciones. Luego, se asigna un grupo a cada procesador, de forma que el grupo g_i es asignado al procesador p_i.
- Ejecución: Durante la fase de ejecución el procesador p_i toma una fracción de $[1/P]$ iteraciones del grupo g_i para su posterior procesamiento, este mecanismo se repite hasta que el procesador termina de ejecutar la totalidad de iteraciones asignadas. Si ocurre un desbalance de carga, uno de los procesadores ociosos buscará entre los $(P-1)$ restantes al procesador más cargado. Luego, procederá a migrar una fracción de $[1/P]$ iteraciones desde el procesador más cargado hacia sí mismo, para su posterior ejecución.

A continuación, se presenta el pseudocódigo asociado a la técnica AFS en el Algoritmo 1, donde se describen la fase de inicialización, la cual es ejecutada al inicio en forma única y la fase de ejecución, la cual es ejecutada por cada procesador en forma continua.

Algoritmo 1 Algoritmo AFS.

```
1: for each p in 1..P do //fase de inicialización
2:     chunk ← N/P
3:     assign_iterations(p, chunk)
4: end for
5: while true do //fase de ejecución
6:     range ← get_local_iterations(1/P)
7:     if empty(range) then
8:         most_loaded_processor ← find_most_loaded_processor()
9:         if most_loaded_processor == nil then
10:            break
11:        end if
12:        range ← get_remote_iterations(most_loaded_processor, 1/P)
13:        if empty(range) then
14:            break
15:        end if
16:    end if
17:    execute(range)
18: end while
```

Planificación por afinidad modificada

AFS propone migrar pequeñas fracciones de iteraciones, por lo tanto las fases de migración deben ser realizadas frecuentemente, produciendo un deterioro del desempeño. La planificación por afinidad modificada (en inglés, Modified Affinity Scheduling o MAFS) propuesta por Fann et al. [10] plantea una política menos conservadora, que permite balancear la carga en forma más eficiente. Para ello, propone migrar $min([N_i/P], N_i^{most} - [N_i/P])$ iteraciones desde el procesador más cargado, donde N_i es el total de iteraciones a procesar en el tiempo t_i y N_i^{most} es el número de iteraciones a procesar por el procesador más cargado. La técnica consiste en dos fases:

- Inicialización: Coincide con la fase de inicialización descrita en AFS.
- Ejecución: Cada procesador obtiene una fracción de $[1/P]$ iteraciones para su posterior procesamiento, hasta que todas las iteraciones sean procesadas. Al detectarse un desbalance de carga, alguno de los procesadores ociosos buscará en los $(P-1)$ restantes al más cargado. Luego, a diferencia de lo que ocurre en AFS, procederá a migrar una fracción de $min([N_i/P], N_i^{most} - [N_i/P])$ iteraciones desde el procesador más cargado hacia sí mismo, para su posterior procesamiento.

Planificación por afinidad agrupada

El aumento en el número procesadores produce un deterioro en el desempeño de las técnicas de planificaciones de tipo AFS y MAFS, al aumentar la cantidad de procesadores el proceso de búsqueda del procesador más cargado se vuelve más pesado. Wang et al. [11] propone una técnica de planificación por afinidad agrupada (en inglés, Clustered Affinity Scheduling o CAFS) que plantea atacar este problema realizando una partición de los procesadores en grupos, donde cada grupo contiene G procesadores, siendo $G = P/\sqrt{P}$ la cantidad de grupos. La distribución de procesadores a través de los G grupos se realiza de la siguiente manera: los procesadores p_1,p_2,...,p_G son asignados a los grupos g_1,g_2,...,g_G respectivamente, luego los procesadores p_{G+1},p_{G+2},...,p_{2G} son asignados a los grupos g_G,g_{G-1},...,g_1 en forma respectiva, este proceso se repite hasta que todos los procesadores estén asignados. En caso de producirse un desbalance de carga, se procede de forma similar a la descrita en el método AFS con la diferencia que el procesador ocioso realiza la búsqueda del procesador más cargado dentro de su grupo y no en totalidad de los procesadores restantes, evitando realizar lecturas remotas y aumentando la tasa de accesos exitosos a memoria. Seguidamente, se presentan las fases de inicialización y ejecución aplicadas en esta técnica:

- Inicialización: Es idéntica a la fase de inicialización descrita anteriormente para la técnica AFS.
- Ejecución: Al igual que en el método AFS cada procesador toma una fracción de $[1/P]$ iteraciones para luego procesarlas, esto se repite hasta finalizar la ejecución de todas las iteraciones pendientes. En caso de ocurrir un desbalance de carga, uno de los procesadores ociosos buscará al procesador más cargado entre los $(G-1)$ procesadores restantes de su grupo. Seguidamente, el procesador ocioso migrará una fracción de $[1/P]$ iteraciones desde el procesador más cargado hacia sí mismo para su posterior procesamiento.

La Figura 2.1 presenta un ejemplo de CAFS, donde 16 fracciones de $[1/P]$ iteraciones son asignadas a 16 procesadores distribuidos entre 4 grupos.

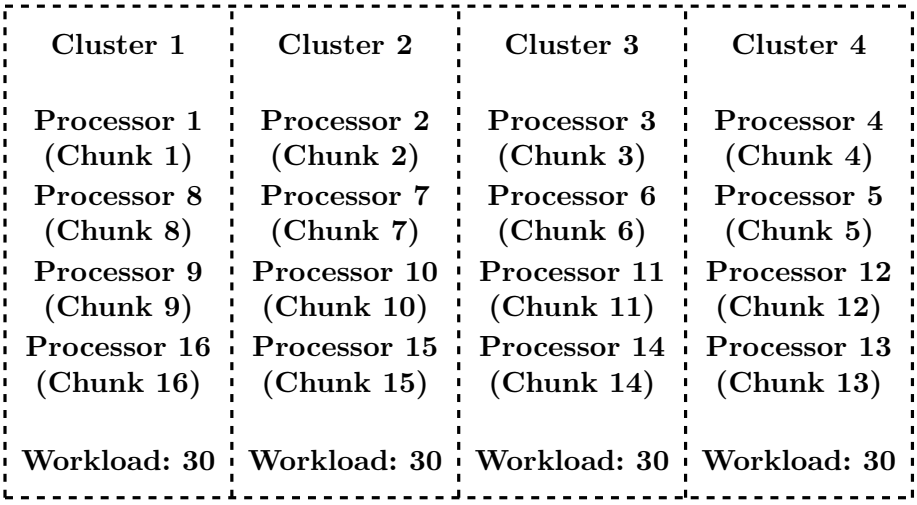

Figura 2.1: Distribución de 16 procesadores en 4 grupos de CAFS.

Planificación por afinidad jerárquica

La adopción de arquitecturas NUMA para la construcción de sistemas de memoria compartida provocó una disminución en la efectividad de las planificaciones AFS y MFS, dado que las mismas no fueron pensadas para arquitecturas donde los accesos de memoria no son uniformes y varían de acuerdo a si se realizan localmente (rápidos) o remotamente (lentos). Por otro parte, CAFS solo permite migraciones locales y no entre grupos, lo que va en desmedro del balance de cargas. La planificación por afinidad jerárquica (en inglés, Hierarchical Affinity Scheduling o HAFS) diseñada por Wang et al. [12] sugiere la implementación de una jerarquía de grupos de procesadores con el fin de modelar arquitecturas NUMA de forma jerárquica, donde cada procesador pertenece a un grupo local, el cual a su vez esta incluido en otro grupo de nivel superior, un súper grupo. En caso de producirse un desbalance, la búsqueda de el procesador más cargado se realizará en el cluster local y en caso de no encontrarse ahí se buscará en el súper grupo. Esta técnica permite reducir los accesos remotos a memoria y realizar un balance de cargas más efectivo. El planificador esta compuesto por estas dos fases:

1. Inicialización: Se realiza el mismo procedimiento que para la técnica AFS con la diferencia de que las iteraciones son distribuidas cíclicamente entre los grupos.

2. Ejecución: Cada procesador toma fracciones de $[1/P]$ iteraciones pendientes y las ejecuta hasta finalizar el procesamiento de todas ellas. En caso de ocurrir un desbalance de carga, uno de los procesadores ociosos

migrará $min(N_{local}/P_{local}, N_{cargado} - N_{local}/P_{local})$ iteraciones desde el procesador vecino más cargado en su grupo local, siendo P_{local} el tamaño del grupo local, N_{local} el acumulado de iteraciones pendientes en el grupo local y $N_{cargado}$ el número de iteraciones pendientes en el procesador más cargado del grupo local. Si todos sus vecinos se encuentran ociosos se aplica un mecanismo análogo al anteriormente descrito, que utiliza el súper grupo del procesador ocioso como área de búsqueda.

Seguidamente, en la Figura 2.2 se presenta un ejemplo de HAFS, en el cual 16 fracciones de $[1/P]$ iteraciones cada una son distribuidas en 4 grupos, compuestos por 4 procesadores cada uno.

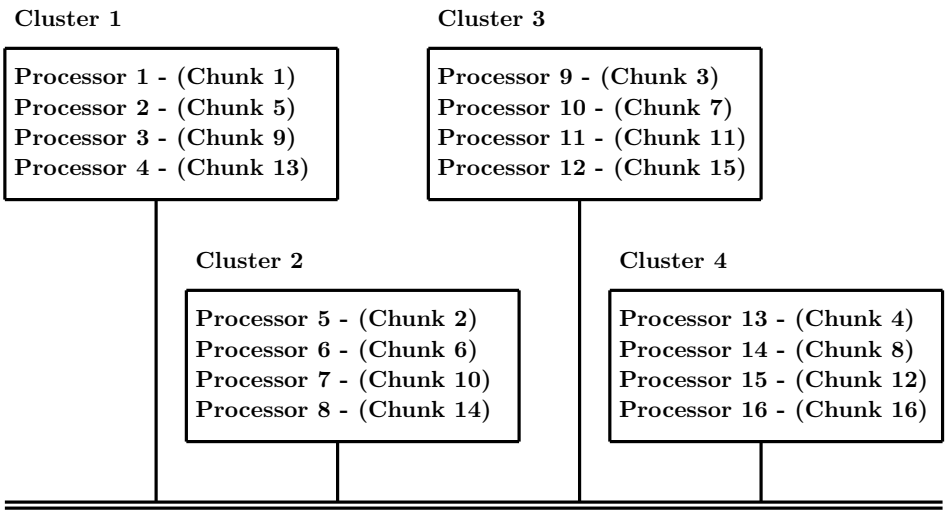

Figura 2.2: Distribución de iteraciones entre 16 procesadores bajo HAFS.

2.3.3. Planificación dinámica autoajustable

Hamidzadeh et al. [15] plantea una técnica de planificación dinámica autoajustable (en inglés, Self-Adjusting Dynamic Scheduling o SADS) que propone planificar un conjunto de procesos independientes mediante la superposición total de las etapas de planificación y ejecución, a través de la designación de uno de los procesadores como el encargado de ejecutar el algoritmo de planificación. La estrategia consiste en determinar planificaciones parciales y realizar la correspondiente asignación de recursos hasta que todos los procesos sean asignados. SADS plantea reajustar la duración de la fase de planificación de acuerdo al tiempo necesario para que el procesador que se encuentre más cargado en ese instante vuelva a estar disponible. Al finalizar

el tiempo preestablecido para la planificación, los procesos son asignados a las unidades de procesamiento de acuerdo a la planificación parcial realizada hasta el momento. El algoritmo de planificación consiste básicamente en realizar una búsqueda sobre el espacio de todas las posibles planificaciones. El cual es representado a través de un grafo, cada nodo simboliza una planificación parcial y cada vértice representa la extensión de la planificación. El planificador realiza una o más iteraciones, en las que el nodo de costo mínimo es expandido mediante el agregado de una nueva asignación. Esto se repite hasta que todos los procesos estén asignadas o se agote el tiempo de planificación. La función de costo utilizada para estimar el tiempo producido por una planificación se basa en los tiempos de ejecución de procesos y las afinidades de memoria entre procesos y unidades de cómputo [16]. El Algoritmo 2 que aparece a continuación presenta el mecanismo de planificación aplicado en esta técnica.

Algoritmo 2 Algoritmo SADS.

```
1: while not_solved(head(queue)) or scheduling_time > 0 do
2:     current_node ← head(queue)
3:     for each x in successors(current_node) do
4:         x.cost ← cost(current_node,x)
5:         queue ← insert(x,queue)
6:     end for
7:     queue ← sort_queue_by_cost(queue)
8: end while
9: if no_more_tasks(head(queue)) then
10:    announce_success
11: else
12:    if no_more_tasks(head(queue)) then
13:        assign_partial_schedule(current_node)
14:        SADS(remaining_task_set)
15:    else
16:        announce_failure
17:    end if
18: end if
```

2.4. Resumen

En este capítulo se describió el problema de planificación por afinidad en arquitecturas multinúcleo. Adicionalmente, se introdujo una formulación matemática del problema de planificación por afinidad, que considera la cantidad de operaciones de comunicación y sincronización entre cada par de procesos para modelar afinidades, así como también los costos temporales generados al efectuar estas operaciones sobre las distintas unidades de procesamiento. Finalmente, se reportó un breve estudio sobre un conjunto de técnicas que resuelven el problema de planificación considerando diferentes tipos de afinidad.

Capítulo 3

Heurísticas de optimización combinatoria

En este capítulo se realiza una breve introducción a la optimización combinatoria y a los problemas NP-difíciles. Seguidamente, se presentan las heurísticas Ávida (en inglés, Greedy Heuristic), Búsqueda en Escalada (en inglés, Hill Climbing) y Búsqueda Local Iterada (en inglés, Iterated Local Search), detallando su funcionamiento y sus principales características. Estas técnicas luego serán propuestas para la solución al problema de planificación de procesos por afinidad en sistemas multinúcleo.

3.1. Optimización combinatoria

La optimización combinatoria es una rama de la optimización matemática cuyo dominio se basa en problemas de optimización, donde el conjunto de posibles soluciones es discreto o se puede reducir a un conjunto discreto. Los problemas de optimización combinatoria consisten en obtener un elemento de un conjunto finito o infinito enumerable, que sea óptimo para la minimización o maximización de un criterio determinado. Formalmente, un problema de optimización combinatoria Π es representado como una tupla (S, f, Ω) donde S es el espacio de soluciones, f la función objetivo que tiene como domino a S y Ω es un conjunto de restricciones [17]. El conjunto de las soluciones factibles (también llamado espacio de búsqueda) S_Ω, está compuesto por aquellas soluciones que cumplen las restricciones Ω. Estos problemas pueden implicar la minimización o maximización de la función objetivo. Para el caso de una minimización, un problema de optimización combinatoria consiste en hallar una solución factible $s^* \in S_\Omega / f(s^*) \leq f(s), \forall s \in S_\Omega$.

3.2. Problemas NP-difíciles

Al momento de evaluar la complejidad computacional de un algoritmo se tiene en cuenta el número de operaciones en función del tamaño de la entrada. De esta forma, el análisis es independiente del lenguaje de programación y de la arquitectura del computador utilizados. Generalmente, se analiza la complejidad computacional para el peor caso, de esta manera se obtiene un cota superior asintótica. Sean $f, g : \mathbb{N}^+ \rightarrow \mathbb{R}^+$ se dice que la función $f(x)$ tiene orden $O(g(x))$ si existen x_0 y k tales que $0 \leq f(x) \leq kg(x); \forall x \geq x_0$. Se dice que un algoritmo tiene complejidad de orden polinomial si en el peor caso tiene un orden $O(g(x))$ para alguna función $g(x)$ polinomial [5]. De acuerdo a la teoría de NP-completitud [5] existen dos clases de problemas: los de clase P, que son aquellos problemas que pueden ser resueltos en tiempo polinomial en un computador determinístico y los de clase NP, que son aquellos problemas que pueden ser resueltos en tiempo polinomial por un computador no determinístico. Otra definición establece que la clase P contiene problemas que pueden resolverse de modo eficiente, mientras que la clase NP contiene problemas cuya solución puede verificarse de modo eficiente. Un problema se define como NP-difícil, si todo problema de la clase NP es reducible a él en tiempo polinomial. Aquellos problemas que pertenecen a la clase NP y ademas son NP-difíciles se denominan NP-completos. La pertenencia de un problema a la clase NP-difícil implica que no se conoce un algoritmo que permita resolverlo en tiempo polinomial. Por lo tanto, no se conoce una forma de resolver este tipo de problemas en tiempos razonables al considerarse instancias de tamaño creciente [18]. Frente a esta dificultad, surgen las técnicas heurísticas y metaheurísticas como herramientas para encontrar una solución aproximada a este tipo de problemas en tiempo razonable. Las técnicas heurísticas son métodos de resolución basados en procedimientos conceptualmente simples para encontrar soluciones de buena calidad (no necesariamente la solución óptima) a problemas difíciles, de un modo sencillo y eficiente. Por otra parte, las técnicas metaheurísticas son estrategias iterativas de alto nivel que guían una heurística subordinada, explorando y explotando el espacio de soluciones con la finalidad de obtener buenas soluciones a problemas difíciles. En particular, el problema de planificación de procesos en sistemas multinúcleo es un problema de optimización combinatoria NP-difícil [5], por lo cual, los métodos exactos son solo aplicables a instancias reducidas del problema. A continuación, se presentan tres heurísticas aplicables a problemas de optimización combinatoria NP-difíciles.

3.3. Heurística Ávida

La Heurística Ávida está catalogada como una heurística de tipo constructiva, la técnica se basa en una idea simple que consiste en generar soluciones partiendo de una solución parcial, que puede ser vacía ó determinada bajo algún mecanismo, por ejemplo, de forma aleatoria [17]. La solución es construida iterativamente tomando decisiones localmente optimas, incorporando el componente de menor costo heurístico (de acuerdo a la información actual) hasta obtener una solución completa ó alcanzar algún criterio de parada. Para ciertos problemas con estructura particular, la solución construida es óptima, aunque en general esto no ocurre. La heurística ávida es sencilla y de fácil aplicación, en contrapartida, la estrategia implica la toma de decisiones en forma local sin tener en cuenta los efectos que estas decisiones puedan tener en el futuro. El Algoritmo 3 presenta el pseudocódigo de la técnica ávida en su forma más general.

Algoritmo 3 Algoritmo Ávido.

```
1: s ← fijar una solución parcial
2: while s no sea completa y no se alcance criterio de parada do
3:     componente ← elegir componente de menor costo
4:     s ← agregar componente a la solución
5: end while
6: return s
```

3.4. Búsqueda en Escalada

La Búsqueda en Escalada es una heurística de mejora iterativa [19], perteneciente a la familia de heurísticas de Búsqueda Local [20]. El método comienza con una solución inicial (generalmente determinada de forma aleatoria o mediante una heurística constructiva). Luego la solución inicial es mejorada mediante un proceso iterativo de búsqueda que consiste en la sustitución de la solución actual por una solución vecina que mejore la función objetivo. La búsqueda se detiene cuando todos los vecinos son peores que la solución actual, alcanzando de esta forma un óptimo local (alcanzar el óptimo global depende de la solución inicial), como se aprecia en la Figura 3.1.

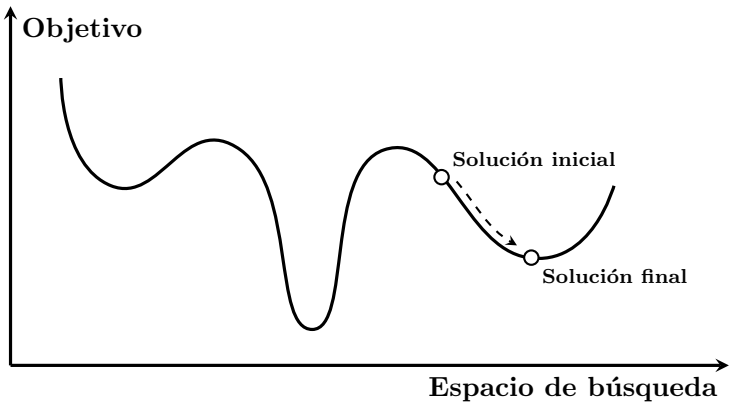

Figura 3.1: Comportamiento de la técnica Búsqueda en Escalada en un problema de minimización.

El vecindario de una solución s, denotado como $N(s)$, es el conjunto de soluciones que se pueden construir a partir de s aplicando un operador de movimiento, que deberá ser definido según el problema. La solución vecina elegida puede ser la mejor entre todos los vecinos (exploración exhaustiva) o puede ser la primera solución vecina que mejore la solución actual (exploración parcial). La variante que realiza una exploración parcial para seleccionar una mejor solución se denomina Escalada Simple y la que aplica exploración exhaustiva Escalada por Máxima Pendiente. Generalmente, este tipo de técnicas obtienen mejores resultados que las heurísticas constructivas, aunque, tienden a estancarse en óptimos locales y requieren de un mayor esfuerzo computacional. Seguidamente, se presenta el pseudocódigo de la técnica de búsqueda en escalada en el Algoritmo 4.

Algoritmo 4 Algoritmo de Búsqueda en Escalada.

```
1: s ← fijar una solución inicial
2: while no se alcance criterio de parada do
3:     N(s) ← generar vecinos candidatos
4:     if no existen vecinos mejores que s then
5:         return s
6:     end if
7:     s ← mejor vecino
8: end while
9: return s
```

3.5. Búsqueda Local Iterada

La Búsqueda Local Iterada [21] es una metaheurística de optimización sencilla y de aplicación general. Esta técnica surge como extensión de los métodos de Búsqueda Local [20], a los que se les adiciona un mecanismo de perturbación para escapar de los mínimos locales, como se puede apreciar en la Figura 3.2.

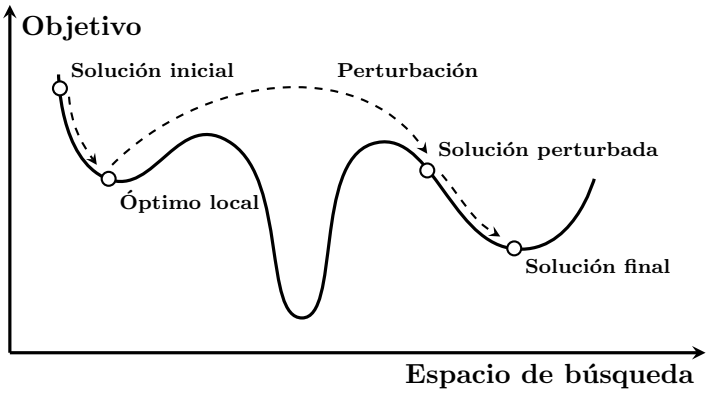

Figura 3.2: Comportamiento de la técnica Búsqueda Local Iterada en un problema de minimización.

En cada iteración, la solución actual es perturbada aleatoriamente y luego mejorada aplicando Búsqueda Local [20] en pos de mejorarla. El nuevo óptimo local obtenido por el método de mejora puede ser aceptado como nueva solución actual si pasa un test de aceptación. Este proceso se repite hasta que se cumpla un criterio de finalización determinado, generalmente asociado con un número máximo de iteraciones, el hallazgo de una solución con una calidad aceptable o la detección de estancamiento. Se presenta el pseudocódigo asociado a la técnica de búsqueda local iterada en el Algoritmo 5.

Algoritmo 5 Algoritmo de Búsqueda Local Iterada.

1: $s_0 \leftarrow$ generar una solución inicial
2: $s^* \leftarrow$ BusquedaLocal(s_0)
3: **repeat**
4: $\quad s' \leftarrow$ Perturbacion(s^*)
5: $\quad s'' \leftarrow$ BusquedaLocal(s')
6: $\quad s^* \leftarrow$ CriterioAceptacion(s^*,s'')
7: **until** se cumpla el criterio de parada
8: **return** s^*

El mecanismo de perturbación conforma uno de los aspectos claves a la hora de aplicar este tipo de técnicas. Las perturbaciones deben modificar la solución actual de forma que los cambios realizados no se reviertan rápidamente, esto colabora con el proceso de búsqueda posibilitando escapar de óptimos locales. Si la perturbación es muy débil, la búsqueda puede caer nuevamente en el óptimo local recientemente visitado, lo que implicaría un estancamiento. Sin embargo, con perturbaciones demasiado fuertes, se puede producir un comportamiento similar a un reinicio aleatorio del proceso de búsqueda, lo que disminuye las probabilidades de encontrar mejores soluciones en la siguiente fase de búsqueda. Para equilibrar estas cuestiones, el mecanismo de perturbación puede cambiar de forma adaptativa durante la búsqueda.

3.6. Resumen

En este capítulo se realizó un introducción a la optimización combinatoria, presentando formalmente los problemas de optimización. Luego, se presento una breve reseña sobre la teoría de la NP-completitud y particularmente sobre los problemas de clase NP-difíciles. Finalmente, se presentaron tres heurísticas para la resolución de problemas de optimización combinatoria, detallando su funcionamiento y sus principales características.

Capítulo 4

Caracterización de aplicaciones paralelas

Este capítulo presenta una clasificación de aplicaciones paralelas, realizada de acuerdo a los patrones de comunicación y sincronización que surgen de la cooperación entre los procesos que las integran. Esta clasificación será utilizada en el mecanismo de caracterización descrito en el Capítulo 6.

4.1. Clasificación de aplicaciones paralelas

En esta sección se definen las tres topologias identificadas, describiendo su forma más general, sus posibles variantes y algunos ejemplos. Las topologias identificadas se representan mediante grafos, donde los nodos simbolizan procesos, las aristas comunicaciones y los rectángulos sincronizaciones. Estas comunicaciones y sincronizaciones entre procesos ocurren a una determinada frecuencia, representada mediante etiquetas numéricas que indican la cantidad de ocurrencias por segundo.

4.1.1. Topología plana

La topología plana agrupa a las aplicaciones de tipo maestro/esclavo, en las cuales un proceso distinguido denominado maestro se encarga de controlar un conjunto de procesos llamados esclavos. En este modelo el maestro envía un conjunto de datos a los esclavos, cada esclavo procesa los datos recibidos y luego envía los resultados obtenidos al maestro. Estos resultados parciales son combinados por el proceso maestro con la finalidad de obtener el resultado final. Algunas variantes de este modelo pueden generarse debido a la utilización de diferentes patrones para la comunicación entre procesos

esclavos, así como también a la forma en que éstos se sincronizan entre sí, o a la existencia de un proceso esclavo prioritario el cual requiere un mayor grado de comunicación por parte del resto de los procesos. Un ejemplo de este tipo de aplicaciones es la implementada por el proyecto SETI@home [22] en el cual se analizan señales de radio buscando indicios de inteligencia extraterrestre. Seguidamente, se representa gráficamente el modelo de topología plana junto con alguna de sus variaciones las cuales surgen de las diferentes formas de interacción entre procesos. La Figura 4.1 representa una topología plana en su forma convencional, donde un proceso maestro intercambia datos con los procesos esclavos de forma equitativa, en este caso particular, a una frecuencia de cincuenta veces por segundo. En la Figura 4.2 se observa una topología plana en la que existe comunicación entre procesos esclavos, más específicamente, se realizan intercambios de información entre un proceso esclavo prioritario y el resto de los procesos esclavos. Finalmente, en la Figura 4.3 se puede apreciar una topología plana donde los procesos esclavos se sincronizan de a pares a una frecuencia de mil veces por segundo.

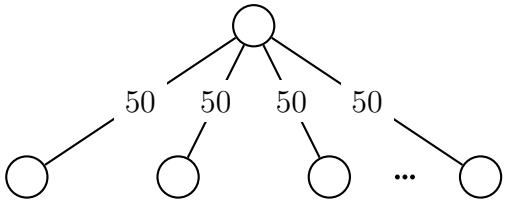

Figura 4.1: Topología plana en su forma convencional.

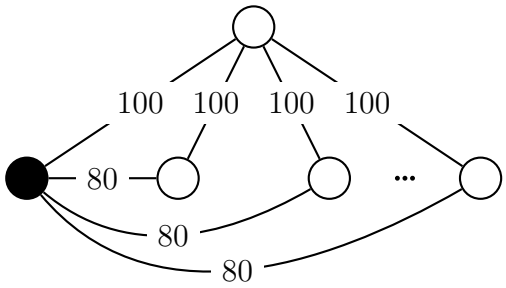

Figura 4.2: Topología plana con comunicación entre esclavos.

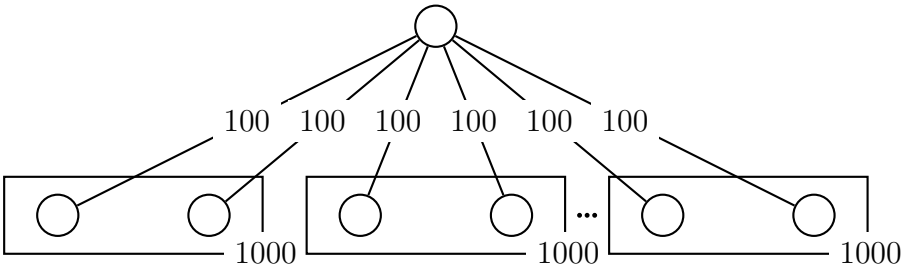

Figura 4.3: Topología plana con sincronización entre esclavos.

4.1.2. Topología dirigida por la aplicación

Esta topología engloba a las aplicaciones de tipo workflow, donde los datos son distribuidos y procesados de acuerdo a un flujo de trabajo específico. El particionamiento de datos puede ser realizado en forma fija o variable y siguiendo diferentes técnicas de descomposición de dominio, como por ejemplos filas y columnas para el caso de procesamiento de matrices e imágenes. Los procesos que componen estas aplicaciones generalmente operan sobre un conjunto de datos, pueden ejecutar operaciones de sincronización al final de una operación o de una iteración y pueden comunicarse con otros procesos, por ejemplo para enviar datos de frontera. Las relaciones de dependencia no siguen un patrón específico, un proceso puede depender de varios procesos o ninguno para comenzar a procesar. Este modelo se encuentra en problemas de simulación como *El juego de la vida* [23] o en el procesamiento de datos o imágenes, entre otros. Las Figuras 4.4 y 4.5 representan topologías dirigidas por la aplicación, con particionamiento por filas y por columnas, respectivamente. Además, ambas sincronizan de a tres procesos a diferencia de la topología representada en la Figura 4.6 que sincroniza por nivel.

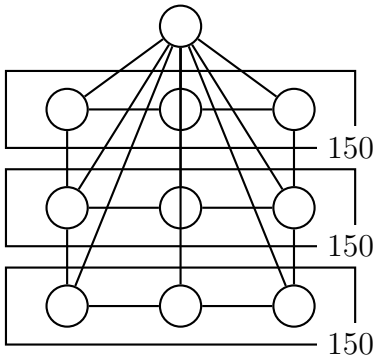

Figura 4.4: Topología dirigida por la aplicación con particionamiento por filas y sincronización de a tres procesos.

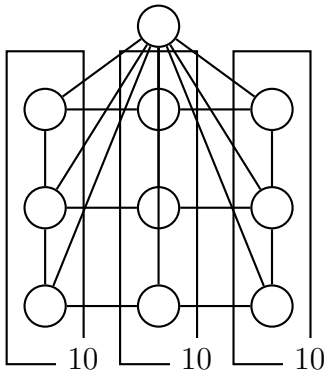

Figura 4.5: Topología dirigida por la aplicación con particionamiento por columnas y sincronización de a tres procesos.

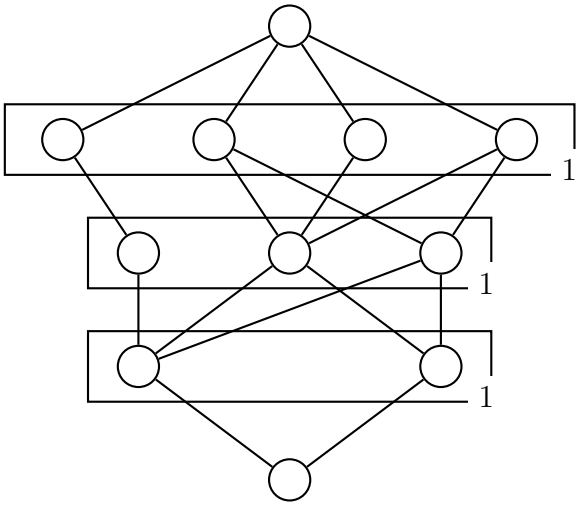

Figura 4.6: Topología dirigida por la aplicación con sincronización de procesos por nivel.

4.1.3. Topología jerárquica

En este tipo de aplicaciones existe un orden jerárquico entre los procesos. Cada proceso se comunica exclusivamente con un conjunto de procesos vecinos, los cuales se encuentran en niveles contiguos, mientras que las sincronizaciones pueden ocurrir por niveles o entre procesos hermanos. Estas aplicaciones presentan forma de árbol, donde los datos fluyen desde la raíz hacia las hojas y el resultado final es producido mediante la fusión de resultados parciales, generados en forma ascendente desde los procesos situados en

el último nivel. Un algoritmo paralelo que sigue este modelo es el algoritmo de Ordenamiento rápido (en inglés Quicksort) [24], el cual está basado en la estrategia *Divide and Conquer*. Básicamente, consiste en la elección de un pivot y el reposicionamiento del resto de los elementos, de modo de que a un lado queden los elementos menores al pivot y al otro lado los mayores. Esto genera dos sub-listas a las cuales se le aplicará el mismo mecanismo, de forma recursiva hasta ordenar todos los elementos. Seguidamente, se presentan dos ejemplos de aplicaciones paralelas pertenecientes a esta topología, con diferentes patrones de sincronización. La Figura 4.7 representa una topología jerárquica con sincronizaciones entre hermanos mientras que la Figura 4.8 ejemplifica una topología jerárquica con sincronización por nivel.

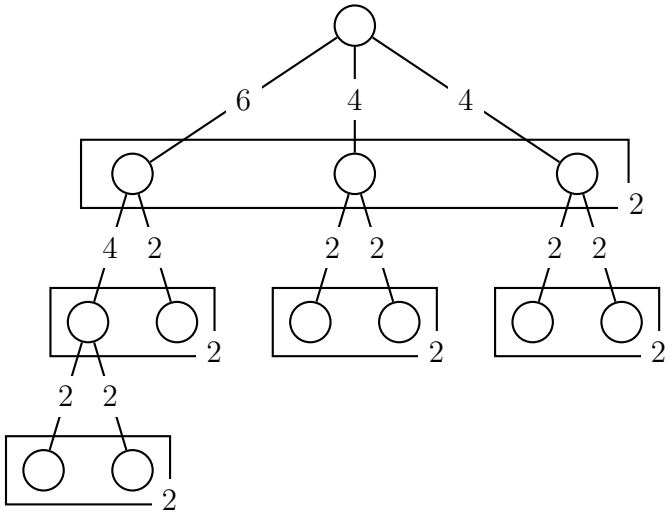

Figura 4.7: Topología jerárquica con sincronización entre hermanos.

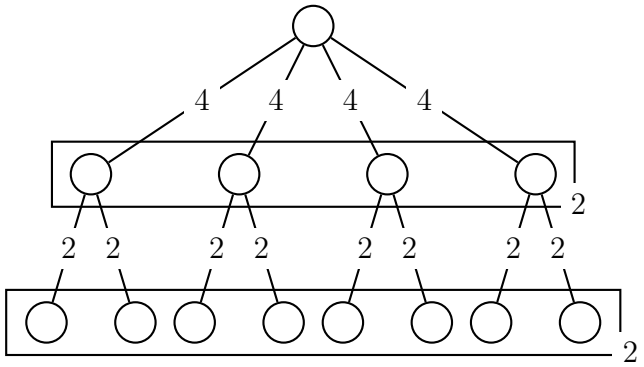

Figura 4.8: Topología jerárquica con sincronización por niveles.

4.2. Resumen

En este capítulo se planteó una clasificación de aplicaciones paralelas que identifica tres topologias: plana, dirigida por la aplicación y jerárquica. Se describió que tipo de aplicaciones integran cada una de las topologias y se representaron gráficamente algunas aplicaciones a modo de ejemplo. La clasificación realizada será de utilidad a la hora de elaborar mecanismos para la caracterización cualitativa y cuantitativa de aplicaciones paralelas, que permitirán determinar la cantidad de procesos y cuantificar los intercambios de información y sincronizaciones entre procesos.

Capítulo 5

Caracterización de sistemas multinúcleo

La heterogeneidad presente en los sistemas multinúcleo hace que los costos de comunicación y sincronización entre los diferentes elementos de cómputo sean variables [25]. Estas variaciones pueden ser causadas por la localidad de datos y las distintas velocidades de acceso a recursos, tales como memorias, caches, puertos de entrada/salida, etc. Por lo tanto, es de gran interés contar con herramientas para la caracterización de infraestructuras heterogéneas que permitan recopilar información valiosa para el diseño e implementación de algoritmos de planificación. En este capítulo se presenta la herramienta HwLoc, que permitirá caracterizar cualitativamente a los sistemas multinúcleo. Adicionalmente, se presenta una técnica de benchmark, mediante la cual será posible caracterizar cuantitativamente los costos de comunicación y sincronización entre las unidades de cómputo disponibles.

5.1. Hardware Locality

Hardware Locality (hwloc) [26] es un proyecto creado y mantenido por el Instituto Nacional de Investigación en Informática y en Automática (por sus siglas en francés, INRIA) situada en Bordeaux, Francia. Este proyecto surge en el año 2009 como remplazo y fusión del proyecto libtopology de INRIA y el proyecto Portable Linux Processor Affinity (por sus siglas en inglés, PLPA) de Open MPI. El software HwLoc recopila información cualitativa de un sistema de cómputo y la representa de forma portable y jerárquica mediante un árbol construido en base a la localidad de los recursos. En este árbol, cada nodo representa un recurso, como ser un socket, un procesador, un módulo de memoria, etc. Adicionalmente HwLoc es capaz de detectar dispositivos

PCI así como también interfaces de red, aceleradores Xeon Phi, etc. HwLoc provee una interfaz de línea de comandos y una interfaz de programación de aplicaciones (en inglés, Application Programming Interface o API) mediante la cual es posible consultar la topología, manipularla, exportarla en diferentes formatos, vincular un proceso a un recurso específico (indicar en qué recurso se va a ejecutar un proceso) pudiendo ser este recurso un nodo NUMA, un procesador, un core, etc. En cuanto a portabilidad vale destacar que HwLoc cuenta con soporte para varios de los sistemas operativos utilizados en la actualidad como ser los sistemas Linux, Solaris, AIX, OS X, FreeBSD, NetBSD, HP-UX, Microsoft Windows, entre otros. Tal como se mencionó anteriormente, HwLoc ofrece una API en C [26] mediante la cual es posible examinar la topología de un computador, exportarla, cargarla en otro computador, obtener información sobre los recursos, así como también vincular hilos o procesos a recursos. Seguidamente, se definen algunos conceptos importantes y se presentan algunos de los comandos provistos por la herramienta.

Topología en HwLoc

La topología de un computador es presentada jerárquicamente en forma de árbol, donde cada nodo representa un recurso del computador. Mediante la topología se describen las relaciones entre los recursos de un computador, así como también la disposición de cada uno de ellos dentro de la misma. Estos recursos pueden variar desde un computador hasta una unidad de procesamiento (en inglés, Processing Unit o PU), la cual se define como el recurso más pequeño capaz de ejecutar un proceso. HwLoc denomina a estos recursos como *objetos*. Cada uno de los objetos pertenecientes a un árbol de recursos es descrito mediante un conjunto de propiedades comunes, a continuación detallaremos algunas de ellas:

- Tipo: tipo de objeto, un objeto puede ser de tipo maquina, nodo NUMA, socket, core, etc.
- Nombre: nombre del objeto si es que existe.
- Índice físico: índice que el sistema operativo utiliza para identificar el objeto.
- Índice lógico: índice que identifica unívocamente objetos del mismo tipo y profundidad.
- Primo siguiente: puntero al siguiente objeto con el mismo tipo y con la misma profundidad.

- Primo anterior: puntero al objeto anterior con el mismo tipo y con la misma profundidad.
- Hermano siguiente: puntero al siguiente objeto que comparte el mismo padre.
- Hermano anterior: puntero al objeto anterior que comparte el mismo padre.
- Padre: puntero al objeto padre, en caso de que el objeto sea la raíz del árbol adopta el valor nulo.
- Hijos: Arreglo de punteros a objetos conformado únicamente por los hijos del objeto.
- Primer hijo: Puntero al primer objeto hijo del arreglo de objetos hijos.
- Último hijo: Puntero al último objeto hijo del arreglo de objetos hijos.
- Aridad: número de hijos del objeto.
- Profundidad: índice vertical en la jerarquía.
- CPU set: conjunto de procesadores lógicos incluidos directa o indirectamente en un objeto, esta propiedad es establecida para la vinculación de procesos o hilos a recursos.
- Rango entre hermanos: índice que identifica unívocamente objetos que son hijos de un mismo padre.
- Datos de usuario: puntero utilizado por las aplicaciones para guardar datos privados.

En la Figura 5.1 se puede observar una topología organizada jerárquicamente en cinco niveles. El primer nivel contiene la raíz del árbol y simboliza un computador en sí mismo, a partir de este nodo se organizan los recursos que conforman el computador. El segundo nivel contiene dos recursos de tipo socket, mientras que los caches se sitúan en el nivel tres, el cuarto nivel contiene los núcleos disponibles en el sistema y en el último nivel se encuentran las unidades de procesamiento.

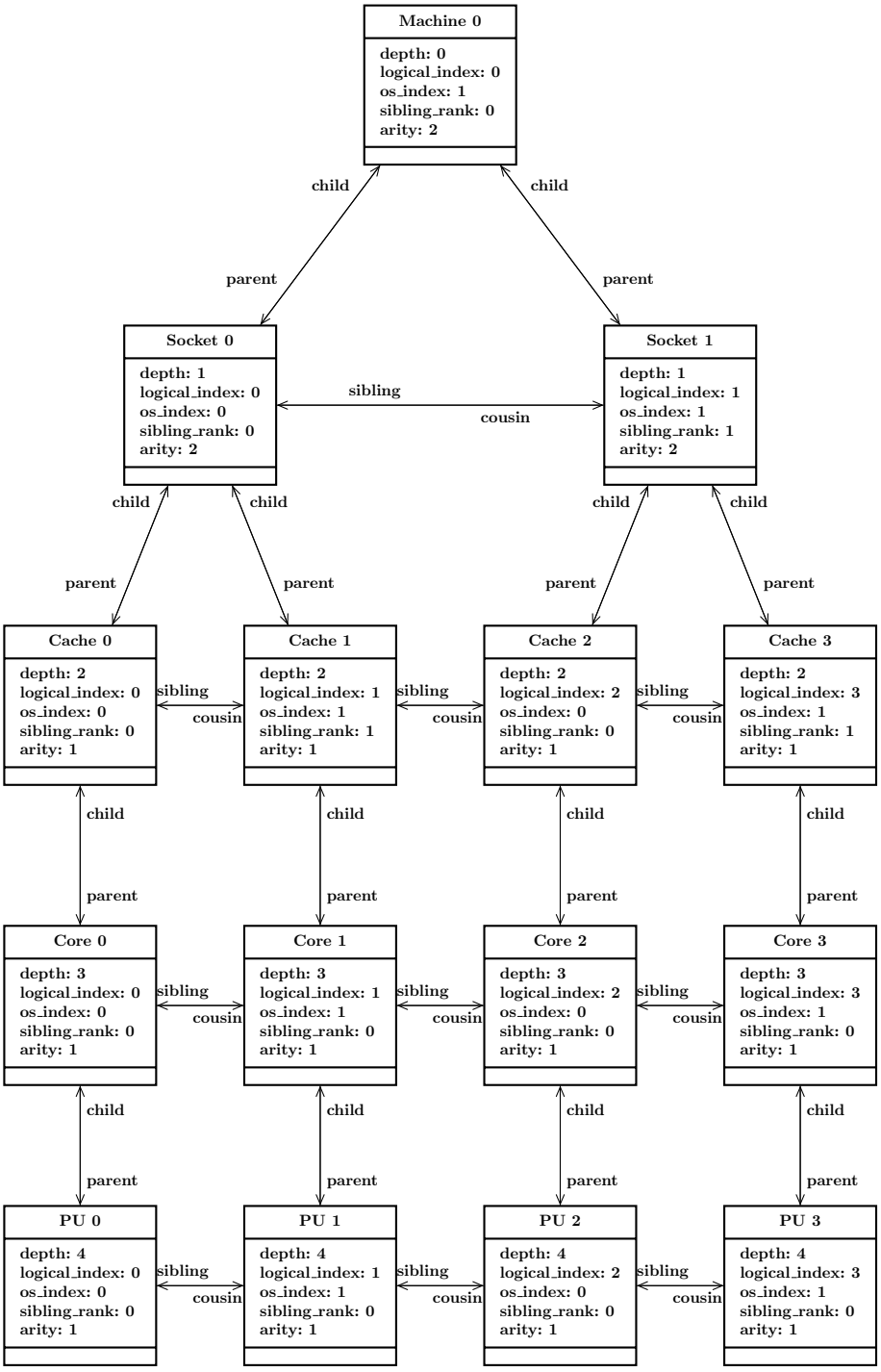

Figura 5.1: Representación interna en HwLoc de una topología de cinco niveles.

Interfaz de línea de comandos

HwLoc provee una interfaz de línea de comandos [26] donde se dispone de un conjunto de comandos que permiten por ejemplo obtener información sobre objetos, vincular hilos o procesos a recursos, exportar una topología en diferentes formatos como XML, PNG, PDF, entre otros. Seguidamente, analizaremos algunos de los comandos provistos por la herramienta.

hwloc-bind Vincula un proceso a uno o más objetos, por ejemplo a un core o socket específico. Este comando también puede ser utilizado para conocer con que recursos fue vinculado un proceso. El formato para especificar locaciones consiste en una lista de cpusets u objetos separadas por espacios.

hwloc-ps Este comando permite obtener una lista de todos los procesos o hilos que se encuentran vinculados con algún recurso del computador.

hwloc-info Retorna información sobre un objeto si es que se especifica alguno, en otro caso se retorna la información de la topología completa.

hwloc-assembler Combina varias topologías descritas en archivos XML en una nueva topología de multiples nodos, también descrita en un archivo XML.

hwloc-diff Retorna en un archivo XML las diferencias existentes entre dos topologías.

lstopo Despliega la topología, la salida puede ser en diferentes formatos como XML, PNG, PDF y otros. Es posible filtrar la topología de forma de ignorar algunos objetos, así como también es posible desplegar más información de los mismos. En la Figura 5.2 se puede apreciar una topología obtenida mediante el comando lstopo, la misma cuenta básicamente con 64 GB de memoria y 8 núcleos, distribuidos equitativamente entre dos nodos NUMA.

lstopo-no-graphics Ofrece exactamente las mismas posibilidades que ofrece el comando lstopo, descrito anteriormente, exceptuando que no soporta salidas gráficas.

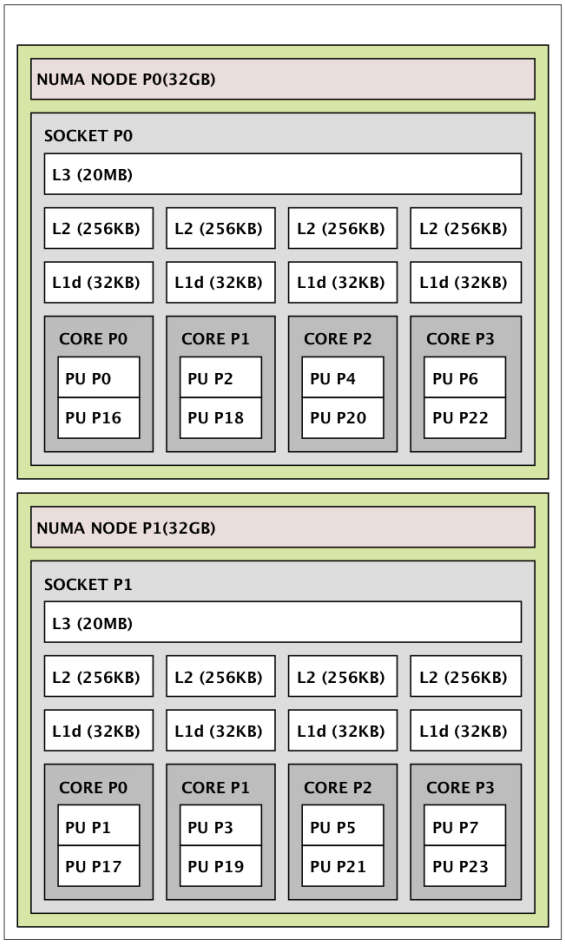

Figura 5.2: Topología obtenida con HwLoc mediante la utilización del comando lstopo.

5.2. Benchmarks

El benchmarking es un método utilizado para medir el rendimiento de un sistema o de componentes específicos mediante la ejecución de una aplicación o un conjunto de aplicaciones. En este caso particular, es de gran importancia determinar los costos de comunicación y sincronización entre las diferentes unidades de cómputo disponibles en un sistema. Para ello, se propone realizar esta caracterización cuantitativa de los costos mediante un conjunto de benchmarks de tipo ping-pong [27]. Los benchmarks utilizados consisten en aplicaciones paralelas que permiten establecer sincronizaciones y comunicaciones punto a punto entre dos unidades de procesamiento. Estas aplicaciones permitirán medir el costo temporal de realizar una operación de comunica-

ción o sincronización entre dos procesos ejecutando en diferentes pares de núcleos de un sistema. Los benchmarks fueron implementados en lenguaje `C` y utilizan la implementación MPICH de la Interfaz de Pasaje de Mensajes (en inglés, Message Passing Interface o MPI) [28]. Las comunicaciones fueron implementadas mediante los métodos send/receive y las sincronizaciones a través del método barrier de MPI.

5.3. Resumen

En este capítulo se presentaron dos herramientas para la caracterización de sistemas de cómputo multinúcleo. Se realizó una descripción de la herramienta HwLoc, detallando sus principales características y funcionalidades. Se propuso un benchmark basado en MPI para la caracterización de los costos de comunicación y sincronización de una arquitectura multinúcleo. Estas dos herramientas permitirán definir un mecanismo para la caracterización cualitativa y cuantitativa de sistemas multinúcleo. De esta forma, se podrán determinar la cantidad de núcleos disponibles y los costos temporales que implican los intercambios de información y las sincronizaciones entre procesos.

Capítulo 6

Heurísticas de optimización para planificación con afinidades

En este capítulo se definen los elementos necesarios para el diseño e implementación de los planificadores, se describen mecanismos para la caracterización de aplicaciones paralelas y de sistemas multinúcleo, la representación de instancias del problema de planificación y el modo en que serán representadas las soluciones (planificaciones). Seguidamente, se proponen tres algoritmos heurísticos para resolver el problema de planificación de procesos por afinidad. Los mismos fueron implementados de acuerdo a las siguientes heurísticas de optimización combinatoria: Heurística Ávida (en inglés, Greedy Heuristic), Búsqueda en Escalada (en inglés, Hill Climbing) y Búsqueda Local Iterada (en inglés, Iterated Local Search).

6.1. Mecanismos de caracterización

La clasificación definida en el Capítulo 4 permite agrupar a las aplicaciones paralelas en tres topologias o modelos. Estos modelos serán presentados al usuario, que indicará de forma interactiva con cuál de ellos se corresponde su aplicación paralela, así como también deberá especificar el número de operaciones de comunicación y sincronización efectuadas entre los diferentes pares de procesos que componen la aplicación paralela. La información provista a través del mecanismo de caracterización interactivo será utilizada para determinar la cantidad de procesos b a planificar y las funciones de comunicación C y sincronización S definidas en la formulación del problema realizada en el Capítulo 2. Estos elementos podrán ser utilizados por los

algoritmos heurísticos, permitiéndoles computar de forma precisa la planificación de aplicaciones paralelas. Por otra parte, la herramienta de software HwLoc y los benchmarks de tipo ping-pong presentados previamente en el Capítulo 5 serán empleados para realizar la caracterización cualitativa de los componentes de un sistema multinúcleo y la caracterización cuantitativa de los costos temporales de comunicación y sincronización entre las unidades de procesamiento. Utilizando la información obtenida a través de la herramienta HwLoc y luego calculando los valores cuantitativos a través de las aplicaciones de benchmark se podrán determinar la cantidad de núcleos a con los que cuenta un sistema multinúcleo, la función de costos de comunicación CC y la función de costos de sincronización SC, descritas en el Capítulo 2. Estos datos cuantitativos podrán ser empleados por los algoritmos de planificación, logrando de este modo determinar planificaciones que consideren las afinidades existentes entre los diferentes componentes de un sistema de cómputo multinúcleo.

6.2. Representación de instancias del problema

Las instancias del problema de planificación de procesos con afinidades serán representadas mediante cuatro matrices: dos matrices con la descripción cuantitativa de las operaciones de comunicación y sincronización (MC y MS) y otras dos matrices con la descripción cuantitativa de los costos de comunicación y sincronización (MCC y MCS). Las matrices MC y MS, ambas de tamaño $b \times b$ (siendo b la cantidad de tareas) describen los valores asociados a las evaluaciones de las funciones $C(t_i, t_j)$ y $S(t_i, t_j)$ $\forall t_i, t_j \in T$, acumulando el número de operaciones de comunicación y sincronización realizadas por cada uno de los procesos, respectivamente. Por otro parte, la matriz MCC de tamaño $a \times a$ (siendo a la cantidad de núcleos), contiene los valores correspondientes a las evaluaciones de la función $CC(n_i, n_j)$ y describe el tiempo necesario para realizar una operación de comunicación $\forall n_i, n_j \in N$. Análogamente, la matriz MSC de tamaño $a \times a$ (siendo a la cantidad de núcleos) almacena los valores asociados a las evaluaciones de la función $CS(n_i, n_j)$ y describe el tiempo necesario para realizar una operación de sincronización $\forall n_i, n_j \in N$.

6.3. Representación de soluciones

Las soluciones al problema de planificación de procesos se representarán a través de vectores de largo a, siendo a el número de núcleos disponibles en el computador. Cada posición del vector se corresponde con una unidad de procesamiento y cada elemento del vector representa un proceso asignado a la correspondiente unidad de cómputo. La Figura 6.1 presenta un ejemplo de planificación de una aplicación paralela compuesta por b procesos sobre un computador de a núcleos.

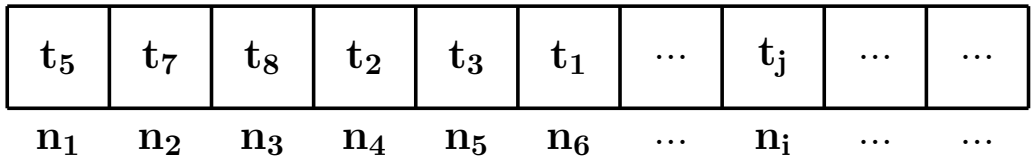

Figura 6.1: Ejemplo de planificación de una conjunto de procesos sobre un conjunto de núcleos.

La solución a un problema de planificación implicará un costo medido en unidades de tiempo, asociado al tiempo necesario para efectuar cada una de las operaciones de sincronización y comunicación entre los procesos planificados. Utilizando las matrices MC, MS, MCC y MCS definidas en la sección anterior es posible calcular el costo temporal asociado a una planificación s, tal como se observa en la Ecuación 6.1. Donde el costo temporal total queda determinando por la suma de los costos relativos a las operaciones de sincronización y comunicación, determinados por las Ecuaciones 6.2 y 6.3, respectivamente.

$$costo(s) = sync(s) + comm(s) \tag{6.1}$$

$$sync(s) = \sum_{n_i \in N} \sum_{n_j \in N} MS(s(n_i), s(n_j)) \times MCS(n_i, n_j) \tag{6.2}$$

$$comm(s) = \sum_{n_i \in N} \sum_{n_j \in N} MC(s(n_i), s(n_j)) \times MCC(n_i, n_j) \tag{6.3}$$

Mediante esta ecuación es posible evaluar la calidad de las soluciones y realizar comparaciones que permitan determinar si una solución es mejor que otra en términos de los costos temporales causados por las operaciones de

comunicación y sincronización. La estructura vectorial presentada anteriormente resulta una manera sencilla de representar una planificación y facilita la implementación de los algoritmos heurísticos que se presentan a continuación.

6.4. Heurística Ávida para la planificación por afinidad

La heurística Ávida consiste en generar una solución parcial inicial de forma aleatoria para posteriormente expandir la misma gradualmente aplicando una estrategia que considere afinidades de software y hardware. El proceso de expansión finaliza cuando todos los procesos se encuentren asignados a algún recurso de procesamiento. Cada una de las asignaciones serán realizadas localmente, considerando la planificación computada hasta ese momento. La estrategia de expansión consiste en seleccionar el proceso que maximice el número de interacciones (comunicaciones y sincronizaciones) y el núcleo que minimice el costo temporal de realizar esas operaciones de interacción, para posteriormente, incorporarlas a la planificación actual. Las afinidades de software serán evaluadas haciendo uso de las matrices de interacción (MC y MS), mientras que las afinidades de hardware serán calculadas utilizando las matrices de costo de interacción (MCC y MCS). El Algoritmo 6 presenta el pseudocódigo para la planificación mediante la heurística Ávida.

Algoritmo 6 Planificador Ávido para la planificación por afinidad.

```
1: solución ← inicializar vector solución
2: proceso ← seleccionar proceso de forma aleatoria
3: núcleo ← seleccionar núcleo de forma aleatoria
4: solución[núcleo] ← proceso
5: while (existan procesos sin asignar) do
6:     proceso ← seleccionar proceso que maximice el nro. de interacciones
7:     núcleo ← seleccionar núcleo que minimice el costo de interacción
8:     solución[núcleo] ← proceso
9: end while
10: return solución
```

6.5. Búsqueda en Escalada para la planificación por afinidad

La heurística de Búsqueda en Escalada parte de una solución inicial generada mediante algún mecanismo de inicialización. Luego se analizan las soluciones vecinas en busca de mejores planificaciones. En caso de existir alguna, ésta reemplazará a la solución actual, en caso contrario, el algoritmo finaliza y retorna la mejor solución obtenida hasta el momento. Es necesario definir previamente el método utilizado para generar la solución inicial, el operador de movimiento aplicado para producir los vecindarios y el tipo de exploración utilizada para seleccionar la mejor solución en un vecindario. La solución inicial será obtenida mediante el planificador Ávido y el operador de movimiento consistirá en intercambiar un par de asignaciones entre sí. Dada un solución s y dos procesos t_i y t_j asignados a los núcleos n_k y n_l, respectivamente, la solución vecina será obtenida aplicando las asignaciones $s[n_k] = t_j$ y $s[n_l] = t_i$. El operador de movimiento seleccionado produce vecindarios de tamaño $(b^2 - b)/2$, donde b es el número de procesos a planificar. El mejor vecino será determinando bajo la realización de una exploración exhaustiva (Escalada por Máxima Pendiente), lo que implica evaluar las $(b^2 - b)/2$ soluciones en pos de determinar la mejor solución en términos de los costos temporales definidos por la Ecuación 6.1. En el Algoritmo 7 se puede apreciar el pseudocódigo asociado a la heurística de Búsqueda en Escalada.

Algoritmo 7 Planificador Búsqueda en Escalada para la planificación por afinidad.

```
1: solución ← generar solución inicial con el planificador Ávido
2: escalar ← true
3: iteraciones ← fijar el número máximo de iteraciones
4: i ← 0
5: repeat
6:     vecindario ← generar vecindario de la solución
7:     if (existen vecinos mejores que solución) then
8:         solución ← seleccionar el mejor de los vecinos
9:     else
10:        escalar ← false
11:    end if
12:    i = i + 1
13: until ((not escalar) || (i >= iteraciones))
14: return solución
```

6.6. Búsqueda Local Iterada para la planificación por afinidad

La heurística de Búsqueda Local Iterada comienza con la generación de una solución candidata que posteriormente es mejorada aplicando la técnica de Búsqueda Local [20]. El proceso iterativo de mejora implica la perturbación de la solución actual y la posterior mejora a través de la aplicación de Búsqueda Local. La nueva solución obtenida es comparada con la solución actual. En caso de que la nueva solución implique un menor costo en términos de tiempo, ésta será aceptada como nueva solución actual, en caso contrario, el proceso continua hasta alcanzar el número máximo de iteraciones. La solución inicial será generada a través del planificador Ávido y el proceso de mejora de las soluciones será realizado mediante la aplicación del algoritmo de Búsqueda en Escalada presentado en la sección anterior. El operador de movimiento utilizado se mantendrá igual al descrito para la Búsqueda en Escalada y el mecanismo de perturbación consistirá en intercambiar una cantidad predeterminada de asignaciones, aplicando el operador de movimiento en forma repetitiva. Seguidamente, se presenta el Algoritmo 8 con el pseudocódigo asociado a la heurística de Búsqueda Local Iterada.

Algoritmo 8 Planificador Búsqueda Local Iterada para la planificación por afinidad.

```
1: solución ← generar solución inicial con el planificador Ávido
2: solución ← mejorar solución con Búsqueda en Escalada
3: iteraciones ← fijar el número máximo de iteraciones
4: i ← 0
5: repeat
6:     perturbación ← perturbar solución
7:     perturbación ← mejorar perturbación con Búsqueda en Escalada
8:     if (perturbación es mejor que solución) then
9:         solución ← perturbación
10:    end if
11:    i = i + 1
12: until (i >= iteraciones)
13: return solución
```

6.7. Resumen

En este capítulo se han presentando un conjunto de elementos necesarios para el diseño e implementación de los algoritmos heurísticos de planificación por afinidad. Se describieron mecanismos para la caracterización de aplicaciones paralelas y sistemas multinúcleo, se definieron las formas en las que serán representadas las instancias del problema y las soluciones, conjuntamente con una ecuación que permitirá medir la calidad de las mismas de acuerdo a la instancia de problema que resuelvan. Finalmente se presentaron los algoritmos de planificación, cubriendo los aspectos necesarios para su implementación.

Capítulo 7

Análisis experimental

Este capítulo presenta los principales resultados obtenidos en los experimentos de evaluación de los algoritmos heurísticos descritos en el Capítulo 6, analizando específicamente la calidad de las soluciones y la eficiencia computacional. Inicialmente, se describe la plataforma de ejecución utilizada para realizar el análisis experimental y el conjunto de instancias de evaluación consideradas en la realización del mismo. Seguidamente, se presenta una heurística para la planificación de procesos en sistemas multinúcleo que no considera afinidades, la cual será utilizada como línea base para la comparación de resultados respecto a las tres heurísticas presentadas anteriormente. Finalmente, se reportan los resultados obtenidos del análisis experimental realizado sobre las heurísticas de optimización. El reporte incluye resultados numéricos, las mejoras porcentuales alcanzadas por las heurísticas de optimización respecto a la heurística de línea base y los márgenes de mejora que presentan los planificadores por afinidad implementados respecto a cotas inferiores.

7.1. Plataforma de ejecución

Las pruebas fueron realizadas sobre un computador multinúcleo ProLiant DL585 G7, con 64 núcleos distribuidos en 4 procesadores AMD Opteron(TM) Processor 6272 a 2.10 GHz, 48 GB de memoria RAM y sistema operativo CentOS Linux, perteneciente a la plataforma de alto desempeño Cluster FING de la Facultad de Ingeniería, Universidad de la República, Uruguay [29].

7.2. Instancias de evaluación

Para la evaluación experimental, los algoritmos de planificación fueron aplicados sobre un conjunto de instancias del problema basadas en tres arquitecturas reales pertenecientes al Cluster FING [29] y en las tres topologias de aplicaciones paralelas: plana, dirigida por la aplicación y jerárquica. Un total de 240 instancias fueron generadas para evaluar los algoritmos de planificación, cada una de ellas describe una aplicación paralela y una infraestructura de cómputo sobre la cual será ejecutada. A continuación, se describen las arquitecturas de cómputo, los tipos de aplicaciones paralelas seleccionadas y el conjunto de instancias de evaluación generado.

7.2.1. Arquitecturas de cómputo

Se seleccionaron tres arquitecturas de cómputo multinúcleo, $A = \{a_1, a_2, a_3\}$ que actualmente forman parte de la plataforma de alto desempeño Cluster FING perteneciente a la Facultad de Ingeniería, Universidad de la República, Uruguay [29]. Las arquitecturas fueron caracterizadas cualitativa a través HwLoc y cuantitativa mediante la utilización de benchmarks de tipo ping-pong. A continuación, se presentan las tres arquitecturas, describiendo la topología de sus componentes y reportando en la Tabla 7.1 el tiempo necesario, en microsegundos, para comunicar o sincronizar dos procesos, caracterizado de acuerdo a los recursos compartidos entre los núcleos involucrados en la interacción.

Arquitectura a_1 La primera arquitectura seleccionada corresponde a un computador multinúcleo ProLiant DL385 G7. En la Figura 7.1 se puede apreciar que el mismo está compuesto por 2 sockets con un par de nodos NUMA cada uno, cada nodo presenta 3 niveles de memoria caché y contiene 6 núcleos, totalizando 24 núcleos y 28 GB de memoria.

Arquitectura a_2 La segunda arquitectura elegida consiste en un computador multinúcleo ProLiant DL385p Gen8. La Figura 7.2 presenta la topología del computador, en ella se observan 32 núcleos y 32 GB de memoria organizados en 2 sockets con 2 nodos NUMA cada uno. Cada nodo cuenta con 8 núcleos y memoria caché de tres niveles.

Arquitectura a_3 La tercera arquitectura seleccionada corresponde a un computador ProLiant DL585 G7. En la Figura 7.3 se pude observar que el computador posee 4 sockets con 2 nodos NUMA, cada nodo contiene 8 núcleos y caches de 3 niveles, totalizando 64 núcleos y 64 GB de memoria.

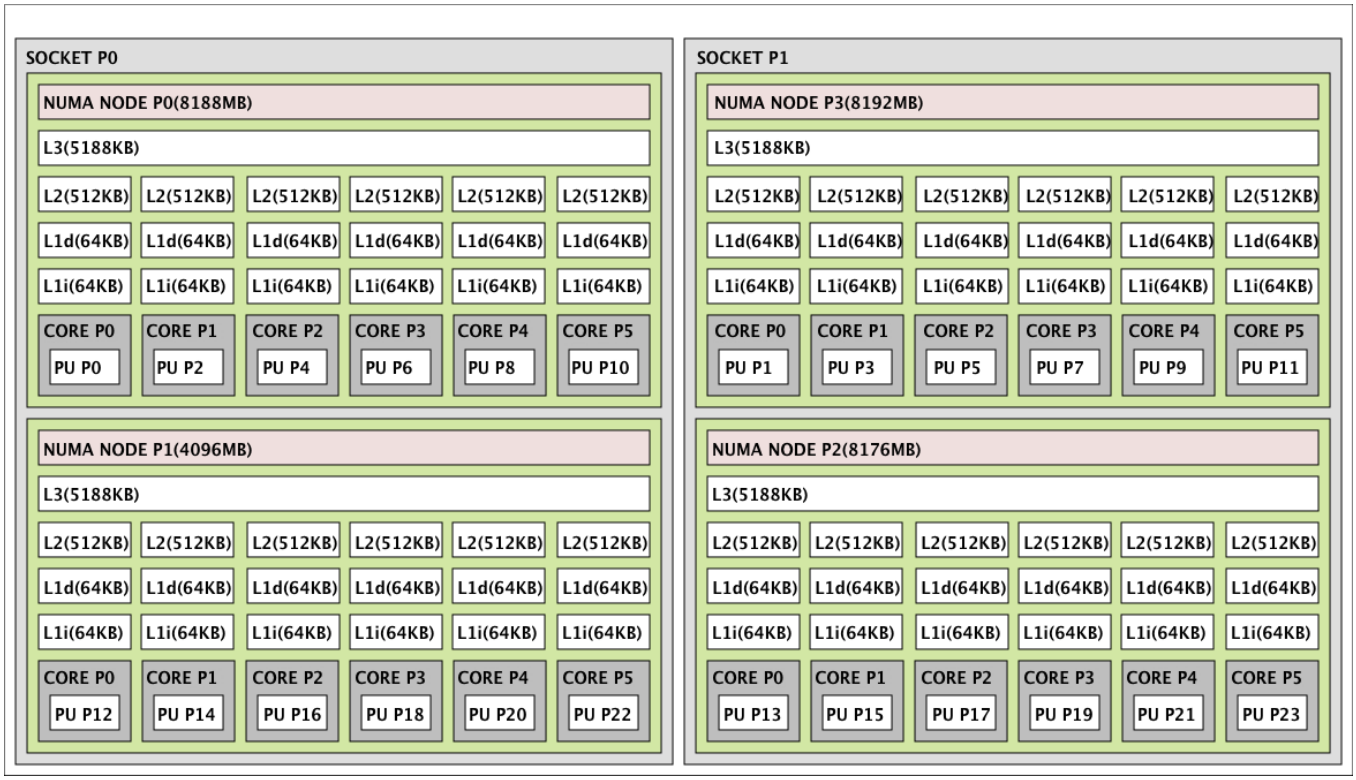

Figura 7.1: Topología generada con HwLoc de la arquitectura a_1.

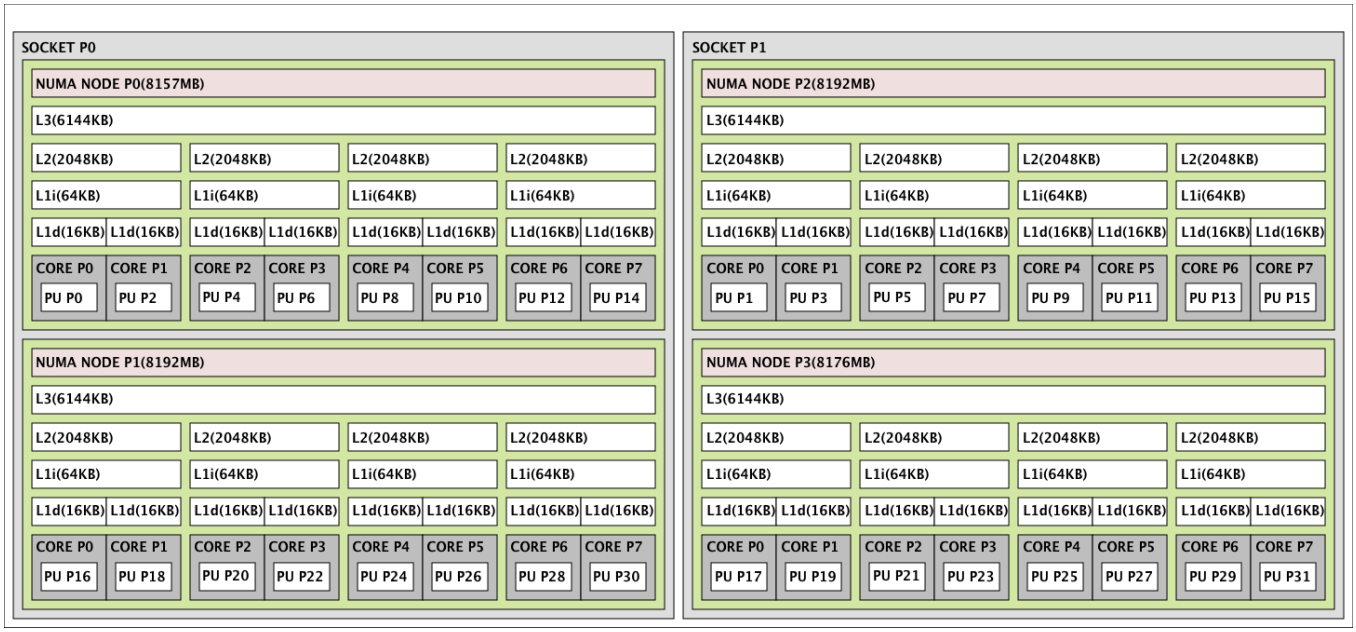

Figura 7.2: Topología generada con HwLoc de la arquitectura a_2.

Figura 7.3: Topología generada con HwLoc de la arquitectura a_3.

Tabla 7.1: Costos de interacción (comunicaciones y sincronizaciones) entre dos procesos para las arquitecturas consideradas en el análisis experimental.

Núcleos		**Interacción (μs)**	
Socket	**Caché**	**Comunicación**	**Sincronización**
	Arquitectura a_1		
comparten	comparten L3	1.44	1.89
comparten	no comparten	1.84	2.11
no comparten	no comparten	1.85	2.12
	Arquitectura a_2		
comparten	comparten L2 y L3	1.18	1.90
comparten	comparten L3	1.36	2.19
comparten	no comparten	1.50	2.19
no comparten	no comparten	1.56	2.25
	Arquitectura a_3		
comparten	comparten L2 y L3	1.38	1.98
comparten	comparten L3	1.33	1.91
comparten	no comparten	1.58	2.07
no comparten	no comparten	1.76	2.09

7.2.2. Aplicaciones paralelas

Se diseñaron e implementaron tres aplicaciones paralelas en representación de las tres topologias de aplicaciones identificadas en el Capítulo 4. El lenguaje elegido para la codificación fue `C++` y las interacciones entre procesos fueron realizadas a través del estándar MPI [28], utilizando la implementación MPICH en su version 3.0.4. Las operaciones de comunicación fueron realizadas utilizando las funciones send/receive, mientras que las sincronizaciones fueron implementadas con la función barrier. Las aplicaciones admiten la configuración de algunos aspectos a través de argumentos, como ser, la cantidad de procesos a emplear, la forma en que los datos serán particionados, el patrón de sincronización de procesos, entre otros.

Transferencia de calor

La primera aplicación pertenece a la familia de topologias planas, la misma describe la evolución de la temperatura en un barra de largo l durante un periodo de tiempo t, resolviendo paralelamente la ecuación de calor bajo el paradigma maestro/esclavo [30]. En el Algoritmo 9 se puede apreciar el pseudocódigo de la aplicación, donde un proceso maestro es el encargado de particionar la barra en secciones y asignarlas a un conjunto de procesos esclavos. Los procesos esclavos calcularán las nuevas temperaturas de la sección asignada resolviendo la ecuación de calor mediante el método de diferencias finitas [30], para posteriormente enviarlas al proceso maestro.

Algoritmo 9 Aplicación de transferencia de calor.

```
1: parallel
2:     pid ← identificador de proceso
3:     tamaño ← cantidad de procesos
4:     sincronización ← patrón de sincronización entre esclavos
5:     tiempo ← periodo de evaluación
6:     datos ← temperatura en cada punto de la barra
7:     if pid == maestro then
8:         particiones ← particionar barra en (tamaño - 1) secciones
9:         for i ≤ tiempo do
10:            for each s in 1..(tamaño - 1) do
11:                partición ← particiones[s]
12:                send(s, datos[partición])
13:            end for
14:            for each s in 1..(tamaño - 1) do
15:                partición ← particiones[s]
16:                datos[partición] ← recv(s)
17:            end for
18:            i ← (i + 1)
19:        end for
20:    else
21:        for i ≤ tiempo do
22:            datos ← recv(maestro)
23:            datos ← calcular(datos)
24:            barrier(sincronización)
25:            send(s, datos)
26:        end for
27:    end if
28: end parallel
```

Flujo de trabajo

La siguiente aplicación presenta una topología dirigida por la aplicación, la misma consiste en un flujo de trabajo genérico construido aleatoriamente. El primer proceso generará un grafo de dependencia de forma aleatoria, este grafo será utilizado por el resto de los procesos para determinar los procesos con los que se realizaran los intercambios de información y sincronizaciones, tal como se observa en el pseudocódigo de la aplicación detallado en el Algoritmo 10. Actualmente, técnicas como el Registros de Imágenes (en inglés, Image Registration o IR) [31] y el procesamiento de datos bajados de un sitio web [32], son modeladas como aplicaciones de flujos de trabajo.

Algoritmo 10 Aplicación de flujo de trabajo.

```
1: parallel
2:     pid ← identificador de proceso
3:     particionamiento ← tipo de particionamiento
4:     datos ← conjunto de datos
5:     if pid == 0 then
6:         grafo ← generar grafo de dependencia aleatoriamente
7:         particiones ← particionar(particionamiento)
8:         for each p in siguienteNivel(grafo) do
9:             partición ← particiones[p]
10:            send(p, datos[partición])
11:        end for
12:    else
13:        if pid < (cantidad de procesos - 1) then
14:            for each p in anteriorNivel(grafo) do
15:                datos ← recv(p)
16:            end for
17:            barrier(grafo)
18:            particiones ← particionar(particionamiento)
19:            for each p in siguienteNivel(grafo) do
20:                partición ← particiones[p]
21:                send(p, datos[partición])
22:            end for
23:        else
24:            for each p in anteriorNivel(grafo) do
25:                datos ← recv(p)
26:            end for
27:        end if
28:    end if
29: end parallel
```

Ordenamiento rápido

La última aplicación desarrollada corresponde a la familia de aplicaciones jerárquicas, la misma ordena un conjunto de números enteros aplicando una estrategia de ordenamiento rápido [24]. En el Algoritmo 11 se puede apreciar el pseudocódigo de la aplicación, donde cada proceso recibe una lista de enteros que será dividida en dos sub-listas de acuerdo a un pivote, una contendrá los enteros mayores al pivote seleccionado y la otra los enteros menores. Las listas serán enviadas a los procesos hijos, en caso de no existir procesos disponibles, el ordenamiento se realizará de forma serial aplicando la misma estrategia.

Algoritmo 11 Aplicación de ordenamiento rápido.

```
1: parallel
2:     pid ← identificador de proceso
3:     sincronización ← patrón de sincronización entre procesos
4:     lista ← lista de números enteros
5:     if pid == 0 then
6:         ordenamientoParalelo(lista, pid)
7:     else
8:         lista ← recv(padre(pid))
9:         ordenamientoParalelo(lista, pid)
10:        barrier(sincronización)
11:        send(padre(pid), lista)
12:    end if
13: end parallel
14: procedure ordenamientoParalelo(lista, pid)
15:    pivote ← seleccionar pivote entere los enteros de la lista
16:    izquierda, derecha ← inicializar listas
17:    for each e in lista do
18:        if e > pivote then
19:            derecha ← agregar(e)
20:        else
21:            izquierda ← agregar(e)
22:        end if
23:    end for
24:    send(hijoizquierdo(pid), izquierda)
25:    send(hijoderechao(pid), derecha)
26:    izquierda ← recv(hijoizquierdo(pid))
27:    derecha ← recv(hijoderechao(pid))
28:    lista ← concatenar(izquierda, derecha)
29: end procedure
```

7.2.3. Conjunto de instancias de evaluación

Esta sección describe como fue generado el conjunto de instancias de evaluación y resume el conjunto propiamente dicho. Las instancias de evaluación fueron creadas bajo la premisa de obtener un conjunto de instancias representativo y heterogéneo. Para ello, la cantidad de procesos en una aplicación dependerá del número de núcleos, pudiendo utilizar el 25 %, 50 % y 100 % de los núcleos disponibles en el sistema. El tamaño de la entrada de una aplicación paralela será clasificado de acuerdo a la cantidad total de interacciones necesarias para resolver el problema, si el número de interacciones se encuentra en el rango $[0, 10^8)$ ésta será catalogada como pequeña (P), si se encuentra en el rango $[10^8, 10^9)$ ésta será catalogada como mediana (M) y si se encuentra en el rango $[10^9, +\infty)$ ésta será catalogada como grande (G). A continuación, se presentan algunos aspectos específicos sobre las aplicaciones desarrolladas para la generación de instancias.

Transferencia de calor La aplicación que evalúa la transferencia de calor (de ahora en más TC) permite configurar el patrón de sincronización entre los procesos esclavos. Los mismos podrán sincronizarse de a 2, 3, 4, 5 y hasta $b - 1$ procesos, siendo b la cantidad de procesos. Las Figuras 7.4 y 7.5 presentan instancias TC de 6 procesos donde los procesos esclavos se sincronizan de a pares y de 8 procesos (todos los esclavos se sincronizan entre sí), respectivamente.

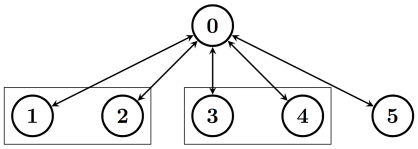

Figura 7.4: Instancia de 6 procesos y sincronizaciones de a 2 esclavos.

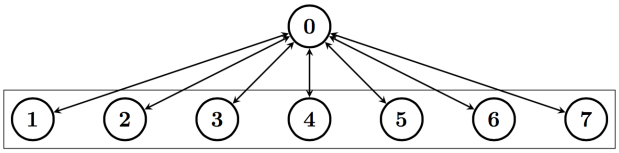

Figura 7.5: Instancia de 8 procesos y sincronizaciones entre todos los procesos esclavos.

Flujo de trabajo La aplicación que representa un flujo de trabajo genérico (de ahora en más FT) admite dos tipos de particionamiento de datos: uno donde los datos son particionados de forma equitativa (fija) y otro en el que las particiones son creadas aleatoriamente (dinámica). Además, cada proceso se sincronizará con los procesos en su mismo nivel antes de enviar las particiones, tal como se observa en las Figuras 7.6 y 7.7, que representan instancias FT de 8 y 12 procesos, respectivamente.

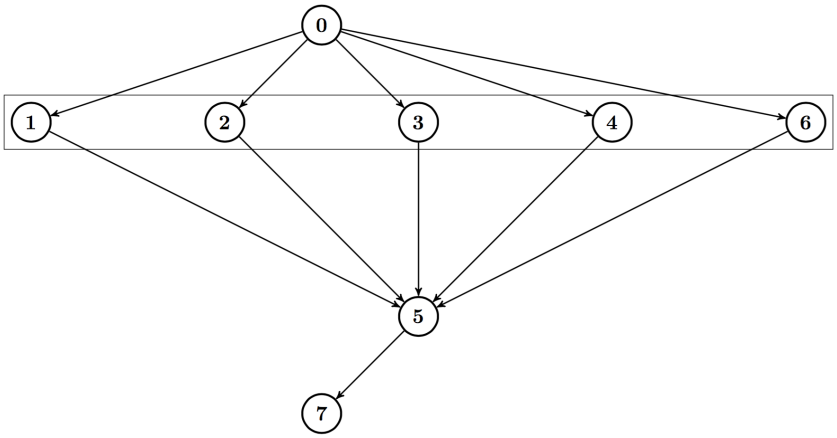

Figura 7.6: Instancia de 8 procesos y sincronizaciones por nivel.

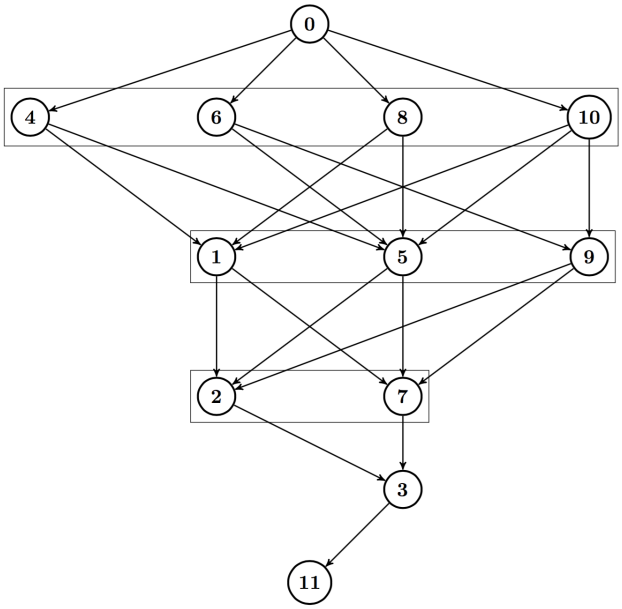

Figura 7.7: Instancia de 12 procesos y sincronizaciones por nivel.

Ordenamiento rápido En la aplicación de ordenamiento rápido (de ahora en más OR) los procesos serán sincronizados antes de realizar envíos de información a sus procesos hijos, estas sincronizaciones podrán ocurrir por niveles o entre procesos hermanos. Las Figuras 7.8 y 7.9 presentan instancias de 12 y 24 procesos, respectivamente, y reflejan los dos patrones de sincronización descritos anteriormente.

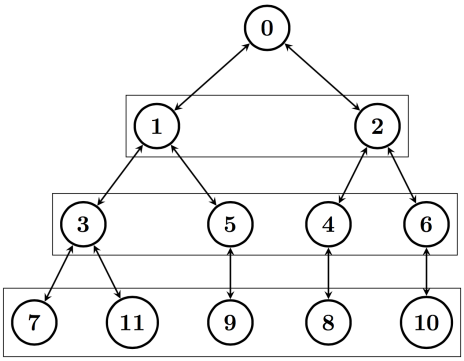

Figura 7.8: Instancia de 12 procesos y sincronizaciones entre procesos de un mismo nivel.

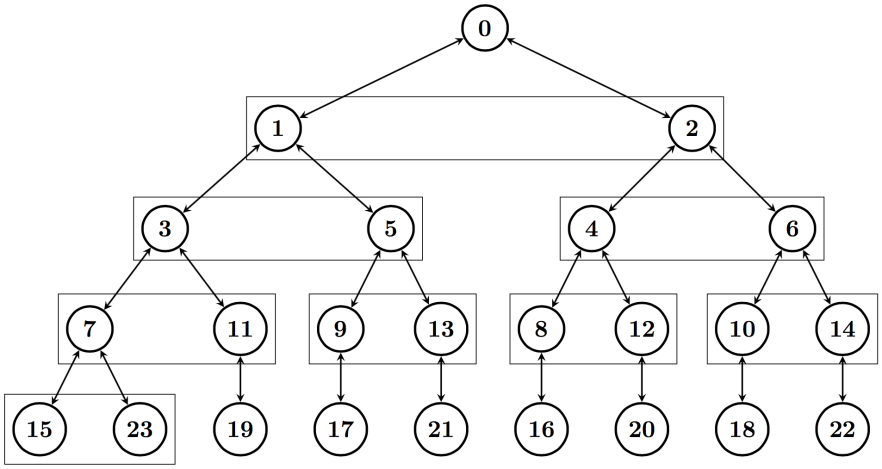

Figura 7.9: Instancia de 24 procesos y sincronizaciones entre procesos hermanos.

La Tabla 7.2 resume las 240 instancias construidas para evaluar los algoritmos de planificación por afinidad, organizadas por arquitectura. En ella, se indica la cantidad de procesos (coincidente con el 25 %, 50 % o 100 % de los núcleos disponibles), el tamaño de la entrada (pequeña, mediana o grande),

la forma de particionamiento aplicada en las aplicaciones FT (fija o dinámica) y los patrones de sincronización utilizados en las aplicaciones TC (2,3,4,5 o $b-1$ procesos esclavos, siendo b la cantidad de procesos) y OR (entre procesos hermanos o por nivel).

Tabla 7.2: Conjunto de instancias de evaluación.

Procesos (cantidad)	**Entrada** (tamaño)	**TC** (sincronización)	**FT** (partición)	**OR** (sincronización)
Arquitectura a_1 (24 núcleos disponibles)				
6	P/M/G	2/3/4/5	fija/dinámica	hermanos/nivel
12	P/M/G	2/3/4/5/11	fija/dinámica	hermanos/nivel
24	P/M/G	2/3/4/5/23	fija/dinámica	hermanos/nivel
Arquitectura a_2 (32 núcleos disponibles)				
8	P/M/G	2/3/4/5/7	fija/dinámica	hermanos/nivel
16	P/M/G	2/3/4/5/15	fija/dinámica	hermanos/nivel
32	P/M/G	2/3/4/5/31	fija/dinámica	hermanos/nivel
Arquitectura a_3 (64 núcleos disponibles)				
16	P/M/G	2/3/4/5/15	fija/dinámica	hermanos/nivel
32	P/M/G	2/3/4/5/31	fija/dinámica	hermanos/nivel
64	P/M/G	2/3/4/5/63	fija/dinámica	hermanos/nivel

7.3. Heurística de línea base

Se seleccionó una heurística de planificación circular (conocida en inglés como Round Robin o RR) como heurística de línea base para la comparación de los resultados computados por las heurísticas de planificación propuestas en el Capítulo 6. La estrategia de planificación no considera afinidades de ningún tipo y consiste básicamente en una asignación cíclica de las unidades de procesamiento [33]. Este tipo de técnica es aplicada para la planificación de aplicaciones paralelas desarrolladas bajo el estándar MPI, el manejador de procesos Hydra [34] y el Open Run-Time Environment (Orte) [35], encargados de orquestar la ejecución de procesos para las implementaciones

MPICH [36] y Open MPI [37], respectivamente, utilizan planificación circular. Inicialmente, se asigna el proceso t_0 al núcleo c_i seleccionado de forma aleatoria. Luego, el proceso t_1 es asignado al núcleo c_{i+1}, este mecanismo continua asignado cíclicamente cada uno de los procesos restantes, como se observa en el Algoritmo 12.

Algoritmo 12 Planificador Round Robin.

```
1: solución ← inicializar vector solución
2: proceso = procesos.removerPrimero()
3: núcleo = seleccionar núcleo de forma aleatoria
4: solución[núcleo] = proceso
5: while (procesos no vacía) do
6:     proceso = procesos.removerPrimero()
7:     núcleo = seleccionar siguiente núcleo de forma cíclica
8:     solución[núcleo] = proceso
9: end while
10: return solución
```

7.4. Calibración paramétrica

Se realizó un análisis informal para determinar la configuración óptima de los valores de los parámetros para cada algoritmo de planificación implementado. Se intentó establecer los parámetros buscando un equilibrio entre la calidad de las soluciones obtenidas y la eficiencia computacional, considerando todas las instancias de evaluación generadas. A continuación, se presentan los parámetros establecidos para los algoritmos de Búsqueda en Escalada y Búsqueda Local Iterada solamente, ya que la heurística Ávida no cuenta con aspectos que puedan configurarse.

Búsqueda en Escalada Se estableció un criterio de parada, el mismo establece que la búsqueda debe detenerse cuando no existen mejores soluciones en la vecindad de la solución actual o se completen a/b iteraciones, siendo a y b el número de núcleos y de procesos a planificar, respectivamente.

Búsqueda Local Iterada Se fijó un mecanismo de perturbación y un criterio de parada. El mecanismo de perturbación consiste en intercambiar 1/6 de las asignaciones determinadas por una solución dada y el criterio de parada indica que la búsqueda debe interrumpirse al alcanzar 10 iteraciones.

7.5. Resultados numéricos

Esta sección presenta los resultados numéricos obtenidos en los experimentos de evaluación de los algoritmos propuestos para resolver el problema de planificación, analizados desde el punto de vista de la calidad de sus resultados y su eficiencia computacional. Para las pruebas se realizaron 40 ejecuciones independientes de los algoritmos de planificación sobre cada una de las instancias del problema presentadas en la sección anterior. Los resultados obtenidos se resumen y comentan en las siguientes subsecciones.

7.5.1. Resultados de los algoritmos de planificación

Las Tablas 7.3, 7.4 y 7.5 presentan el costo temporal (calculado mediante la Ecuación 6.1) asociado a las soluciones computadas por los algoritmos de planificación para cada una de las instancias del problema, detallando el costo mínimo (mejor) y el costo promedio (media), expresados en segundos. Los resultados fueron organizados de acuerdo a los tres tipos de aplicaciones paralelas implementadas, la Tabla 7.3 resume los resultados obtenidos para las instancias de tipo TC (Transferencia de calor), la Tabla 7.4 a las de tipo FT (Flujo de trabajo) y la Tabla 7.5 a las de tipo OR (Ordenamiento rápido), en todas ellas los algoritmos de planificación han sido identificados a través de sus siglas, HA (Heurística Ávida), BE (Búsqueda en Escalada) y BLI (Búsqueda Local Iterada). Las instancias de evaluación fueron identificadas mediante la nomenclatura $a \times p \times e$ donde a identifica la arquitectura, p la cantidad de procesos y e el tamaño de la entrada. Los mejores valores obtenidos son mostrados en negrita.

Tabla 7.3: Costo de las soluciones computadas para instancias de tipo TC.

Instancia	**HA**		**BE**		**BLI**	
	mejor	media	mejor	media	mejor	media
$a_1 \times 6 \times P$	**365.70**	**369.02**	**365.70**	**369.02**	**365.70**	**369.02**
$a_1 \times 6 \times M$	**1828.48**	**1845.08**	**1828.48**	**1845.08**	**1828.48**	**1845.08**
$a_1 \times 6 \times G$	**9142.39**	**9225.42**	**9142.39**	**9225.42**	**9142.39**	**9225.42**
$a_1 \times 12 \times P$	**706.62**	726.14	**706.62**	**722.64**	**706.62**	**722.64**
$a_1 \times 12 \times M$	3559.02	3620.34	**3533.09**	**3613.19**	**3533.09**	**3613.19**
$a_1 \times 12 \times G$	17838.59	18170.94	**17665.44**	**18065.95**	**17665.44**	**18065.95**
$a_1 \times 24 \times P$	1021.27	1089.81	**998.59**	1062.22	**998.59**	**1062.21**
$a_1 \times 24 \times M$	5048.56	5420.09	**4992.97**	5310.68	**4992.97**	**5310.40**
$a_1 \times 24 \times G$	**24964.86**	27066.69	24968.69	26555.23	24972.53	**26551.50**
$a_2 \times 8 \times P$	530.97	538.65	**483.15**	**490.89**	**483.15**	**490.89**
$a_2 \times 8 \times M$	2655.59	2712.47	**2415.74**	**2454.43**	**2415.74**	**2454.43**
$a_2 \times 8 \times G$	12243.26	12799.89	**12078.72**	**12272.13**	**12078.72**	**12272.13**
$a_2 \times 16 \times P$	877.08	917.38	**876.94**	**908.30**	**876.94**	**908.30**
$a_2 \times 16 \times M$	4522.86	4670.68	4386.86	4542.26	**4384.69**	**4541.48**
$a_2 \times 16 \times G$	21931.64	23097.84	**21923.45**	22707.46	**21923.45**	**22707.42**
$a_2 \times 32 \times P$	965.92	1075.18	942.86	1062.33	**942.84**	**1062.32**
$a_2 \times 32 \times M$	4815.70	5380.60	4714.72	5312.32	**4712.83**	**5311.75**
$a_2 \times 32 \times G$	24138.48	27026.85	23570.94	26560.18	**23567.88**	**26558.45**
$a_3 \times 16 \times P$	**889.33**	919.18	**889.33**	**917.81**	**889.33**	**917.81**
$a_3 \times 16 \times M$	4453.48	4594.27	**4446.63**	**4589.05**	**4446.63**	**4589.05**
$a_3 \times 16 \times G$	22318.82	22981.59	**22233.16**	**22945.25**	**22233.16**	**22945.25**
$a_3 \times 32 \times P$	977.76	1085.61	963.21	**1073.57**	**963.19**	1073.59
$a_3 \times 32 \times M$	4909.16	5436.80	4816.53	5368.62	**4815.97**	**5367.86**
$a_3 \times 32 \times G$	**24081.79**	27092.90	24082.59	**26840.03**	24084.07	26840.10
$a_3 \times 64 \times P$	1397.12	1828.68	**1383.69**	**1819.76**	1383.78	1819.85
$a_3 \times 64 \times M$	6971.88	9158.18	6919.26	9099.47	**6918.52**	**9098.81**
$a_3 \times 64 \times G$	35030.16	45924.26	**34591.41**	45497.35	34592.84	**45495.48**

Tabla 7.4: Costo de las soluciones computadas para instancias de tipo FT.

Instancia	**HA**		**BE**		**BLI**	
	mejor	media	mejor	media	mejor	media
$a_1 \times 6 \times P$	329.32	329.81	**290.52**	300.30	**290.52**	**300.09**
$a_1 \times 6 \times M$	1646.62	1646.62	**1466.64**	**1507.32**	**1466.64**	**1507.32**
$a_1 \times 6 \times G$	**7246.96**	7740.03	**7246.96**	**7493.50**	**7246.96**	**7493.50**
$a_1 \times 12 \times P$	699.01	752.27	606.45	692.35	**605.81**	**692.01**
$a_1 \times 12 \times M$	3302.92	3382.68	3129.64	3130.07	**3107.89**	**3120.40**
$a_1 \times 12 \times G$	16216.47	16956.49	15766.80	15964.90	**15319.50**	**15629.68**
$a_1 \times 24 \times P$	1465.15	1532.66	1438.78	1504.71	**1432.42**	**1501.53**
$a_1 \times 24 \times M$	7505.61	7704.23	7392.24	7621.42	**7142.90**	**7493.64**
$a_1 \times 24 \times G$	40967.34	42088.09	40474.28	41710.68	**40008.18**	**41451.46**
$a_2 \times 8 \times P$	441.95	446.56	**398.31**	**405.15**	**398.31**	**405.15**
$a_2 \times 8 \times M$	**1895.48**	1938.44	**1895.48**	1922.63	**1895.48**	**1920.08**
$a_2 \times 8 \times G$	10540.70	10772.04	10072.65	10154.36	**9786.54**	**9992.20**
$a_2 \times 16 \times P$	822.49	827.55	806.03	808.47	**802.32**	**803.59**
$a_2 \times 16 \times M$	3597.02	3896.69	3389.74	3738.41	**3309.67**	**3681.86**
$a_2 \times 16 \times G$	18003.45	19607.46	17205.05	18858.96	**16594.33**	**18515.39**
$a_2 \times 32 \times P$	2008.42	2036.23	1887.21	1965.58	**1885.65**	**1963.11**
$a_2 \times 32 \times M$	8933.85	9300.20	**8883.68**	**9249.25**	8923.53	9261.54
$a_2 \times 32 \times G$	44350.24	48259.05	**43852.77**	**47909.76**	44074.73	47966.72
$a_3 \times 16 \times P$	839.13	860.16	829.80	844.13	**826.52**	**842.49**
$a_3 \times 16 \times M$	3989.95	4271.74	3556.51	3968.43	**3473.34**	**3902.44**
$a_3 \times 16 \times G$	18033.55	20711.77	**17212.98**	19753.56	17785.99	**19457.72**
$a_3 \times 32 \times P$	2073.31	2135.11	2027.23	2099.22	**2016.28**	**2093.43**
$a_3 \times 32 \times M$	9739.15	10286.09	9462.98	9962.01	**9325.59**	**9830.25**
$a_3 \times 32 \times G$	49834.79	52560.41	**46845.66**	50554.16	47016.38	**50461.01**
$a_3 \times 64 \times P$	4245.73	4252.81	**4096.52**	**4139.58**	4164.78	4213.08
$a_3 \times 64 \times M$	19416.45	19674.60	19448.93	**19650.79**	**19405.40**	19660.92
$a_3 \times 64 \times G$	97722.26	98820.56	**95328.29**	**95749.37**	96724.57	96732.52

Tabla 7.5: Costo de las soluciones computadas para instancias de tipo OR.

Instancia	**HA**		**BE**		**BLI**	
	mejor	media	mejor	media	mejor	media
a_1×6×P	**464.60**	**515.07**	**464.60**	**515.07**	**464.60**	**515.07**
a_1×6×M	**2305.99**	**2443.67**	**2305.99**	**2443.67**	**2305.99**	**2443.67**
a_1×6×G	**11617.61**	**12537.25**	**11617.61**	**12537.25**	**11617.61**	**12537.25**
a_1×12×P	**628.68**	644.64	**628.68**	644.64	**628.68**	**643.49**
a_1×12×M	3449.97	3905.35	3350.23	3826.49	**3336.49**	**3819.61**
a_1×12×G	19720.42	20738.95	19309.23	20387.82	**19107.45**	**20286.93**
a_1×24×P	1230.29	1243.95	1213.54	1228.65	**1195.26**	**1197.50**
a_1×24×M	5098.30	5558.03	5042.83	5504.55	**5033.69**	**5491.52**
a_1×24×G	24591.24	27668.46	24345.38	27296.50	**24289.13**	**27234.21**
a_2×8×P	531.51	549.56	**512.62**	**534.67**	**512.62**	**534.67**
a_2×8×M	2613.34	2621.63	**2558.91**	**2559.48**	**2558.91**	**2559.48**
a_2×8×G	13537.04	13581.69	**13063.60**	13115.51	**13063.60**	**13103.13**
a_2×16×P	786.06	802.32	781.91	794.99	**752.42**	**779.81**
a_2×16×M	4182.23	4530.14	4096.47	4482.91	**4055.14**	**4458.96**
a_2×16×G	18996.22	19716.99	18710.12	19278.96	**18708.66**	**19269.55**
a_2×32×P	1131.98	1136.05	1118.65	1120.69	**1117.11**	**1117.49**
a_2×32×M	5465.69	5790.19	**5427.79**	5754.76	5448.33	**5748.36**
a_2×32×G	26650.95	27629.16	**25869.47**	**27106.18**	25892.77	27110.62
a_3×16×P	808.59	841.33	805.12	829.23	**804.89**	**829.11**
a_3×16×M	4332.34	4731.16	4253.79	4680.83	**4227.58**	**4667.73**
a_3×16×G	20108.51	20551.22	**20071.92**	20440.59	**20071.92**	**20401.13**
a_3×32×P	1197.28	1198.87	1172.99	1174.34	**1160.28**	**1164.07**
a_3×32×M	5771.02	6148.55	5722.93	6109.50	**5719.02**	**6010.32**
a_3×32×G	27131.20	28458.47	**27051.08**	**28148.04**	27078.98	28166.29
a_3×64×P	1381.02	1416.26	**1369.42**	**1403.24**	1393.29	1412.81
a_3×64×M	**6768.28**	7150.29	6801.72	**7144.23**	6845.14	7167.96
a_3×64×G	37545.26	38306.77	37239.68	38064.58	**37154.68**	**37928.99**

Las tablas presentadas anteriormente, muestran que el algoritmo BLI obtiene los mejores resultados en casi la totalidad de las instancias. Los resultados conseguidos por el algoritmo BE no son tan buenos como los obtenidos con BLI, aunque sí cercanos. Además, se observa que la calidad de las soluciones obtenidas mediante el método de Búsqueda en Escalada empeora a medida que el tamaño de las instancias aumenta, obteniendo resultados inferiores a los obtenidos con la heurística BLI, lo cual es lógico ya que BLI propone mejorar la técnica BE mediante la exploración del espacio de soluciones, posibilitando encontrar mejores planificaciones. La heurística Ávida presenta los peores resultados, reportando costos que superan ampliamente los mínimos obtenidos por los demás algoritmos, sobre todo para las instancias más grandes. Por otra parte, los mejores resultados de la heurística Ávida fueron reportados para las instancias de tipo TC, donde la baja heterogeneidad de los patrones de interacción aumenta la eficacia del algoritmo.

7.5.2. Resultados comparativos

En esta sección se presenta un estudio comparativo donde se analiza la mejora porcentual obtenida por los algoritmos de planificación por afinidad HA, BE y BLI sobre la calidad de las soluciones respecto a los resultados conseguidos por el algoritmo de planificación circular. El porcentaje de mejora asociado a una solución obtenida mediante una planificación por afinidad (PA) respecto a otra obtenida mediante una planificación circular (PC) serán calculados de acuerdo a la Ecuación 7.1. La Tabla 7.6 resume los porcentajes de mejora y la desviación estándar para cada una de las instancias del problema agrupadas por arquitectura y cantidad de procesos, bajo la nomenclatura $a \times p$ donde a identifica la arquitectura y p la cantidad de procesos, los mejores valores alcanzados se muestran en negrita.

$$mejora = \frac{PC - PA}{PC} \times 100 \tag{7.1}$$

Tabla 7.6: Mejora porcentual alcanzada por las heurísticas de optimización respecto a la de línea base.

Instancia	**HA**		**BE**		**BLI**	
	media	σ	media	σ	media	σ
			TC			
$a_1 \times 6$	**19.33 %**	0.56 %	**19.33 %**	0.56 %	**19.33 %**	0.56 %
$a_1 \times 12$	18.03 %	1.10 %	**18.45 %**	1.24 %	**18.45 %**	1.24 %
$a_1 \times 24$	6.63 %	1.06 %	8.43 %	1.07 %	**8.44 %**	1.07 %
$a_2 \times 8$	15.08 %	2.81 %	**19.49 %**	0.31 %	**19.49 %**	0.31 %
$a_2 \times 16$	10.74 %	1.62 %	**12.39 %**	0.99 %	**12.39 %**	0.98 %
$a_2 \times 32$	4.34 %	0.56 %	5.90 %	0.24 %	**5.91 %**	0.24 %
$a_3 \times 16$	18.82 %	0.49 %	**18.95 %**	0.47 %	**18.95 %**	0.47 %
$a_3 \times 32$	14.40 %	0.41 %	**15.25 %**	0.53 %	**15.25 %**	0.54 %
$a_3 \times 64$	5.23 %	0.42 %	6.05 %	0.18 %	**6.06 %**	0.17 %
			FT			
$a_1 \times 6$	9.31 %	3.63 %	**13.19 %**	0.20 %	**13.19 %**	0.18 %
$a_1 \times 12$	1.99 %	1.74 %	8.05 %	0.52 %	**9.65 %**	1.51 %
$a_1 \times 24$	8.48 %	0.31 %	9.35 %	0.68 %	**10.04 %**	0.48 %
$a_2 \times 8$	1.50 %	6.65 %	6.56 %	3.00 %	**7.79 %**	2.93 %
$a_2 \times 16$	5.35 %	0.69 %	8.95 %	0.90 %	**10.53 %**	1.47 %
$a_2 \times 32$	8.24 %	0.84 %	**8.97 %**	1.29 %	8.86 %	1.39 %
$a_3 \times 16$	10.44 %	4.26 %	14.87 %	2.23 %	**16.13 %**	1.60 %
$a_3 \times 32$	10.33 %	1.46 %	13.60 %	0.84 %	**13.92 %**	0.40 %
$a_3 \times 64$	11.67 %	0.40 %	**13.98 %**	1.13 %	13.21 %	1.00 %
			OR			
$a_1 \times 6$	**16.39 %**	4.30 %	**16.39 %**	4.30 %	**16.39 %**	4.30 %
$a_1 \times 12$	3.85 %	1.28 %	5.49 %	2.20 %	**5.90 %**	2.35 %
$a_1 \times 24$	3.87 %	0.54 %	5.10 %	0.63 %	**5.40 %**	1.72 %
$a_2 \times 8$	3.86 %	1.84 %	6.98 %	1.52 %	**7.05 %**	1.52 %
$a_2 \times 16$	6.25 %	0.56 %	8.10 %	1.06 %	**8.28 %**	1.01 %
$a_2 \times 32$	5.03 %	2.68 %	6.61 %	2.65 %	**6.62 %**	2.51 %
$a_3 \times 16$	10.25 %	1.63 %	10.85 %	1.87 %	**11.03 %**	1.89 %
$a_3 \times 32$	9.34 %	2.62 %	10.29 %	2.09 %	**10.52 %**	1.71 %
$a_3 \times 64$	6.71 %	0.64 %	7.23 %	0.89 %	**7.43 %**	0.73 %

Al analizar los porcentaje de mejora presentados en la Tabla 7.6, la tabla muestra que la calidad de los resultados obtenidos mediante las heurísticas de optimización superan de forma considerable a los conseguidos a través de la heurística de línea base. El planificador BLI resulta ser el más eficaz, obteniendo mejoras superiores al 5.4 %, que en algunos casos alcanzan el 19.49 %. La calidad de los resultados obtenidos por la heurística BE es levemente inferior a la conseguida mediante BLI, las mejoras conseguidas oscilan entre el 5.1 % y 19.49 %. Las mejoras reportadas por el algoritmo Ávido resultan ser las más pobres, variando entre el 1.5 % y el 19.33 %. Las desviaciones estándar obtenidas fueron generalmente reducidas, excepto para instancias de tipo OR, donde los algoritmos presentaron un comportamiento poco consistente y obtuvieron los porcentajes de mejora más bajos. Por otro lado, se observa que los mejores resultados fueron alcanzados para las instancias más pequeñas, en las cuales la mayoría de los procesos que componen la aplicación pueden ser ubicados en un socket o en un nodo NUMA, donde los costos de interacción son más bajos. Seguidamente, se presentan en la Gráfica 7.10, los porcentajes de mejora promedio agrupados de acuerdo al tipo de aplicación.

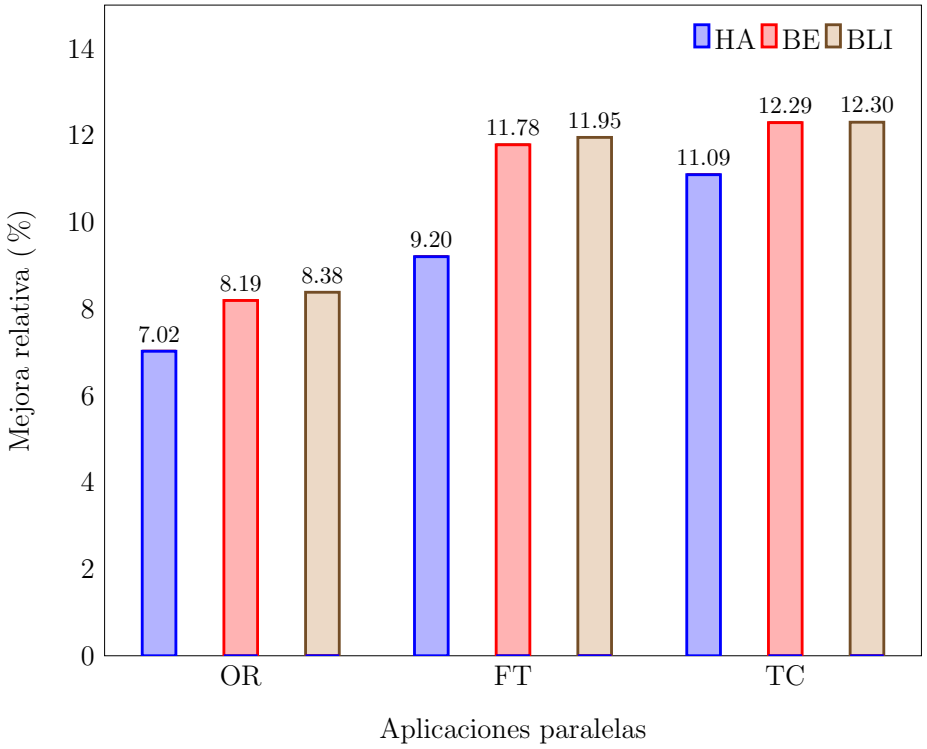

Figura 7.10: Mejora porcentual obtenida respecto a la heurística de línea base mediante las heurísticas de optimización agrupadas por tipo de instancia.

En la gráfica anterior se puede observar que los algoritmos propuestos obtuvieron mejoras porcentuales que oscilan entre el 11.09 % y el 12.3 % para las instancias de tipo TC, 9.2 % y el 11.95 % para las de tipo FT y 7.02 % y el 8.38 % para las de tipo OR. Las mejoras de calidad más amplias fueron las obtenidos para aplicaciones de tipo TC donde la homogeneidad presente en los patrones de interacción y la gran cantidad de sincronizaciones (interacción más costosa, de acuerdo a la Tabla 7.1) producto del esquema iterativo utilizado en las aplicaciones de este tipo, permiten alcanzar resultados de mayor calidad. Por otro lado, se observa que para las instancias de tipo OR se obtuvieron los porcentajes de mejora más bajos, esto puede deberse a la forma de árbol en la que son organizados los procesos en este tipo de aplicaciones, donde los datos son distribuidos por diversos canales (ramas), lo que provoca una disminución considerable en el margen de mejora.

7.5.3. Comparación con cotas inferiores

Siendo que el problema de planificación en sistemas heterogéneos no puede ser resuelto en tiempos razonables mediante método exactos, se propone utilizar la heurística BLI con la finalidad de obtener cotas inferiores para este problema, a través de las mismas se podrá medir la efectividad de las heurísticas de optimización propuestas para resolver el problema considerado en este trabajo. Las cotas inferiores fueron computadas resolviendo cada una de las instancias mediante la aplicación de la heurística BLI, estableciendo previamente un criterio de parada de diez minutos. La comparación consiste en calcular la diferencia entre el porcentaje de mejora computada como cota inferior (CI) y el porcentaje de mejora obtenido mediante una planificación por afinidad (PA), tal como se observa en la Ecuación 7.2. A continuación, se presenta la Tabla 7.7 con los márgenes de mejora potencial agrupados por arquitectura y cantidad de procesos, utilizando la nomenclatura $a \times p$ donde a identifica la arquitectura y p la cantidad de procesos. Los márgenes de mejora más reducidos aparecen en negrita.

$$margen = mejora(CI) - mejora(PA) \tag{7.2}$$

Tabla 7.7: Márgenes de mejora potencial agrupados por arquitectura y cantidad de procesos.

Instancia	HA	BE	BLI
		TC	
a_1×6	**0.00 %**	**0.00 %**	**0.00 %**
a_1×12	0.42 %	**0.00 %**	**0.00 %**
a_1×24	1.84 %	0.04 %	**0.03 %**
a_2×8	4.41 %	**0.00 %**	**0.00 %**
a_2×16	1.65 %	**0.00 %**	**0.00 %**
a_2×32	1.61 %	0.05 %	**0.04 %**
a_3×16	0.12 %	**0.00 %**	**0.00 %**
a_3×32	0.87 %	**0.02 %**	**0.02 %**
a_3×64	0.83 %	**0.00 %**	**0.00 %**
		FT	
a_1×6	4.48 %	0.61 %	**0.60 %**
a_1×12	7.75 %	1.69 %	**0.09 %**
a_1×24	2.87 %	2.00 %	**1.30 %**
a_2×8	6.30 %	1.24 %	**0.01 %**
a_2×16	5.39 %	1.79 %	**0.21 %**
a_2×32	2.84 %	**2.12 %**	2.22 %
a_3×16	7.37 %	2.95 %	**1.69 %**
a_3×32	5.19 %	1.92 %	**1.61 %**
a_3×64	2.55 %	**0.24 %**	1.01 %
		OR	
a_1×6	**0.00 %**	**0.00 %**	**0.00 %**
a_1×12	2.05 %	0.42 %	**0.00 %**
a_1×24	3.84 %	2.61 %	**2.31 %**
a_2×8	3.19 %	0.07 %	**0.00 %**
a_2×16	2.46 %	0.62 %	**0.44 %**
a_2×32	4.02 %	2.44 %	**2.43 %**
a_3×16	0.97 %	0.37 %	**0.19 %**
a_3×32	1.66 %	0.71 %	**0.48 %**
a_3×64	2.08 %	1.56 %	**1.35 %**

En la Tabla 7.7 se puede apreciar que los márgenes de mejora más amplios fueron obtenidos para el algoritmo HA, alcanzando como máximo el 7.75 %. Las otros dos algoritmos presentaron márgenes de mejora más reducidos, alcanzando un margen de mejora máximo del 2.95 % para el caso de BE y 2.43 % para BLI, lo que se implica una alta precisión en las soluciones computadas por los mismos. En la Gráfica 7.11, se pueden observar los márgenes de mejora potencial agrupados en base al tipo de instancia.

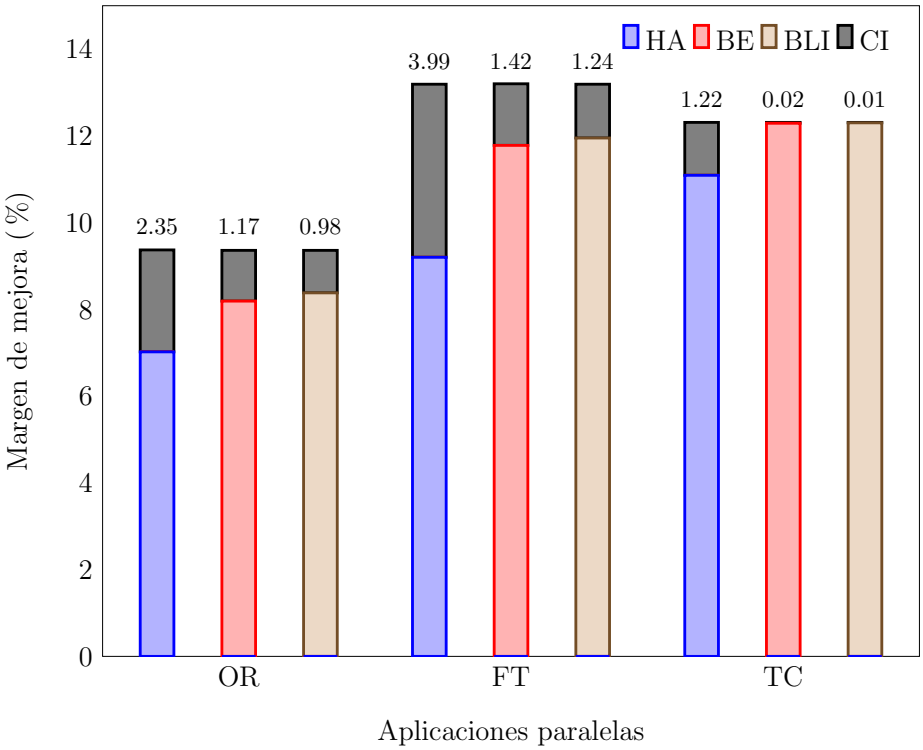

Figura 7.11: Margen de mejora potencial calculados sobre los 3 tipos de instancia respecto a las cotas inferiores.

En la gráfica anterior, se puede apreciar que los márgenes de mejora calculados para el algoritmo HA alcanzan casi el 4.0 % para las instancias de tipo FT y como era de esperar, resultó ser más efectivo para las aplicaciones de tipo TC, alcanzando el 1.22 %. Por otro lado, la heurística BE y la metaheurística BLI presentan márgenes de mejora inferiores al 1.5 %, volviéndose estos casi nulos para instancias de tipo TC, lo que demuestra la gran eficacia de los mismos para resolver este tipo de instancias.

7.5.4. Análisis de la eficiencia computacional

En esta sección se realiza un análisis de la eficiencia computacional que presentan los tres algoritmos de planificación implementados. En la Tabla 7.8 se resumen los tiempos promedio de ejecución (en microsegundos) sobre cuarenta ejecuciones independientes realizadas para cada instancia del problema. Las mismas fueron identificadas de acuerdo a la nomenclatura $a \times p$ donde a identifica la arquitectura y p la cantidad de procesos.

Tabla 7.8: Eficiencia computacional (μs).

Instancia	**HA**	**BE**	**BLI**
$a_1 \times 6$	20.90	34.62	141.43
$a_1 \times 12$	49.03	198.28	1652.06
$a_1 \times 24$	122.14	894.31	7768.94
$a_2 \times 8$	34.72	91.89	501.82
$a_2 \times 16$	93.95	509.70	4268.33
$a_2 \times 32$	248.70	2044.53	17990.45
$a_3 \times 16$	187.01	801.02	5834.98
$a_3 \times 32$	574.04	4125.72	36202.02
$a_3 \times 64$	1627.31	16732.47	151770.69

Analizando los tiempos de ejecución, queda de manifiesto que el algoritmo HA es el más rápido, esta situación puede ser explicada como consecuencia de que el algoritmo se limita a construir la solución y luego no aplica métodos para mejorarla. Además, se observa que los tiempos de ejecución tienden a crecer linealmente en relación a los aumentos en la cantidad de procesos, esto resulta lógico, ya que encontrar soluciones más grandes implica realizar más iteraciones, lo que produce un aumento en el tiempo de cómputo. Pese a ser el algoritmo más rápido, la calidad de las soluciones reportadas por HA son inferiores a las obtenidas por BE y BLI, sobre todo en instancias de tipo FT. El algoritmo BE presenta tiempos de ejecución superiores a los de HA, esta diferencia se acentúa notoriamente en las instancias más grandes. Esto responde a que la heurística BE plantea mejorar la solución actual analizando exhaustivamente un conjunto de soluciones vecinas (vecindario) que crece de forma potencial a media que la cantidad de procesos aumenta. El algoritmo BLI resulta ser el más lento, reporta los tiempos de ejecución

más altos, aproximadamente diez veces más altos que los reportados por BE. Esto se explica porque el proceso iterativo de mejora utilizado por BLI implica perturbar la solución actual y luego mejorarla aplicando Búsqueda Local, además, recordemos que el criterio de parada utilizado para este caso fue de diez iteraciones, lo que coincide con el factor de diferencia. De acuerdo a lo observado, se hace notoria la lentitud del algoritmo de BE, pese a esto, el algoritmo resulta ser más efectivo que los algoritmos de planificación HA y RR cuando se analizan los tiempos de ejecución y la calidad de las soluciones de forma conjunta.

Capítulo 8

Conclusiones y trabajo futuro

Este capítulo resume las conclusiones del trabajo presentado en los capítulos anteriores. Adicionalmente, se presentan las líneas de investigación que han dejado aspectos abiertos o interesantes y que merecen una profundización en el futuro.

8.1. Conclusiones

Este trabajo está dirigido al estudio del problema de la planificación por afinidad en sistemas multinúcleo. Las principales contribuciones consisten en la presentación de una formulación matemática del problema, la elaboración de mecanismos para la caracterización de aplicaciones paralelas y sistemas multinúcleo, y el diseño e implementación de tres algoritmos de planificación. La formulación del problema introducida en este trabajo considera, entre otros aspectos, la cantidad de operaciones de comunicación y sincronización efectuadas entre procesos, así como también el costo que implica efectuar cada una de ellas, el cual depende de la plataforma de ejecución. Por lo tanto, será necesario aplicar mecanismos de caracterización sobre las aplicaciones y los sistemas de cómputo. La caracterización de aplicaciones paralelas se elaboró en base a una clasificación previamente realizada, en la que se identificaron tres topologias: plana, dirigida por la aplicación y jerárquica. En base a esta clasificación, se definió un mecanismo de caracterización interactivo que permite modelar las afinidades entre los procesos que componen una aplicación paralela. Se diseño un mecanismo de caracterización para sistemas multinúcleo, el mismo se basa en la utilización de la herramienta HwLoc (Hardware Locality) [38] para obtener la información cualitativa del computador y un benchmark de tipo ping-pong [27] para evaluar los costos de interacción entre las unidades de cómputo. Se propusieron

tres heurísticas de optimización para la resolución del problema: Heurística Ávida (en inglés, Greedy Heuristic), Búsqueda en Escalada (en inglés, Hill Climbing) y Búsqueda Local Iterada (en inglés, Iterated Local Search). Los algoritmos fueron evaluados desde el punto de vista de la calidad de las soluciones y la eficiencia computacional, y además se realizaron estudios comparativos respecto a una heurística de planificación circular (en inglés, Round Robin) que no considera afinidades. Para la evaluación empírica de las técnicas de planificación seleccionadas se realizó la caracterización cualitativa y cuantitativa de tres arquitecturas de cómputo reales, pertenecientes a la plataforma de alto desempeño Cluster FING propiedad de la Facultad de Ingeniería, Universidad de la República, Uruguay [29]. Además, se caracterizaron tres aplicaciones que fueron previamente diseñadas bajo la premisa de que cada una de ellas represente a una de las topologias identificadas en la clasificación de aplicaciones paralelas. El análisis empírico realizado reflejó que las heurísticas de búsqueda (escalada e iterada) son capaces de obtener soluciones de buena calidad en tiempos de ejecución reducidos, que no superan el segundo, inclusive para las instancias de prueba más grandes. En cuanto a la calidad de las soluciones, las heurísticas de búsqueda presentaron resultados superiores en promedio a los obtenidos mediante planificaciones ávidas y circulares, conservando esta superioridad en la totalidad de los experimentos. Particularmente, la heurística de Búsqueda en Escalada resultó ser más efectiva que la técnicas de planificación ávida y de planificación circular al considerar conjuntamente la calidad de las soluciones y los tiempos de ejecución. Por otro parte, el análisis posibilitó identificar varios aspectos y características del problema estudiado, así como también sobre el comportamiento de los algoritmos al momento de resolver problemas de este tipo.

8.2. Trabajo futuro

Las principales lineas de trabajo futuro se enfocan en cuatro aspectos: mejorar la eficacia de los algoritmos de planificación, ampliar el conjunto de instancias de prueba, extender la solución planteada a sistemas de cómputo distribuidos e integrar los algoritmos de planificación a sistemas reales de manejo de recursos. Los resultados obtenidos en la comparación con cotas inferiores permitieron detectar que existe un margen de mejora considerable para algunos de los algoritmos de planificaciones propuestos. Por ello, la primera línea se orienta a mejorar la eficacia de las soluciones computadas, por ejemplo, a través de un análisis experimental que permita hallar la configuración óptima de los parámetros utilizados en cada uno de los algoritmos de planificación. La segunda línea consiste en ampliar el conjunto de instancias

de prueba, incorporando instancias que se correspondan con el modelo de programación paralela BSP [39], y accediendo a sistemas multinúcleo que permitan la generación de instancias del problema de mayor tamaño que las consideradas en este trabajo. La tercer línea se orienta a extender la solución planteada a sistemas distribuidos, por lo tanto, será necesario trabajar sobre un mecanismo de caracterización de infraestructuras de cómputo distribuidas. Esto podría realizarse con la herramienta HwLoc[23] que entre otras cosas permite combinar topologias provenientes de distintos nodos en una única topología [40] o con la herramienta NetLoc(Portable Network Locality) [41] que ofrece una representación abstracta de un sistema distribuido, proporcionando información de la topología de red y de cada uno de los nodos del sistema. Finalmente, la cuarta línea refiere a integrar los algoritmos de planificación desarrollados a sistemas reales de manejo de recursos, con la finalidad de aplicar los algoritmos desarrollados para la resolución de problemas reales en sistemas heterogéneos de gran escala.

Bibliografía

[1] I. Foster y C. Kesselman: *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[2] H. El-Rewini, T. G. Lewis y H. H. Ali: *Task Scheduling in Parallel and Distributed Systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.

[3] J. Leung, L. Kelly y J. H. Anderson: *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. CRC Press, Inc., Boca Raton, FL, USA, 2004.

[4] M. M. Eshaghian: *Heterogeneous Computing*. Artech House computer science library. Artech House, 1996.

[5] M. R. Garey y D. S. Johnson: *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[6] C. Blum y A. Roli: *Metaheuristics in Combinatorial Optimization: Overview and Conceptual Comparison*. ACM Comput. Surv., 35(3):268–308, 2003.

[7] Z. Majo y T. R. Gross: *Memory Management in NUMA Multicore Systems: Trapped Between Cache Contention and Interconnect Overhead*. En *Proceedings of the International Symposium on Memory Management*, páginas 11–20, New York, NY, USA, 2011.

[8] J. Torrellas, A. Tucker y A. Gupta: *Benefits of Cache-affinity Scheduling in Shared-memory Multiprocessors: A Summary*. En *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, páginas 272–274, New York, NY, USA, 1993.

[9] S. Subramaniam y D. L. Eager: *Affinity Scheduling of Unbalanced Workloads*. En *Proceedings of the ACM/IEEE Conference on Supercomputing*, páginas 214–226, Los Alamitos, CA, USA, 1994.

[10] Y.-W. Fann, C.-T. Yang, C.-J. Tsai y S.-S. Tseng: *IPLS: An Intelligent Parallel Loop Scheduling for Multiprocessor Systems*. International Conference on Parallel and Distributed Systems, 0:775–782, 1998.

[11] Y.-M. Wang, H.-H. Wang y R.-C. Chang: *Clustered affinity scheduling on large-scale NUMA multiprocessors*. Journal of Systems and Software, 39(1):61–70, 1997.

[12] Y.-M. Wang, H.-H. Wang y R.-C. Chang: *Hierarchical loop scheduling for clustered NUMA machines*. Journal of Systems and Software, 55(1):33–44, 2000.

[13] S. Iturriaga y S. Nesmachnow: *Evolutionary algorithms for affinity scheduling heuristics in heterogeneous computing systems*. En *XL Latin American Computing Conference, Montevideo, Uruguay*, páginas 1–12, 2014.

[14] E. P. Markatos y T. J. LeBlanc: *Using Processor Affinity in Loop Scheduling on Shared-memory Multiprocessors*. En *Proceedings of the ACM/IEEE Conference on Supercomputing*, páginas 104–113, Los Alamitos, CA, USA, 1992.

[15] B. Hamidzadeh y D. J. Lilja: *Dynamic Scheduling Strategies for Shared-memory Multiprocessors*. En *Proceedings of the 16th International Conference on Distributed Computing Systems, Hong Kong*, páginas 208–215, 1996.

[16] B. Hamidzadeh, L. Y. Kit y D. J. Lilja: *Dynamic Task Scheduling Using Online Optimization*. IEEE Trans. Parallel Distrib. Syst., 11(11):1151–1163, 2000.

[17] C. H. Papadimitriou y K. Steiglitz: *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1982.

[18] P. Brucker: *Scheduling Algorithms*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 3era edición, 2001.

[19] M. A. Lones: *Sean Luke: essentials of metaheuristics.* Genetic Programming and Evolvable Machines, 12(3):333–334, 2011.

[20] E. Aarts y J. K. Lenstra: *Local Search in Combinatorial Optimization*. John Wiley & Sons, Inc., New York, NY, USA, 1era edición, 1997.

[21] H. R. Lourenço, O. C. Martin y T. Stützle: *Iterated local search*. En *Handbook of Metaheuristics, International Series in Operations Research and Management Science*, volumen 57, páginas 321–353, 2002.

[22] *SETI@home*. http://www.setiathome.ssl.berkeley.edu, Consultado en enero 2015.

[23] A. Adamatzky: *Game of Life Cellular Automata*. Springer Publishing Company, Incorporated, 1era edición, 2010.

[24] V. Kumar: *Introduction to Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2da edición, 2002.

[25] B. Goglin: *Understanding and managing hardware affinities on hierarchical platforms With Hardware Locality*. https://www.open-mpi.org/projects/hwloc/tutorials/20130121-HiPEAC-hwloc-tutorial.pdf, Consultado en julio 2015.

[26] *Portable Hardware Locality (HwLoc)*. http://www.open-mpi.org/projects/hwloc, Consultado en agosto 2014.

[27] R. H. Reussner, P. Sanders, L. Prechelt y M. Müller: *SKaMPI: A Detailed, Accurate MPI Benchmark*. En *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 5th European PVM/MPI Users' Group Meeting, Liverpool, UK*, páginas 52–59, 1998.

[28] W. Grop, E. Lusk y A. Skjellum: *Using MPI: Portable Parallel Programming with Message-Passing Interface*. MIT Press, Cambridge, Massachusetts, 2da edición, 1994.

[29] S. Nesmachnow: *Computación científica de alto desempeño en la Facultad de Ingeniería, Universidad de la República*. Revista de la Asociación de Ingenieros del Uruguay, 61:12–15, 2010.

[30] L. F. Mamoade: *Parallel Numerical Solution of the 2D Heat Equation*. http://www.bcamath.org/documentos_public/archivos/actividades/presentatio.pdf, Consultado en septiembre 2015.

[31] P. Czarnul: *A Workflow Application for Parallel Processing of Big Data from an Internet Portal*. En *Proceedings of the International Conference on Computational Science, Cairns, Queensland, Australia*, páginas 499–508, 2014.

[32] S. Pandey, W Voorsluys, M. Rahman, R. Buyya, J. E. Dobson y K. Chiu: *Brain Image Registration Analysis Workflow for fMRI Studies on Global Grids*. En *The IEEE 23rd International Conference on Advanced Information Networking and Applications, Bradford, United Kingdom*, páginas 435–442, 2009.

[33] W Stallings: *Operating Systems - Internals and Design Principles*. Pitman, 2011.

[34] *Hydra Process Management Framework*. https://wiki.mpich.org/mpich/index.php/Hydra_Process_Management_Framework, Consultado en agosto 2014.

[35] R. H. Castain, T. S. Woodall, D. J. Daniel, J. M. Squyres, B. Barrett y G. E. Fagg: *The Open Run-Time Environment (OpenRTE): A Transparent Multi-cluster Environment for High-Performance Computing*. En *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 12th European PVM/MPI Users' Group Meeting, Sorrento, Italy*, páginas 225–232, 2005.

[36] *MPICH*. http://www.mpich.org, Consultado en julio 2015.

[37] *Open MPI*. http://www.open-mpi.org, Consultado en julio 2015.

[38] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault y R. Namyst: *hwloc: A Generic Framework for Managing Hardware Affinities in HPC Applications*. En *Proceedings of the 18th Euromicro Conference on Parallel, Distributed and Network-based Processing, Pisa, Italy*, páginas 180–186, 2010.

[39] L. G. Valiant: *A Bridging Model for Parallel Computation*. Commun. ACM, 33(8):103–111, 1990.

[40] B. Goglin: *Managing the topology of heterogeneous cluster nodes with hardware locality (hwloc)*. En *International Conference on High Performance Computing & Simulation, Bologna, Italy*, páginas 74–81, 2014.

[41] B. Goglin, J. Hursey y J. M. Squyres: *Netloc: Towards a Comprehensive View of the HPC System Topology*. En *43rd International Conference on Parallel Processing Workshops, Minneapolis, MN, USA*, páginas 216–225, 2014.