UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Solución en paralelo del problema de la mochila

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Autor: David Verdeguer López
Tutor: Pedro Alonso

Curso 2016-2017

Resumen

El problema de la mochila es un problema NP-Completo bien conocido, que, en sus diferentes variantes, tiene un gran interés, tanto teórico como práctico, y ha sido sujeto de un amplio número de estudios. Si bien la formulación del problema es sencilla, su resolución es más compleja. Dado que el problema de la mochila es un problema NP-Completo, aún con los mejores algoritmos que existen actualmente la tarea de resolver instancias del problema grandes requeriría mucho tiempo de cómputo, por lo que estos algoritmos se suelen paralelizar para reducir el tiempo que necesitan para obtener una solución.

En este trabajo se ha implementado una versión paralela del algoritmo de las dos listas desarrollado por Horowitz Y Sahni, utilizando un modelo de programación híbrido MPI/OpenMP. Para ello primero se ha implementado la versión secuencial, que luego se ha utilizado para a) implementar la versión paralela en base a ella y b) ejecutar pruebas y comparar la ganancia de rendimiento obtenida al paralelizar el algoritmo. Estas implementaciones se han realizado con el lenguaje de programación C. La implementación paralela se ha realizacion con la intención que sea ejecutada en un clúster actual, con varios nodos, cada uno teniendo disponible varios núcleos.

Para ello se ha realizado un estudio del problema de la mochila, de donde proviene y su importancia histórica, así y como se presentan las principales instancias y variaciones del problema, junto con otros problemas importantes que son derivados de, o pueden reducirse al problema de la mochila. También se presentan las dos librerías de paralelización a utilizar, MPI y OpenMP, describiendo sus características principales y los mecanismos de paralelización que utilizan, además de enumerar las principales utilidades que ofrecen para ese propósito (funciones, directivas, etc).

Palabras clave: mochila, paralelo, openmp, mpi

Resum

El problema de la motxilla és un problema NP-Complet ben conegut, que, en les seues diferents variants, té un gran interés, tant teòric com pràctic, i ha sigut subjecte d'un ampli nombre d'estudis. Si bé la formulació del problema és senzilla, la seua resolució és més complexa. Atés que el problema de la motxilla és un problema NP-Complet, encara amb els millors algoritmes que existixen actualment la tasca de resoldre instàncies grans del problema requeriria molt de temps de còmput, raó per la qual aquestos algoritmes es solen paralelizar per a reduir el temps que necessiten per a obtindre una solució.

En este treball s'ha implementat una versió paral·lela de l'algoritme de les dos llistes desenvolupat per Horowitz I Sahni, utilitzant un model de programació híbrid MPI/OpenMP. Per això primer s'ha implementat la versió seqüencial, que després s'ha utilitzat per a) implementar la versió paral·lela basant-se en ella i b) executar proves i comparar el increment de rendiment obtingut al paralelizar l'algoritme. Estes implementacions s'han realitzat amb el llenguatge de programació C. La implementació paral·lela s'ha realitzat amb l'intenció que siga executada en un cluster actual, amb diversos nodes, aquestos tenint disponible diversos nuclis.

Per a aquesta tarea s'ha realitzat un estudi del problema de la motxilla, d'on prové i la seua importància històrica, així com es presenten les principals instàncies i variacions del problema, junt amb altres problemes importants que són derivats de, o poden reduir-se al problema de la motxilla. També es presenten les dos llibreries de paralelización a utilitzar, MPI i OpenMP, descrivint les seues característiques principals i els mecanismes de paralelización que utilitzen, a més d'enumerar les principals utilitats que oferixen per a eixe propòsit (funcions, directives, etc).

Paraules clau: motxilla, paral·lel, openmp, mpi

Abstract

The knapsack problem is a well-known NP-Complete problem, which, in its different variants, has a great interest, both theoretical and practical, and has been the subject of a large number of studies. Although the formulation of the problem is simple, its resolution is more complex. Since the knapsack problem is a NP-Complete problem, even with the best algorithms that currently exist, the task of solving large instances of the problem would require a lot of computing time, so these algorithms are often parallelized to reduce the time they need to get a solution.

In this work a parallel version of the two lists algorithm developed by Horowitz and Sahni has been implemented, using an MPI/OpenMP hybrid programming model. To do this, the sequential version has been implemented, which has been used to a) implement the parallel version on top of it and b) run tests and compare the performance gained by parallelizing the algorithm. These implementations have been done with the programming language C. The parallel implementation has been developed with the intention of executing it in a modern cluster, with several nodes, each having several cores available.

For this, the knapsack problem has been studied, stating its historical importance, as well its main instances and variations of the problem are presented, together with other important problems that are derived, or can be reduced to, the knapsack problem. It is also presented the two parallelization libraries to be used, MPI and OpenMP, describing their main characteristics and the parallelization mechanisms they use, as well as listing the main utilities they offer for that purpose (functions, directives, etc.).

Key words: knapsack, parallel, openmp, mpi

Índice general

Índice de algoritmos

Índice de figuras

Índice de tablas

CAPÍTULO 1
Introducción

1.1 Transfondo

El problema de la mochila, conocido en inglés como Knapsack problem, es un problema NP-Completo de optimización combinatoria. Ha sido ampliamente estudiado a lo largo del tiempo, haciéndose referencia a él ya en el año 1897. [1] El nombre del problema de la mochila fue dado por el matemático Tobias Dantzig a principios del siglo XX. [2] El nombre del problema hace referencia al problema común de empacar tus cosas más valiosas sin llegar a sobrecargar tu equipaje, siendo modelado con una mochila capaz de soportar cierto peso, y un conjunto de objetos que se quieren introducir en está, de los que se conoce su valor y su peso. La solución a este problema es el conjunto de objetos que caben en la mochila, de tal forma que no exista otro conjunto de objetos que quepan en la mochila y tengan un valor total superior. El problema de la mochila no es un solo problema en sí, sino un término global para una familia de problemas que comparten ciertas características, que se puede expresar generalmente como:

"Dado un conjunto de entidades, cada una de ellas teniendo asociado un valor y un tamaño, se desea seleccionar uno o más subconjuntos disjuntos de tal forma que la suma de los tamaños en cada subconjunto no excede un límite dado, y la suma de los valores de las entidades seleccionadas es maximizada"

El problema de la mochila, en sus diferentes variantes, tiene un gran interés, tanto teórico como práctico, y ha sido sujeto de un amplio número de estudios. El interés práctico se deriva del hecho de que los problemas clasificados como un tipo de problema de la mochila tienen un gran número de aplicaciones en la vida real, de las cuales, algunos de los ejemplos más comunes son: presupuestos de inversión, carga de mercancías y el corte de productos. Desde el punto de vista teórico, el problema de la mochila tiene una estructura muy simple, pero que permite la aplicación de un gran número de propiedades combinatorias. Esto, unido al hecho de que otros problemas combinatorios más complejos pueden resolverse con una serie de subproblemas que entran dentro de la categoría del problema de la mochila, explican el interés teórico que despiertan estos problemas.

1.2 Motivación

El problema de la mochila tiene una gran utilidad práctica, siendo utilizado en multitud de casos reales, por lo que es indispensable que existan algoritmos que sean capaz de resolver el problema eficientemente. Debido a que el problema es NP-Completo, y según el estado actual de la teoría de la complejidad, no existe ningún algoritmo que sea capaz de resolver el problema en tiempo polinomial. Los algoritmos más populares para la re-

solución de este problema tienen una complejidad pseudopolinomial para proporcionar un resultado exacto. Existen algoritmos capaces de computar una aproximación al resultado del problema en tiempo polinómico, pero para las aplicaciones reales muchas veces una aproximación no es suficiente, siendo necesario encontrar el resultado exacto.

Debido a que con los mejores algoritmos que existen actualmente la tarea de resolver instancias del problema grandes requeriría mucho tiempo de cómputo, estos algoritmos se suelen paralelizar para reducir el tiempo que necesitan para obtener una solución. Los supercomputadores actuales suelen tomar forma de clusters, por lo que es ideal realizar una paralelización explotando las características de estos. Un cluster es un conjunto de nodos unidos por una conexión de altas prestaciones que actúan conjuntamente. Hoy en día es bastante típico que cada nodo tenga varias CPUs, cada una con varios núcleos. Para aprovechar la potencia del cluster se puede utilizar un modelo de programación híbrida MPI/OpenMP. Mientras que MPI es una librería dedicada a la paralelización en sistemas de memoria distribuida, que en los cluster se utiliza para operaciones entre los nodos, OpenMP está dedicado a la paralelización multihilo en sistemas de memoria compartida, siendo utilizado para aprovechar toda la potencia de las CPUs y núcleos disponibles en cada nodo.

1.3 Objetivos

El principal objetivo de este Trabajo de Fin de Grado es la implementación de un algoritmo paralelizado para la resolución del problema de la mochila aprovechando al máximo todos los recursos computacionales disponibles en un cluster. El algoritmo a desarrollar será probado, y se comparará el rendimiento obtenido con el rendimiento del algoritmo sin paralelizar, para comprobar que existe una reducción del tiempo de cómputo en la versión paralela respecto a la secuencial para casos de la misma talla.

Para cumplir el objetivo se empezará realizando un estudio del problema de la mochila, su historia, características principales y diferentes instancias existentes. Se implementará el algoritmo secuencial para la resolución del problema, que más adelante servirá como base para el algoritmo paralelo. El algoritmo a utilizar será el algoritmo de las dos listas. Una vez la parte secuencial este completa se empezará con la parte paralela, para la cual se investigara sobre las diferentes librerías a utilizar (OpenMP y MPI) y como se van a aplicar al proyecto. Se proseguirá implementando la versión paralela del algoritmo, utilizando las librerías mencionadas. A continuación se realizarán las pruebas pertinentes y se comparan los resultados obtenidos.

CAPÍTULO 2
El problema de la mochila

2.1 Definición formal

Por un lado se tiene la mochila en la que se van a introducir los ítems, por otro lado se tiene una lista de objetos, para generalizar llamados ítems, numerados de 1 a n. Los siguientes datos son conocidos:

- W = capacidad máxima de la mochila
- v_i = valor del i-ésimo ítem
- w_i = peso del i-ésimo ítem

En general, en la literatura se suele asumir que la capacidad, los valores y los pesos son siempre enteros positivos. Pero a pesar de ello, el problema puede ser fácilmente extendido para contemplar el caso de valores reales, y en según qué casos el uso de valores negativos.

En base a está información el problema se puede expresar matemáticamente de la siguiente forma:

$$\text{maximizar} \quad \sum_{i=1}^{n} v_i x_i$$

$$\text{sujeto a} \quad \sum_{i=1}^{n} w_i x_i <= W$$

$$x_i = 0 \text{ o } 1, \qquad i = 1, ..., n$$

Figura 2.1: Formulación matematica del problema de la mochila

La solución al problema viene dada por el vector de variables binarias $X(x_1, x_2, ..., x_n)$. Estas variables binarias tienen el siguiente significado:

$$x_i = \begin{cases} 1 & \text{si el objeto i es seleccionado} \\ 0 & \text{si el objeto i no es seleccionado} \end{cases}$$

2.2 El problema de la mochila binaria

La definición formal del problema de la mochila presentada en la sección anterior no solo representa las características genéricas de los problemas que entran dentro de la categoría del problema de la mochila, sino que también define el problema de la mochila más básico y extendido. Está versión del problema de la mochila es conocida como el problema de la mochila binaria, también llamada mochila 0/1. Este nombre viene dado por el hecho de que para cada ítem disponible solo hay dos opciones, elegir si cogerlo o no, siendo esta una decisión binaria.

De entre todas las definiciones e instancias del problema de la mochila, la más importante entre ellas el problema de la mochila binaria. Las principales razones porque esta versión del problema de la mochila es tan importante y es de las más estudiadas son las siguientes:

1. Puede ser vista como el problema de programación lineal más simple
2. Aparece como subproblema en muchos problemas más complejos
3. Es ampliamente usada en situaciones prácticas

2.3 El problema de la mochila acotado

El problema de la mochila se puede generalizar eliminando la restricción de que solo hay una unidad de cada ítem, por lo que el mismo ítem se puede escoger varias veces. Para ello se tiene que conocer otra característica de cada ítem. b_i para i 1..n, es la cota superior para el ítem i, el número máximo de copias de ese ítem que se pueden meter en la mochila.

En está instancia del problema el vector $X(x_1, x_2, ..., x_n)$ ya no es un vector de variables binarias, sino un vector de variables enteras, donde x_i representa el número de copias del i-ésimo objeto introducidas en la mochila.

La formulación matemática de esta versión del problema es la siguiente:

$$\text{maximizar} \quad \sum_{i=1}^{n} v_i x_i$$

$$\text{sujeto a} \quad \sum_{i=1}^{n} w_i x_i <= W$$

$$0 <= x_i <= b_i, \qquad i = 1, ..., n$$

Figura 2.2: Formulación matematica del problema de la mochila acotado

2.4 El problema de la mochila no acotado

El problema puede ser generalizado aún más en base a la instancia anterior, en este caso eliminando la restricción al número de copias de cada ítem que pueden ser insertadas en la mochila, pudiendo introducir un número ilimitado de copias del mismo ítem en la mochila.

En está instancia las variable bi para i 1..n desaparecen. El significado del vector solución no cambia respecto al problema de la mochila acotado, xi representa el número de copias del i-ésimo objeto introducidas en la mochila.

El problema se define de la siguiente forma:

$$\text{maximizar} \quad \sum_{i=1}^{n} v_i x_i$$

$$\text{sujeto a} \quad \sum_{i=1}^{n} w_i x_i <= W$$

$$0 <= x_i <= \text{ínf}, \qquad i = 1, ..., n$$

Figura 2.3: Formulación matematica del problema de la mochila no acotado

2.5 Problema de la suma de subconjuntos

El problema de la suma de subconjuntos es otro problema bien conocido, importante en criptografía y en la teoría de la complejidad. Este problema está directamente relacionado con el problema de la mochila, de hecho, es un caso particular de este, en concreto del problema de la mochila binaria. La principal diferencia respecto a este, es que para todos los objetos su valor es igual a su peso, de ahí que también sea conocido como el problema de la mochila independiente del valor. Otra diferencia respecto al problema de la mochila, es que en el problema de la suma de subconjuntos es más común la asunción de que los pesos (y por consecuencia los valores) pueden tener un valor negativo.

La definición del problema es la siguiente:

Dado un conjunto de números enteros N ¿es posible encontrar un subconjunto no vacío, para el cual la suma de sus elementos sea s?

La definición presentada es una definición general, pero al igual que el problema de la mochila, el problema de la suma de subconjuntos presenta variaciones en sus definiciones. Tambien es comun encontrarlo definido con el valor objetivo s fijo a 0, o relajando la condición de igualdad respecto al valor objetivo s, permitiendo soluciones cuyo valor sea menor o igual a este.

La definición formal del problema de la suma de subconjuntos es la siguiente:

Dado un conjunto de n números enteros y un valor objetivo s con:

n_i = valor del item i

El objetivo del problema es encontrar un vector de variable binarias $X(x_1, x_2, ..., x_n)$ que solucione la siguiente ecuación:

$$\sum_{i=1}^{n} w_i x_i = W, x_i \in \{0, 1\}$$

donde

$$x_i = 0 \text{ o } 1, \qquad i = 1, ..., n$$

$$x_i = \begin{cases} 1 & \text{si el objeto i es seleccionado} \\ 0 & \text{si el objeto i no es seleccionado} \end{cases}$$

Figura 2.4: Formulación matematica del problema de la suma de subconjuntos

Como se puede observar el problema de la suma de subconjuntos no es un problema de optimización combinatoria, sino que es un problema de decisión. A pesar de ello, puede resolverse de la misma forma que el problema de la mochila binaria, alterando ligeramente su definición:

$$\text{maximizar} \quad \sum_{i=1}^{n} w_i x_i$$

$$\text{sujeto a} \quad \sum_{i=1}^{n} w_i x_i <= s$$

$$x_i = 0 \text{ o } 1, \qquad i = 1, ..., n$$

Figura 2.5: Adaptacion del problema de la suma de subconjunto

A pesar de ser posible resolver el problema de la suma de subconjuntos enfocandolo como un problema de la mochila binaria, merece un trato especial, ya que existen algoritmos específicos para este problema que proporcionan mejores resultados.

2.6 Otros Problemas

Hasta ahora han sido nombrados las instancias del problema de la mochila más extendidos, así como el problema de la suma de subconjuntos, otro problema altamente importante y directamente relacionado con el problema de la mochila. Pero estas no son las únicas variaciones de este problema. Existe una familia de problemas de la mochila que su principal característica diferenciadora es que existen diferentes "mochilas.en las que introducir los ítems.

Extendiendo el problema de la mochila binaria con m contenedores, de capacidad m_j $(j = 1, ..., m)$, disponibles. Introduciendo las variables binarias x_{ij} donde un valor de 1 significa que el ítem i se ha elegido para el contenedor j, y 0 significa lo contrario, se obtendría la siguiente formulación:

$$\text{maximizar} \quad \sum_{i=1}^{n} \sum_{j=1}^{m} v_i x_{i_j}$$

sujeto a

$$\sum_{i=1}^{n} w_i x_{ij} <= W_j \qquad j = 1, ..., m$$

$$\sum_{j=1}^{m} x_{ij} <= 1 \qquad j = 1, ..., m$$

$$x_{ij} = 0 \text{ o } 1, \quad i = 1, ..., n, \quad j = 1, ..., m$$

Figura 2.6: Formulación del problema de la mochila multiple

De forma equivalente a como se generaliza el problema de la mochila binaria para permitir múltiples contenedores, se pueden expandir el resto de instancias del problema presentadas anteriormente para contemplar el uso de varias "mochilas".

También existen otros problemas, que si bien no se les categoriza directamente como problemas de la mochila, al igual que con el problema de la suma de subconjuntos, pueden ser reinterpretados como un problema de la mochila (con variaciones eso sí), y

en algunos casos tienen subproblemas que sí que entran dentro de la categoría de problemas de la mochila, y que juegan un papel decisivo en los algoritmos que resuelven el problema completo.

2.7 Complejidad Computacional

Todos los problemas introducidos hasta el momento son NP-Completos, hecho que ha sido demostrado por varios autores en la literatura. Esto implica que no existe un algoritmo que tenga una complejidad polinomial en función de n, a no ser que algun dia se demuestre que $P = NP$.

Sin embargo, para los problemas de mochila unica (un solo contenedor) existen algoritmos cuya complejidad temporal y espacial es pseudo polinomial, es decir, están acotados por un polinomio en función de n y c, por ejemplo $O(nc)$. Está complejidad no contradice el hecho de que sea NP-Completo, ya que un algoritmo tiene una complejidad pseudo-polinomial si depende de longitud de la entrada (el número de bits requeridos para representarla) y el valor numérico de la entrada. En general la longitud de la entrada es exponencial respecto a su valor numérico.

Es decir, los problemas de mochila unica son problemas débilmentente NP-Completos. Un problema es débilmentente NP-Completo si existe un algoritmo capaz de resolver el problema en tiempo polinomial en función de la dimensión del problema y la magnitud de los datos, en lugar del logaritmo en base 2 de sus magnitudes. Por lo tanto, estos algoritmos son técnicamente funciones exponenciales al tamaño de la entrada, y no son considerados algoritmos polinomiales.

Al contrario que los problemas de mochila unica, los problemas de mochilas múltiples no son débilmente NP-completos, sino que son fuertemente NP-Completos, es decir, no puede existir un algoritmo capaz de resolver estos problemas en tiempo pseudo-polinomial a no ser que se demuestre que P = NP.

CAPÍTULO 3
Implementación Secuencial

3.1 Historia

Tradicionalmente, el problema de la mochila ha estado altamente relacionado con la programación dinámica como forma de resolución. De hecho, suele ser uno de los primeros problemas a estudiar cuando se imparten enseñanzas sobre programación dinámica. Esto es debido no solo a que el problema tiene una definición clara y simple, sino que el algoritmo de programación dinámica usado para resolver el problema también es bastante intuitivo y fácil de implementar.

No solo está relacionado el problema de la mochila con la programación dinámica por su valor didáctico a la hora de explicarla sino que el primer algoritmo creado capaz de resolver el problema de la mochila de forma exacta fue producido por la teoría de la programación dinámica de Richard Bellman en los años cincuenta [3].

En los años sesenta y setenta, aparte de estudiar más profundamente la programación dinámica como método de resolución del problema de la mochila, se empezó a experimentar e investigar utilizando algoritmos de ramificación y poda. Cuando este enfoque se fue desarrollando, probó que era el único método capaz de resolver el problema con un alto número de variables. El algoritmo más conocido de este periodo fue creado por Horowitz Y Sahni [4]

3.2 Algoritmo de las dos listas

El algoritmo a implementar, y que actuará como base para ser paralelizado es el algoritmo de las dos listas, desarrollado inicialmente por Horowitz y Sahni. Este algoritmo está considerado actualmente como el mejor algoritmo secuencial capaz de resolver el problema de la mochila de forma exacta.

El algoritmo propuesto por Horowitz y Sahni se basa en la idea principal de dividir el problema original, con n variables, en dos subproblemas, cada uno con $n/2$ variables. De está idea deriva el nombre del problema, ya que cada subproblema es una lista. Este hecho provoca que, en el peor caso sean necesarias dos listas de tamaño $2^{(n/2)}$, en lugar de necesitar una lista con 2^n elementos. Esto implica una mejora sustancial del rendimiento de aproximadamente una raíz cuadrada ($sqrt(2^n) = 2^{(n/2)}$).

El algoritmo de las dos listas de Horowitz y Sahni consta de dos partes diferenciadas: (1) la Etapa de Generación, donde se separa la entrada en dos listas y por cada lista se construyen todos los posible subconjuntos de forma ordenada, y (2) la etapa de búsqueda, donde se busca la solución al problema original utilizando una combinación de las

dos listas generadas anteriormente. A continuación se define el algoritmo de forma más detallada:

Etapa de Generación:

1. Dividir el conjunto de n ítems en dos subconjuntos disjuntos de igual tamaño: $V1 = [n_1, n_2, ..., n_{n/2}]$ y $V2 = [n_{n/2+1}, n_{n/2+2}, ..., n_n]$
2. Generar todos los subconjuntos de $V1$ (un total de $2^{n/2}$ subconjuntos) calcular su peso $a_i.w$ (suma de los pesos de todos los ítems en el conjunto a_i) y su valor $a_i.v$ (suma de los valores de todos los ítems en el conjunto a_i). Organizar está información en tripletas $(a_i, a_i.w, a_i.v)$ y ordenar las tripletas de forma no decreciente en función del peso $a_i.w$ formando la lista A
3. Generar todos los subconjuntos de $V2$ (un total de $2^{n/2}$ subconjuntos) calcular su peso $b_i.w$ (suma de los pesos de todos los ítems en el conjunto b_i) y su valor $b_i.v$ (suma de los valores de todos los ítems en el conjunto b_i). Organizar está información en tripletas $(b_i, b_i.w, b_i.v)$ y ordenar las tripletas de forma no creciente en función del peso $b_i.w$ formando la lista B

Etapa de búsqueda:

1. Calcular el máximo valor $MaxB_i$ de entre todos los subconjuntos B_i a B_n
2. Buscar una solución entre todas las combinaciones de elementos de A y B

Algoritmo 1 Fase de búsqueda

Entrada: dos listas ordenadas A y B
Salida: valor total de la solucion final *Best*

```
MaxB_n = b_n.v
for i = n − 1 to 1 do
  if b_i.v > MaxB_{i+1} then
    MaxB_i = b_i.v
  else
    MaxB_i = MaxB_{i+1}
  end if
end for
Best = 0
i = 1
j = 1
while i <= n and j <= n do
  if a_i.w + b_j.w > W then
    j++ and continue
  end if
  if a_i.v + MaxB_j > Best then
    Best = a_i.v + MaxB_j
  end if
  i++
end while
```

3.3 Generar todos los subconjuntos

La primera etapa del algoritmo de las dos listas, la etapa de generación, consiste en generar dos listas A y B con todos los posibles subconjuntos de aproximadamente $n/2$ ítems cada una. Ambas listas están ordenadas respecto al peso total de cada subconjunto, una de forma no decreciente (A) y la otra de forma no creciente (B).

El proceso para generar la lista A comienza con una lista inicial naturalmente ordenada A_1, que contiene el subconjunto vacío, y un subconjunto con solo el primer ítem ($A_1 = [0, w_1]$; teniendo solo en cuenta los pesos). Para generar la lista A_2 primero se genera la lista A_2^1 añadiendo el segundo ítem a cada elemento de la lista A_1, quedando $A_2^1 = [w_2, w_1 + w_2]$. Dado que ambas listas están ordenadas se puede utilizar un algoritmo para fusionar dos listas ordenadas para obtener la lista A_2. Repitiendo el proceso para cada ítem de $V1$ se llega a obtener la lista A. La lista de subconjuntos B se obtiene de forma análoga a esta.

Algoritmo 2 Generacion de la lista ordenada de subconjutos

Entrada: $V1$
Salida: A

$A_1 = w_1$
for $i = 2$ to $n/2$ **do**
  Generar A_i^1 añadiendo el item i a todos los elementos de A_{i-1}
  Generar A_i fusionando las dos listas ordenadas A_{i-1} y A_i^1
end for

CAPÍTULO 4
Librerías de paralelización

4.1 OpenMP

OpenMP (Open Multi-Processing) es una API (application programming interface, interfaz de programación de aplicaciones en español) diseñada para facilitar el desarrollo de aplicaciones multiproceso en entornos de memoria compartida. OpenMp es un estándar de-facto, y es uno de los modelos de programación paralela más utilizados hoy en dia.

OpenMP está definido y gestionado por el consorcio tecnológico OpenMP Architecture Review Board (o OpenMP ARB). Este está formado por un grupo de los principales vendedores de hardware (AMD, ARM, Intel, Nvidia) y software (Oracle, Red Hat). También forman parte otras organizaciones, como universidades y agencias gubernamentales.

OpenMP utiliza un modelo de programación paralela altamente escalable y portable, basado en el modelo de ejecución fork-join. OpenMp proporciona una interfaz para el desarrollo de aplicaciones paralelizadas simple y flexible, siendo aplicable desde en un ordenador de sobremesa estándar hasta en un supercomputador. Permite añadir concurrencia a los programas escritos en C, C++ y Fortran en la mayoría de plataformas, arquitecturas y sistemas operativos, incluyendo Linux, macOs y Windows. Ha sido implementado en varios compiladores comerciales como Visual Studio, GCC e Intel Fortran.

Utilizando el modelo de ejecución fork-join, OpenMP es una implementación multihilo, donde la ejecución comienza con un solo hilo (el hilo maestro o master thread), y al llegar a la parte del código a ser ejecutado en paralelo se divide en un determinado número de hilos de trabajo (slave threads), dividiendo las tareas entre ellos. Estos hilos trabajan concurrentemente, con el entorno de ejecución asignando los diferentes hilos entre los procesadores (y núcleos) disponibles. Cuando la ejecución del código paralelo acaba, los hilos generados se unen de vuelta al hilo maestro, que continúa su ejecución hasta el final del programa, o hasta la siguiente sección paralela.

Las secciones del código que van a ejecutarse en paralelo se marcan utilizando directivas de compilación, que crearán los hilos correspondientes antes de que la sección se ejecute. Cada hilo tiene un identificador numérico, que se puede obtener utilizando una función proporcionada por OpenMP. El id del hilo maestro es siempre 0.

Por defecto, cada sección de código paralelo es ejecutada por cada hilo independientemente del resto de hilos, pero OpenMP incorpora construcciones que permiten dividir una tarea entre los hilos, de forma que cada hilo ejecuta solo una parte del código. Esto permite obtener tanto paralelismo de datos como paralelismo de tareas utilizando OpenMP.

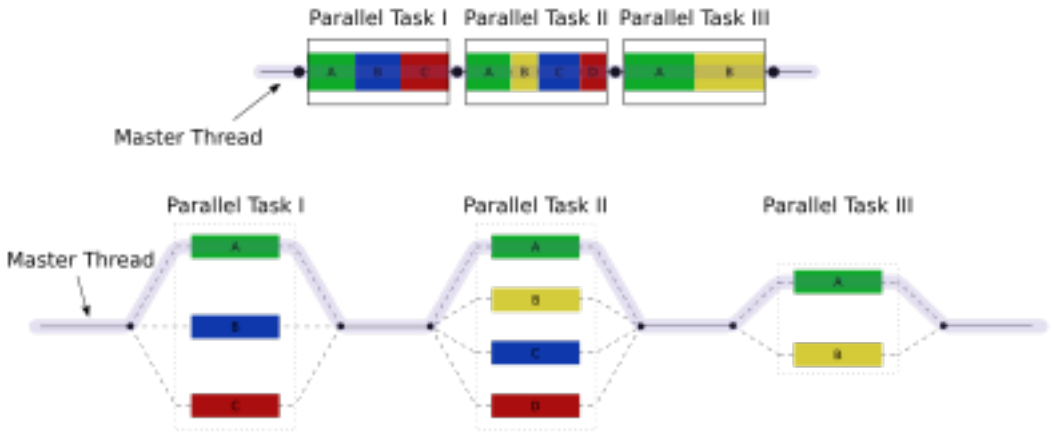

Figura 4.1: Modelo Fork-Join

OpenMP está compuesto por un grupo de directivas de compilador, funciones, y variables de entorno. Con estos elementos implementa construcciones para crear hilos, distribuir tareas, distribuir datos y mecanismos de sincronización. En C/C++ las directivas están implementadas utilizando pragmas. A continuación se muestran las principales construcciones de OpenMP:

omp_set_num_threads	Afecta al número de hilos utilizados para las subsecuentes regiones paralelas que no especifican la cláusula num_threads.
omp_get_num_threads	Devuelve el número de hilos actualmente activos.
omp_get_thread_num	Devuelve el id del hilo que llama a la función.

Tabla 4.1: Funciones de OpenMP

OMP_NUM_THREADS	Establece el número de hilos a utilizar en las secciones paralelas.
OMP_SCHEDULE	Establece el tipo de planificación y el tamaño de bloque a utilizar.

Tabla 4.2: Variable de Entorno de OpenMP

parallel	Crea un grupo de hilos y comienza la ejecución en paralelo.
Reparto de Tareas	
for	Especifica que las iteraciones del bucle asociado serán ejecutadas en paralelo por el grupo de hilos.
sections	Contiene un conjunto de bloques de código que se distribuyen entre los diferentes hilos para su ejecución.
single	Especifica que el bloque de código asociado es ejecutado por un solo hilo.
Reparto de Datos	
shared	Declara que una o más variables van a ser compartidas por los hilos, lo que significa que son visibles y accesibles por todos los hilos.
private	Declara que una o más variables van a ser privadas para cada hilo, lo que significa que cada hilo tendrá una copia local. Una variable privada no es inicializada y tampoco se mantiene fuera de la región paralela.
firstprivate	Declara que una o más variables van a ser privadas para cada hilo y las inicializa al valor existente al encontrar la directiva.
reduction	Declara una variable y un tipo de operación (suma, resta, and lógico, ...). Es una forma segura de combinar el trabajo de los hilos al finalizar la directiva.
Sincronización	
critical	Marca el siguiente bloque de código como sección crítica. Este bloque no podrá ser ejecutado por más de un hilo simultáneamente.
atomic	Marca que la actualización de la memoria de la siguiente instrucción se realizará de forma atómica.
barrier	Especifica una barrera explícita en la que cada hilo esperará a que el resto de hilos alcancen este punto.
nowait	Especifica que los hilos que completen sus tareas asignadas pueden seguir sin tener que esperar a que el resto de hilos acaben. Si está directiva no está presente, existe una barrera implícita al final de las directivas de reparto de tareas.
Planificación	
schedule(tipo, tamaño-bloque)	Especifica cómo se reparten las tareas entre los diferentes hilos: • static: Las iteraciones se dividen en bloques de tamaño tamaño-bloque y se asignan a los hilos siguiendo un reparto round-robin. • dynamic: Cada hilo ejecuta un bloque de iteraciones, y al acabar pide otro bloque, hasta que no quede ninguno por realizar • guided: Un gran número de iteraciones contiguas son asignadas a un hilo. Dicha cantidad de iteraciones decrece exponencialmente con cada nueva asignación hasta un mínimo especificado por el parámetro tamaño-bloque. • runtime: el tipo de planificación y el tamaño de bloque se especifican al ejecutar el programa

Tabla 4.3: Directivas de OpenMP

4.2 MPI

MPI ("Message Passing Interface", Interfaz de Paso de Mensajes) es una especificación de la interfaz de una librería para paso de mensajes, definiendo la sintaxis y la semántica de las funciones contenidas. MPI está diseñada para ser utilizada en programas que exploten la existencia de múltiples procesadores y nodos de cálculo interconectados. Los objetivos de MPI son un alto rendimiento, escalabilidad y portabilidad. MPI es actualmente el modelo dominante en los ambientes de cómputo de alto rendimiento. De hecho, se ha convertido en el estándar de facto para comunicaciones entre procesos que modelan un programa paralelo siendo ejecutado en un entorno de memoria distribuida. Los supercomputador de memoria distribuida actuales, como por ejemplo los clusters, suelen ejecutar ese tipo de programas.

MPI es solamente una especificación, no es una implementación. Existen diferentes implementaciones de MPI. La especificación es de una interfaz de una librería, MPI no es un lenguaje de programación. Todas las operaciones de MPI están expresadas como funciones, subrutinas, o métodos, dependiendo para qué lenguaje sea una implementación, de los cuales, C, C++ y Fortran son parte del estándar MPI.

Las primeras discusiones sobre MPI empezaron en el verano de 1991, donde se propuso un Workshop sobre estándares para paso de mensajes en entornos de memoria distribuida, que fue llevado a cabo en 1992. En este workshop se discutieron las características esenciales que debería contener una interfaz estándar de paso de mensajes, y se estableció un grupo de trabajo que continuaría el proceso de estandarización. El primer borrador del estándar MPI fue presentado en la conferencia Supercomputing ‘93. Después de un periodo de comentarios públicos, en el que se produjeron algunos cambios en el estándar, la versión 1.0 de MPI fue lanzada en Junio de 1994. El desarrollo de PMI incluyó a unas 80 personas de 40 organizaciones diferentes, principalmente de Europa y Estados Unidos. La mayoría de principales vendedores de ordenadores concurrentes se involucraron con MPI, junto con investigadores de diferentes universidades, organizaciones gubernamentales, y miembros de la industria.

La primera implementación realizada de MPI, concretamente de la versión 1.x, fue realizada por el ANL (“Argonne National Laboratory”, Laboratorio Nacional Argonne) y la universidad del estado de Mississippi y es conocida como MPICH. La mayoria de compañias de supercomputadores de principios de los 90, utilizaban MPICH, o en su defecto desarrollaban su propia implementación.

Otra de las principales implementaciones de MPI fue creada por el centro de supercomputación de Ohio (Ohio Supercomputer Center), y es conocida como LAM/MPI. Está implementación, junto contras implementaciones recientes como FT-MPI, LA-MPI y PACX-MPI, se fusionaron para crear una nueva implementación conocida como OpenMPI, que ha dia de hoy está presente en muchos supercomputadores del TOP-500 supercomputers.

Existen varias implementaciones que derivan de trabajos previos como MPICH y LAM, incluyendo implementación comerciales desarrolladas por HP, Intel y Microsoft. A pesar de que la especificación de MPI define una interfaz en C, C++ o Fortran, el lenguaje utilizado para implementar una aplicación basada en MPI no está limitado a estos, existiendo "bindings"(un binding es una adaptación de una biblioteca para ser usada en un lenguaje de programación distinto de aquel en el que ha sido escrita) para muchos otros lenguajes de programación, incluyendo Java, Perl, Python y Ruby.

MPI está basado en el modelo de programación de paso de mensajes, en el cual, se asume que existen varios procesos y los datos son movidos del espacio de direccionamiento de un proceso al espacio de direccionamiento de otro proceso, mediante el uso de

operaciones cooperativas entre ambos procesos. Adicionalmente, MPI implementa varias extensiones al modelo "clásico" de paso de mensajes como: operaciones colectivas, operaciones de acceso a memoria remota, creación dinámica de procesos, y operaciones de entrada/salida paralelas. Tanto las comunicaciones punto a punto, como las comunicaciones colectiva están soportadas en MPI. MPI soporta tanto paso de mensajes síncrono como asíncrono, dependiendo de si el proceso que envía el mensaje espera a que el mensaje sea recibido, o no. En el paso de mensajes asíncrono, el proceso que envía, no espera a que el mensaje sea recibido, y continúa su ejecución, siendo posible que vuelva a generar un nuevo mensaje y a enviarlo antes de que se haya recibido el anterior. Para este propósito, MPI utiliza un sistema de "buzones", en los que se almacenan los mensajes a espera de que un proceso los reciba. En el paso de mensajes síncrono, el proceso que envía el mensaje espera a que un proceso lo reciba para continuar su ejecución.

Uno de los conceptos principales de MPI son los comunicadores. Un comunicador es un objeto que conecta grupos de procesos en una sesión de MPI, de forma que estos procesos son capaces de enviarse mensajes entre sí. A cada proceso dentro de un comunicador se le asigna un identificador independiente conocido como rank. El comunicador ordena los procesos que contienen en una topología ordenada según el rank de cada proceso. Los comunicadores pueden ser creados y particionados utilizando varios comandos de MPI. En MPI existe un comunicador básico, denominado MPI_COMM_WORLD, que agrupa a todos los procesos activos durante la ejecución de una aplicación. la variable rank permite determinar que proceso ejecuta determinada porción de código, por lo que permite controlar y alterar la ejecución del programa.

Las funciones proporcionadas por MPI incluyen, pero no están limitadas a, comunicaciones punto a punto con operaciones de envío/recepción, chambiar la topologia logica de los procesos entre tipo grafo y cartesiana, intercambio de datos entre pares de procesos, difusión de datos entre procesos, combinar resultados parciales (operaciones de recogida y reducción), sincronizar nodos, además de obtener información sobre la red de procesos como el número total de procesos, la identidad del proceso, etc. Estas funciones pueden agruparse en cuatro categorías:

Inicializar, administrar y finalizar comunicaciones.	Permiten gestionar la inicialización y finalización de la biblioteca de paso de mensajes, identificar el número de procesos (size) y el rango de los procesos (rank), y crear, modificar y borrar los comunicadores existentes.
Comunicaciones punto a punto	Incluye operaciones de comunicación entre dos procesos, es decir, comunicaciones punto a punto, para diferentes tipos de actividades de envío y recepción, tanto bloquean como no bloqueantes, además, incluye el mecanismo llamado "ready-send" mediante el cual una peticion de envio solo puede realizarse si previamente se ha realizado una petición para recibir el mensaje.

Comunicaciones colectivas	Proveen operaciones de comunicaciones entre todos los procesos de un comunicador (Que pueden ser todos los procesos existentes o un subconjunto de estos definido por el programa). Incluyen operaciones tipo difusión (broadcast), recolección (gather), distribución (scatter) y reducción.
Tipo de datos derivados	Permiten crear y gestionar tipos de datos complejos (por ejemplo struct en c) de forma que estos tipos de datos se puedan utilizar en la comunicaciones de MPI, ya que muchas funciones de MPI requieren que se especifique el tipo de los datos a enviar. Con este objetivo incorpora wrappers a los tipos de datos primitivos, para que puedan ser utilizados por MPI (MPI_INT para variables del tipo int en c).

Tabla 4.4: Funciones de MPI

Las funciones de paso de mensajes más básicas de MPI son MPI_Send y MPI_Recv. La primera función envía un mensaje al proceso designado mientras que la segunda se utiliza para recibir un mensaje de un proceso. Para que sea posible transmitir correctamente un mensaje en el sistema, MPI de añadir cierta información adicional a los datos que el programa desea transmitir. Está información adicional forma la cabecera del mensaje. Está cabecera consciente la siguiente información:

- El rank del receptor
- El rank del emisor
- Una etiqueta
- Un comunicador

Estos valores pueden ser utilizados por el receptor para diferenciar los mensajes recibidos. La etiqueta es un valor entero especificado por el usuario, que puede utilizarse para distinguir entre mensajes recibidos de un mismo proceso. Por ejemplo, suponiendo que el proceso A le mande dos mensajes al proceso B, ambos mensajes conteniendo un único valor en coma flotante. Uno de los valores tiene que ser usado en un cálculo, mientras que el otro ha de ser mostrado. Para poder diferenciarlos y determinar cual es cual, A puede utilizar diferentes etiquetas para cada uno de ellos. Si B utiliza las mismas etiquetas a la hora de recibir, cuando reciba los mensajes "sabrá" que hacer con cada cual.

CAPÍTULO 5
Implementación Paralela

5.1 Distribución de trabajo

La mayoría de sistemas de alto rendimiento actuales consisten en en un gran número de núcleos agrupados en nodos con una arquitectura de memoria compartida, y varios de estos nodos interconectados entre sí, formando un cluster de altas prestaciones. Esto provoca que existan buenas razones para adoptar un modelo de programación híbrido MPI/OpenMP en aplicaciones para ese tipo de arquitecturas, pero esto añade complejidad al programa paralelo.

MPI es una especificación de una librería de paso de mensajes diseñada para el cómputo de altas prestaciones en sistemas de memoria distribuida. Con MPI, los datos se mueven de un proceso a otro mediante operaciones cooperativas. Cada proceso MPI tiene su propio espacio de direccionamiento. MPI incluye tanto comunicación como sincronización, y se ha convertido en el estándar de facto para procesos corriendo en sistemas de memoria distribuida.

OpenMP es una especificación para un conjunto de pragmas, librerías y variables de entorno utilizadas para aplicaciones multihilo con memoria la compartida entre todos los hilos. OpenMP es ampliamente utilizado para procesos de memoria compartida que corren en un procesador multinúcleo.

El modelo de programación híbrido MPI/OpenMP es ampliamente utilizado debido a su potencial. Este modelo híbrido aproveche ambas técnicas de computación paralela: utiliza un modelo de comunicaciones bien definido para comunicaciones entre procesos corriendo en diferentes nodos heterogéneos, y utiliza grupos de hilos corriendo en cada nodo para poder aprovechar completamente las arquitecturas multiprocesador / multinúcleo. El modelo híbrido MPI/OpenMP utiliza MPI para realizar las comunicaciones entre nodos y la programación con memoria compartida de OpenMP en cada nodo.

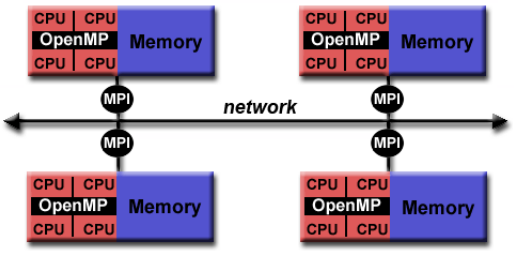

Figura 5.1: Modelo Híbrido MPI/OpenMP

$$A = (A_1, A_2, ..., A_p) \quad A_i = (a_i^1, a_i^2, ..., a_i^k) \quad 1 <= j <= p$$

$$B = (B_1, B_2, ..., B_p) \quad B_j = (b_j^1, b_j^2, ..., b_j^k) \quad 1 <= j <= p$$

Figura 5.2: Division de A y B en bloques

El modelo MPI/OpenMP ofrece ciertas ventajas respecto a utilizar solamente un modelo MPI puro. En un modelo de programación paralela MPI, si la cantidad de procesos MPI va creciendo en cada nodo, la memoria consumida por los procesos eventualmente superará la memoria disponible en el nodo, provocando una pérdida de rendimiento. Además, la escalabilidad de MPI es limitada debido al límite de ancho de banda de los dispositivos de transmisión internodo.

En el modelo híbrido que combina la programación MPI con la programación OpenMP, la memoria de cada nodo es compartida por los hilos de OpenMP, por lo que la cantidad de memoria consumida es bastante menor que la que consumiría el mismo número de procesos MPI. Adicionalmente, la escalabilidad es mayor dado que hay menos comunicaciones MPI entre nodos, y las comunicaciones intranodo entre los hilos de OpenMP es mucho mas rapida que las comunicaciones MPI. El número de hilos OpenMP por cada proceso MPI (idealmente un proceso MPI por nodo) tienen que ser escogidos cuidadosamente, y el mecanismo de bloqueo MPI aplicado correctamente, ya que en este modelo híbrido pueden aparecer interbloqueos.

5.2 Algoritmo de las dos listas paralelo

El algoritmo paralelo a implementar, utilizando el modelo de programación híbrida MPI/OpenMP propuesto es una variación del algoritmo de las dos listas original implementado secuencialmente.

La idea principal del algoritmo no varía respecto a la versión secuencial, dividir el problema original, con n variables, en dos subproblemas, cada uno con $n/2$ variables. Una vez separada la entrada en dos listas, por cada una de ellas se construyen en paralelo todos los posibles subconjuntos de forma ordenada. Después se pasa a la etapa de búsqueda, donde se busca en paralelo la solución al problema original utilizando una combinación de las dos listas generadas anteriormente.

Para generar las dos listas A y B de subconjuntos es crucial utilizar un algoritmo adecuado que permita un correcto reparto del esfuerzo de cómputo entre los nodos / hilos disponibles, y que minimice las sincronizaciones y comunicaciones intranodo necesarias.

La principal diferencia de la versión paralela del algoritmo respecto a la versión secuencial radica en la fase de búsqueda. La fase de búsqueda del algoritmo se ha dividido en dos subfases: una fase previa de poda, y la propia fase de búsqueda. Para estas dos fases las dos listas A y B se dividen en p bloques, de similar tamaño k, siendo p el número de hilos disponibles (número de hilos por nodo * número de nodos) En este punto se supone que la estructura de A y B es, respectivamente:

El objetivo de la fase de poda es reducir el tiempo de la fase de búsqueda en cada hilo reduciendo el espacio de búsqueda. Para ello se introducen los siguientes dos lemas:

Lema 1: por cada pareja de bloques $(A_i, B_j), 1 <= i, j <= p$, si $a_i^1 + b_j^k > M$, entonces cualquier par de elementos (a_i^r, b_j^s), donde $a_i^r \in A_i$, $b_j^s \in B_j$, no es una solución al problema de la mochila.

Lema 2: por cada pareja de bloques $(A_i, B_j), 1 <= i, j <= p$, si $a_i^k + b_j^1 < M$, entonces cualquier par de elementos (a_i^r, b_j^s), donde $a_i^r \in A_i$, $b_j^s \in B_j$, no es una solución al problema de la mochila.

Algoritmo 3 Fase de poda paralela

Entrada: dos listas ordenadas A y B
Salida: pares de bloques escogidos

```
Dividir A y B en p bloques de tamaño k
for i = 1 to p do
  parallel
  for j = i to p + i − 1 do
    X = a_i^1 + b_{jmodp}^k
    Y = a_i^k + b_{jmodp}^1
    if X = M or Y = M then
      stop Una solucion encontrada
    else if X <M or Y >M then
      Guardar (A_i, B_{jmodp})
    end if
  end for
end for
Combinar las listas parciales de pares de bloques
```

La fase de búsqueda del algoritmo reparte los bloques que son candidatos a contener la solución óptima al problema entre los hilos, cada hilo busca la solución en sus bloques asignados de forma similar al algoritmo secuencial, y se reducen todas las soluciones encontradas por los hilos, quedándose solamente con la mejor, que es la solución final.

La definición completa del algoritmo de las dos listas paralelo quedaría de la siguiente forma:

Etapa de Generación:

1. Dividir el conjunto de n ítems en dos subconjuntos disjuntos de igual tamaño: $V1 = [n1, n2, ..., n_{n/2}]$ y $V2 = [n_{n/2+1}, n_{n/2+2}, ..., n_n]$

2. Generar en paralelo y de forma ordenada todos los subconjuntos de $V1$ (un total de $2^{n/2}$ subconjuntos) calcular su peso $a_i.w$ (suma de los pesos de todos los ítems en el conjunto a_i) y su valor $a_i.v$ (suma de los valores de todos los ítems en el conjunto a_i). Organizar está información en tripletas $(a_i, a_i.w, a_i.v)$ formando la lista A

3. Generar en paralelo y de forma ordenada todos los subconjuntos de $V2$ (un total de $2^{n/2}$ subconjuntos) calcular su peso $b_i.w$ (suma de los pesos de todos los ítems en el conjunto b_i) y su valor $b_i.v$ (suma de los valores de todos los ítems en el conjunto b_i). Organizar está información en tripletas $(b_i, b_i.w, b_i.v)$ formando la lista B

Etapa de poda:

1. Dividir las listas de subconjuntos A y B en p bloques de tamaño k

2. Asignar el bloque A_i al hilo i

3. Cada hilo ejecuta el algoritmo de poda propuesto

4. Se combinan las listas de pares de bloques de cada hilo obtenidas en el paso 3 para obtener una lista con todos los pares de bloques válidos.

Etapa de búsqueda:

1. Se reparten los pares de bloques válidos entre los hilos
2. Por cada par de bloques calcular el máximo valor $MaxB_i$ de entre todos los subconjuntos b_j^1 a b_j^k
3. Buscar una solución entre todas las combinaciones de elementos de A_i y B_j
4. Combinar los resultados obtenidos por todos los hilos para hallar la solución óptima

CAPÍTULO 6
Evaluación Experimental

6.1 Configuración de las pruebas

Las pruebas se han realizado en el cluster Kahan, perteneciente a la Universitat Politècnica de València. El cluster está compuesto de 6 nodos, con dos procesadores cada uno, interconectados por una red de altas prestaciones InfiniBand. Cada nodo consta de:

1. 2 procesadores AMD Opteron 16 Core 6272, 2.1GHz, 16MB
2. 32GB de memoria DDR3 1600
3. Disco 500GB, SATA 6 GB/s
4. Controladora InfiniBand QDR 4X (40Gbps, tasa efectiva de 32Gbps)

En las pruebas realizadas que utilizan múltiples nodos, cada nodo se ha configurado un proceso MPI. La implementación de MPI utilizada es Open-MPI 1.6.2, configurado con soporte para InfiniBand y para el sistema de colas. Como compilador se ha utilizado GNU GCC 4.4.6, con soporte para OpenMP 3.0. Adicionalmente, el sistema operativo utilizado por los nodos del cluster es CentOS 6.3.

Para evaluar el rendimiento de la implementación paralela respecto a la versión secuencial, ambas versiones han sido desarrolladas y probadas bajo las mismas condiciones. De la versión paralela del algoritmo se han realizado dos implementaciones, la primera utilizando solamente OpenMP. Está primera versión paralela se ha utilizado como base para desarrollar la versión paralela final, siguiendo el modelo de programación paralela híbrido MPI/OpenMP.

Tanto el algoritmo de las dos listas original propuesto por Horowitz y Sahni como la versión paralela tienen una complejidad espacial de $O(2^{n/2})$, es decir, la cantidad de memoria requerida por el algoritmo crece exponencialmente respecto al tamaño de la

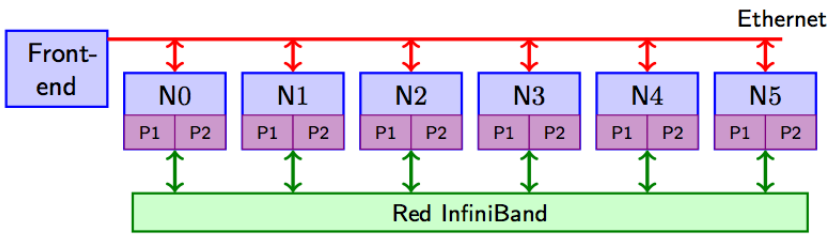

Figura 6.1: Arquitectura del clúster Kahan

entrada (número de ítems en este caso). Por ello, el tamaño de problema a probar está limitado. Teniendo esto en cuenta, se han probado siete tamaños de problema diferentes, variando de 48 a 60 en incrementos de 2.

Para cada tamaño de problema se ha utilizado un generador aleatorio de instancias para el problema de la mochila, desarrollado expresamente para este trabajo, para generar 10 instancias del problema. Para cada instancia, se ha ejecutado y calculado su tiempo de ejecución medido en milisegundos. Para obtener el tiempo de ejecución para un tamaño de problema se ha calculado la media aritmética del tiempo de ejecución de todas las instancias de ese tamaño. El tiempo de ejecución contempla tanto el tiempo de cómputo, como los sobrecostes de inicialización, comunicaciones, sincronización y gestión de memoria.

6.2 Evaluación en un unico nodo

Para evaluar el rendimiento en un solo nodo, es decir en un entorno de memoria compartida, se han realizado una serie de experimentos comparando la implementación secuencial del algoritmo de las dos listas con la versión paralela del mismo, desarrollada utilizando la librería OpenMP.

n	Secuencial	OpenMP	Speedup
48	1467.10	1302.05	1.13
50	2942.96	2590.85	1.14
52	6463.37	5409.50	1.19
54	13505.31	11017.83	1.23
56	27330.62	22238.08	1.23
58	55724.70	44935.14	1.24
60	113626.31	90585.35	1.25

Tabla 6.1: Tiempos de ejecucion en un unico nodo

La tabla muestra la comparación del rendimiento entre la versión secuencial del algoritmo y la implementación paralela con OpenMP. Como se puede observar, con un speedup medio de 1.20, la versión paralela tiene un mejor rendimiento que la versión secuencial.

Un detalle que es importante destacar, es el hecho de que conforme aumenta la talla del problema, también mejora el speedup. Dado que el uso de librerías de paralelización suele llevar un sobrecoste asociado, al tener que manejar los diferentes hilos y tener en cuenta factores como la sincronización y el reparto de datos, y este sobrecoste está contemplado en el tiempo de ejecución, al incrementar la talla del problema, la importancia de estos sobrecostes respecto al tiempo de ejecución total se reduce, aumentando la eficiencia.

Para poder analizar mejor el rendimiento de la implementación paralela se han medido el tiempo de ejecución de las dos fases del algoritmo independientemente. Para poder comparar los tiempos, en la medición de los tiempos del algoritmo paralelo se han juntado las fases de poda y búsqueda, dado que la fase de poda no aparece en el algoritmo original pero deriva de su fase de búsqueda.

La tabla muestra la comparación de los tiempos de ejecución de las dos fases del algoritmo, junto con los tiempos de ejecución totales. Se puede observar que la fase de generación ocupa una porción del tiempo de ejecución mucho mayor que la fase de búsqueda. Hay que destacar que la introducción de la fase de poda dentro de la fase de búsqueda de

	Secuencial			OpenMP		
n	Total	Generación	Búsqueda	Total	Generación	Búsqueda
48	1451.38	1074.26	374.99	1298.64	985.90	311.30
50	2932.31	2165.49	763.70	2561.90	1966.30	593.22
52	6380.19	4871.99	1502.94	5384.59	4169.68	1210.81
54	13359.58	10329.52	3020.44	10959.08	8570.37	2380.57
56	27212.99	21252.36	5942.54	22041.90	17476.21	4549.91
58	55640.81	43696.85	11906.48	44466.95	35187.99	9247.17
60	113580.57	89280.25	24214.99	90295.96	71647.62	18579.98

Tabla 6.2: Tiempos de ejecucion por fases

la implementación paralela proporciona buenos resultados, dado que el speedup de está, un 1.27 de media, es ligeramente superior al de la fase de generación, con un speedup medio de 1.18. Pero, teniendo en cuenta que la fase de búsqueda tiene una contribución mucho menor al tiempo de ejecución total, la fase de búsqueda en la implementación paralela provoca una reducción del tiempo de ejecución mayor.

6.3 Evaluación en múltiples nodos

En está sección se va a comparar el rendimiento de la implementación paralela con un modelo híbrido MPI/OpenMP siendo ejecutada en varios nodos respecto al algoritmo secuencial original.

		2 Nodos		4 Nodos		6 Nodos	
n	Secuencial	Tiempo	Speedup	Tiempo	Speedup	Tiempo	Speedup
48	1467.10	1060.90	1.38	1021.59	1.44	1067.23	1.37
50	2942.96	2111.70	1.39	1997.41	1.47	2194.29	1.34
52	6463.37	4333.79	1.49	3936.58	1.64	4019.82	1.61
54	13505.31	8687.58	1.55	7722.48	1.75	7553.10	1.79
56	27330.62	17500.89	1.56	15237.89	1.79	14510.10	1.88
58	55724.70	35199.74	1.58	30276.82	1.84	27466.11	2.03
60	113626.31	71335.30	1.59	60349.67	1.88	55317.67	2.05

Tabla 6.3: Tiempos de ejecucion en multiples nodos

La tabla muestra los tiempos de ejecución y el speedup respecto a la versión secuencial de la implementación paralela híbrida, corriendo en un número variable de nodos y variando el tamaño de la entrada al problema. Se puede observar que el rendimiento de la versión híbrida es significativamente superior al de la versión secuencial, y, con un speedup medio de 1.51, 1.69, 1.73 al ser ejecutada en 2, 4, y 6 nodos respectivamente, mejora el rendimiento de la versión paralelizada solamente con OpenMP.

La tabla anterior muestra el rendimiento de la versión híbrida corriendo en un solo nodo respecto a la implementación pura con OpenMP. Los tiempos de ejecución son muy similares, siendo ligeramente superiores en la versión híbrida. Estos son los resultados esperados, ya que al ser ejecutado sobre un solo nodo la parte de MPI no proporciona ningún tipo de poder de cómputo adicional respecto a la versión pura, pero sí que tiene unos sobrecostes asociados, que provocan el pequeño aumento de tiempo de ejecución.

De los resultados obtenidos con la version hibrida MPI/OpenMP se puede deducir que, al igual que la versión pura de OpenMP, los sobrecostes de utilizar las librerías paralelas tienen un claro impacto en el tiempo de ejecución total, y conforme se aumenta la

		OpenMP		MPI 1 Nodo	
n	Secuencial	Tiempo	Speedup	Tiempo	Speedup
48	1467.10	1302.05	1.13	1324.28	1.11
50	2942.96	2590.85	1.14	2643.88	1.11
52	6463.37	5409.50	1.19	5548.53	1.16
54	13505.31	11017.83	1.23	11222.33	1.20
56	27330.62	22238.08	1.23	22802.61	1.20
58	55724.70	44935.14	1.24	45894.05	1.21
60	113626.31	90585.35	1.25	92597.56	1.23

Tabla 6.4: Comparacon de tiempos de ejecucion en un unico nodo

talla del problema, la importancia de estos sobrecostes respecto al coste total se diluye, mejorando así la eficiencia.

6.4 Coste de comunicaciones

La versión paralela del algoritmo de las dos listas implementada utilizando un modelo de programación paralela híbrido MPI/OpenMP realiza un uso intensivo del sistema de comunicaciones de MPI para intercambiar datos de un nodo a otro en todas las fases del algoritmo, no solo durante la inicialización o para la recogida de resultados final.

En [8] Wan et al. proponen un esquema de distribución de tareas diseñado para evitar en la manera posible las comunicaciones. En el artículo presenta una comparativa entre la versión original del algoritmo, OHCI (original heterogeneous cooperative implementation) y la implementación mejorada, IHCI (improved heterogeneous cooperative implementation)

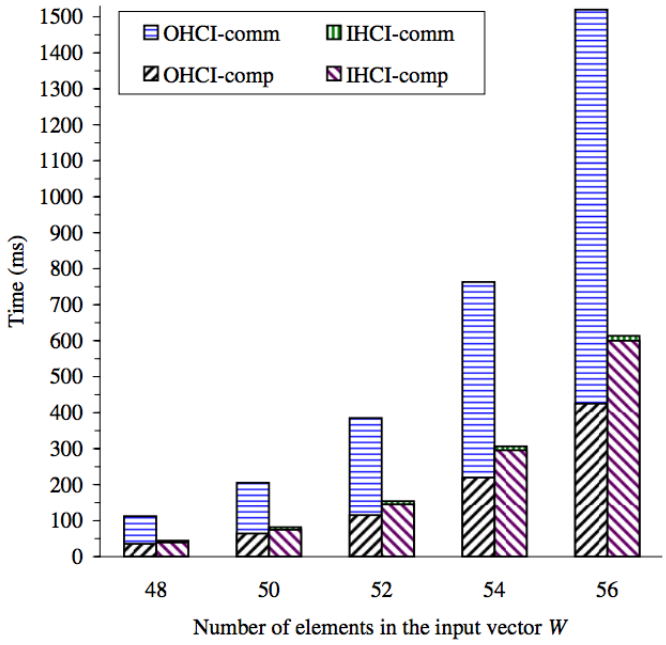

Figura 6.2: Comparación del coste de comuncaciones

Wan, L. et al. (2015) A novel cooperative accelerated parallel two-list algorithm for solving the subset-sum problem on a heterogeneous cluster

La figura muestra una comparación de tiempos de ejecución entre OHCI e IHCI, separando el tiempo de comunicaciones con el de cómputo, para diferentes tamaños de problema. Se puede observar muy fácilmente el impacto que tienen las comunicaciones en el algoritmo original, necesitando más tiempo de ejecución que la parte de computo.

Dada la importancia que tienen, para reducir todo lo posible el sobrecoste de las comunicaciones se han utilizado las operaciones colectivas proporcionadas por MPI cuando ha sido necesario. Este tipo de operaciones son muy convenientes ya que reducen el coste de las comunicaciones respecto a realizarlo mediante operaciones de comunicación punto a punto.

CAPÍTULO 7
Conclusiones

En este trabajo de final de grado se ha realizado un estudio en profundidad del problema de la mochila, analizando su historia, principales características y variaciones, y se ha investigado el algoritmo de las dos listas, uno de los principales algoritmos que existen para resolver el problema, extendiendolo con un modelo de programación paralela híbrida MPI/OpenMP. Se han implementado varias versiones del algoritmo, una secuencial, un versión paralela para entornos de memoria compartida y la versión final con un modelo híbrido MPI/OpenMP, y el rendimiento de las mismas ha sido analizado, siendo probado en el cluster de 6 nodos multinúcleo Kahan, propiedad de la UPV.

En los resultados del análisis se ha observado que las versiones paralelas tienen un mejor rendimiento que la versión secuencial, por lo que se ha cumplido el objetivo principal de realizar una versión paralela de un problema o algoritmo, reducir el tiempo de ejecución. Con los resultados obtenidos también ha sido posible constatar la escalabilidad horizontal de la versión paralela híbrida MPI/OpenMP, ya que al incrementar el número de nodos con los que se realizan los cálculos, se aumentaba la eficiencia, y por lo tanto se reducía el tiempo de ejecución. Los resultados obtenidos permiten llegar a la conclusión de que se podría reducir aún más el tiempo de cómputo si se dispusiera de más nodos en el cluster.

7.1 Trabajos Futuros

Las versiones paralelas del algoritmo de las dos listas implementadas en este trabajo distan de ser la mejor implementación posible. Los dos principales puntos de mejora de las implementaciones son los siguientes:

1. Adaptar el esquema de distribución de tareas propuesto en [8]. Este esquema ha sido diseñado para tratar de reducir al mínimo posible las comunicaciones necesarias en el transcurso de la ejecución del algoritmo, dado que la versión original del mismo hace un uso intensivo de estas. En el artículo comparan ambos esquemas de trabajo, llegando a la conclusión de que el rendimiento del esquema que proponen es significativamente superior a la versión tradicional. Dado una de las implementaciones realizadas no utiliza comunicaciones al ser en un entorno de memoria compartida, está mejora solo se podría aplicar a la versión paralela híbrida MPI/OpenMP.
2. Implementar el algoritmo de fusión de listas paralelo óptimo. En el algoritmo de las dos listas, tanto en la versión secuencial como paralela, a la hora de generar las listas de subconjuntos A y B su utiliza una subrutina para fusionar dos listas ordenadas. Dado que el tamaño de las listas a fusionar es de orden exponencial ($O(2^{n/2})$

respecto a la entrada, es crucial que el algoritmo de fusión sea lo más eficiente posible. Dado que la implementación paralela del algoritmo de fusión no es trivial, actualmente está implementado el algoritmo secuencial tradicional, por lo que de sustituir este por el algoritmo paralelo óptimo propuesto en [11], se podría reducir drásticamente el tiempo de cómputo.

Bibliografía

[1] G.B. Mathews. On the partition of numbers. *Proceedings of the London Mathematical Society*, 28:486–490, 1897.

[2] Dantzig, Tobias. *Numbers: The Language of Science*. 1930.

[3] Bellman, Richard. *The theory of dynamic programming*. 1954.

[4] E. Horowitz, S. Sahni. Computing partitions with applications to the knapsack problem. *Journal of the ACM (JACM)*, 21:2:277–292, 1974.

[5] S. Martello, P. Toth. *Knapsack problems: algorithms and computer implementations*. John Wiley and Sons, Inc., 1990.

[6] D.-C. Lou, C.-C. Chang. A parallel two-list algorithm for the knapsack problem. *Parallel Computing*, 22:14:1985–1996, 1997.

[7] K. Li, Q. Li, W. Hui, S. Jiang. Optimal parallel algorithm for the knapsack problem without memory conflicts. *Journal of Computer Science and Technology*, 19:6:760–768, noviembre, 2004.

[8] L. Wan, K. Li, K. Li. A novel cooperative accelerated parallel two-list algorithm for solving the subset-sum problem on a heterogeneous cluster. *Journal of Parallel and Distributed Computing*, 97:C:112–123, noviembre 2016

[9] S. K. Ragila, S. Vodela. Cost Optimal Parallel Algorithm for 0-1 Knapsack Problem. Consultado en https://www.cs.rit.edu/~ark/fall2015/654/team/18/presentation1.pdf.

[10] Garey, M. R. and Johnson, D. S. *Computers and Intractability: a Guide to the Theory of NP-Completeness*. W.H. Freeman, New York, 1979.

[11] S. G. Akl, N. Santoro. Optimal parallel merging and sorting without memory conflicts. *Computers, IEEE Transactions on*, 100:11:1367–1369, 1987.

[12] L. Q. Nguyen. Hybrid MPI and OpenMP* Model, 28 de Junio de 2014. Consultado en https://software.intel.com/en-us/blogs/2014/06/28/hybrid-mpi-and-openmp-model.

[13] OpenMP Application Program Interface Version 4.0, Julio de 2013. Consultat a http://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf.

[14] OpenMP 4.0 API C/C++ Syntax Quick Reference Card, 2013. Consultat a http://www.openmp.org/wp-content/uploads/OpenMP-4.0-C.pdf.

[15] MPI: A Message-Passing Interface Standard Version 3.1, Junio de 2015. Consultat a http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf.

[16] P. S. Pacheco. A user's guide to MPI, Marzo de 1998. Consultado en http://moss.csc.ncsu.edu/~mueller/cluster/mpi.guide.pdf.