Curso 2003/04
CIENCIAS Y TECNOLOGÍAS/15
I.S.B.N.: 84-7756-614-3

MARÍA ISABEL DORTA GONZÁLEZ

Esqueletos paralelos para la técnica de ramificación y acotación

Directora
COROMOTO LEÓN HERNÁNDEZ

SERVICIO DE PUBLICACIONES
UNIVERSIDAD DE LA LAGUNA

SOPORTES AUDIOVISUALES E INFORMÁTICOS
Serie Tesis Doctorales

Contents

List of Figures

List of Tables

Resumen (en Español)

Capítulo 1: Introducción

La *Optimización Combinatoria* es una de las ramas de la Programación Matemática que más ha progresado en las últimas dos décadas. Tiene un amplio rango de aplicaciones en las Ingenierías (Electrónica, Civil, Mecánica y Sistemas), así como en Economía, Ciencias Sociales, Biología, Telecomunicaciones, etc. Además está estrechamente relacionada con otras disciplinas tales como la Investigación Operativa y las Ciencias de la Computación. Los avances en optimización combinatoria no se deben únicamente a su progreso teórico sino también al continuo desarrollo e innovación en la Tecnología Informática, con lo que cada día es posible resolver problemas de mayor tamaño y de forma más eficiente. En esta memoria se aborda la resolución de algunos problemas de optimización combinatoria desde el punto de vista de la Informática.

Muchos de los problemas de optimización combinatoria pertenecen a la clase de los problemas NP-duros [38]. Siguiendo la notación introducida por Ibaraki [46], se define un problema de optimización combinatoria como una t-upla $\Pi = (I, S, f, g)$ donde:

- I es el conjunto de instancias de Π. Si $x \in I$ se dice que x es una *instancia* (o una entrada) de Π.

- Dada una instancia $x \in I$, $S(x)$ denota el *conjunto de soluciones factibles* de x.

- Dada una instancia $x \in I$ y una solución factible $\sigma \in S(x)$, $f(x, \sigma)$ representa el costo de σ con respecto a Π y x. La función f se denomina *función objetivo*.

- $g \in \{\max, \min\}$. El objetivo de Π es encontrar una solución factible que optimice f en función de g: dado un valor $x \in I$, se trata de determinar una *solución óptima* $\sigma' \in S(x)$ tal que $f(x, \sigma') = g\{f(x, \sigma) \mid \sigma \in S(x)\}$.

La solución exacta de la mayoría de los problemas de optimización combinatoria implica la enumeración de los elementos de un espacio de soluciones exponencial. Fijado un problema, la búsqueda exhaustiva de una solución se puede acelerar a través de técnicas exactas como Divide y Vencerás, Ramificación y Acotación o Programación Dinámica. En nuestro caso estamos interesados en la resolución de problemas utilizando la técnica de Ramificación y Acotación. Para ello se introduce una modificación en la definición anterior añadiendo el concepto de *subproblema*.

Un *subproblema* es una t-upla $\Pi_i = (I, S_i, f, g)$, donde $S_i(x)$ es un subconjunto del espacio subyacente I.

Dado un subproblema Π_i se puede *descomponer* en $\Pi_{i1}, \Pi_{i2}, ..., \Pi_{ik}$ mediante una operación de *ramificación*, donde $S_i = \bigcup_{j=1}^{k} S_{ij}$. Así cualquier solución factible $\sigma \in S_i$ pertenece a algún S_{ij}, y viceversa, cualquier $\sigma \in S_{ij}$ pertenece a S_i.

Sea $\mathcal{Q}$ el conjunto de subproblemas generados. Un subproblema $\Pi_i \in \mathcal{Q}$ se dice *vivo* si no ha sido ramificado o estudiado aún. El conjunto de subproblemas vivos se denotará por $\mathcal{L}$. Para cada subproblema estudiado de $\mathcal{Q}$, se calculan la *cota inferior* y la *cota superior*. La mayor cota inferior obtenida hasta el momento se denomina *valor de la mejor solución* y se denota bs. La solución obtenida para bs se denomina *mejor solución* y se almacena en $\mathcal{T}$. La elección del siguiente subproblema vivo a estudiar se realiza mediante la *función de búsqueda* s, tal que $s(\mathcal{L}) = \mathcal{L}$.

Una vez introducida la notación, la resolución de un problema de optimización combinatoria mediante la técnica de Ramificación y Acotación se puede abordar siguiendo el algoritmo:

Paso 1 (inicialización):

$\mathcal{L} := \{\Pi_0\}$ (problema inicial),

$\mathcal{Q} := \{\Pi_0\}$,

$bs := -\infty$

$\mathcal{T} := \emptyset$ (conjunto vacío).

Paso 2 (búsqueda):

Si $\mathcal{L} = \emptyset$ ir al Paso 9. En otro caso, seleccionar un $\Pi_i := s(\mathcal{L})$ e ir al Paso 3.

Paso 3 (actualización):

Si $lower\_bound(\Pi_i) > bs$, hacer

$bs := lower\_bound(\Pi_i)$ y

$\mathcal{T} := \{\sigma\}$,

siendo σ es una solución factible de Π_i y $f(x, \sigma) = lower\_bound(\Pi_i)$.

Ir al Paso 4.

Paso 4: Si $\Pi_i \in G$ (el conjunto de los problemas parciales Π_i resueltos durante el cálculo de $upper\_bound(\Pi_i)$) ir al Paso 8. En otro caso ir al Paso 5.

Paso 5 (test de la cota superior):

Si $upper\_bound(\Pi_i) \leq bs$ ir al Paso 8. En otro caso ir al Paso 6.

Paso 6: Si existe un $\Pi_k (\neq \Pi_i) \in \mathcal{Q}$ tal que $f(\Pi_k) \geq f(\Pi_i)$ ir al Paso 8. En otro caso ir al Paso 7.

Paso 7 (ramificación):

Descomponer Π_i en $\Pi_{i1}, \Pi_{i2}, ..., \Pi_{ik}$.

Hacer $\mathcal{L} := \mathcal{L} \cup \{\Pi_{i1}, \Pi_{i2}, ..., \Pi_{ik}\}$ - $\{\Pi_i\}$ y

$\mathcal{Q} := \mathcal{Q} \cup \{\Pi_{i1}, \Pi_{i2}, ..., \Pi_{ik}\}$.

Volver al Paso 2.

Paso 8 (finalizar Π_i):

Hacer $\mathcal{L} := \mathcal{L}$ - Π_i. Volver al Paso 2.

Paso 9 (finalización):

Parar. La mejor solución bs es $f(\Pi_0)$. $\mathcal{T}$ almacena una solución óptima de Π_0 si $bs > -\infty$. En el caso de $bs = -\infty$, Π_0 es no factible.

Los algoritmos de Ramificación y Acotación siempre converge en un número finito de pasos. Sin embargo, la eficiencia y el espacio requerido para el almacenamiento dependen de la función de búsqueda s. Las funciones de búsqueda utilizadas en la práctica son: la *búsqueda en profundidad*, la *búsqueda en anchura* y la *búsqueda primero-el-mejor*. En términos generales, se puede decir que las búsquedas en profundidad exploran los nodos en orden inverso al de su creación, empleando una pila para almacenar los nodos que se van generando pero no han sido examinados aún. Por otro lado, las búsquedas en anchura exploran los nodos en el mismo orden en que son creados, utilizando una cola para almacenar los nodos que se han generado pero no han sido examinados aún. Las búsqueda primero-el-mejor exploran los nodos de acuerdo con los valores de las mejores cotas.

El esquema de trabajo tanto de la técnica de ramificación y acotación como del resto de técnicas exactas, se puede generalizar para desarrollar herramientas software que permitan el análisis, diseño e implementación de resolutores para problemas concretos, como es el caso de los *esqueletos*. Desafortunadamente, en muchas aplicaciones aparecen casos de tamaño irresoluble, esto es, no es posible encontrar la solución óptima en tiempo razonable. Sin embargo, es posible incrementar el tamaño de los problemas que se pueden abordar por medio de técnicas de paralelización. Ocultando el paralelismo en el interior de librerías de esqueletos algorítmicos, se puede facilitar el proceso de programación de ciertas aplicaciones [14]. Con ello se reduce la participación del usuario a la definición de los elementos que determinan su problema y a la instanciación de los procedimientos que parametrizan el esqueleto elegido. En nuestra propuesta, implementamos un esqueleto basado en el paralelismo de granja, también denominado *maestro-esclavo*. Esta implementación paralela para memoria distribuida fue desarrollada haciendo uso de la librería MPI [88] (la herramienta estándar en programación usando paso de mensaje). Lo que nos ha permitido adaptar la herramienta a diferentes plataformas hardware. Otra de las herramientas de paralelización utilizada para la paralelización del esqueleto fue OpenMP [72]. Se trata de una colección de directivas de compilación, funciones de librería y variables de entorno que pueden ser usadas para especificar el paralelismo en arquitecturas de memoria compartida. En la memoria se explica de forma detallada el uso básico de estas dos herramientas.

Se ha considerado el Problema de la Mochila 0/1 [69] como caso de estudio con el que se realizará durante la memoria la exposición de la metodología de resolución aportada, la comparativa con otras metodologías y el estudio de los resultados computacionales.

Dicho problema se define de la siguiente manera: “Se dispone de una mochila de capacidad C y de un conjunto de N objetos. Se supone que los objetos no se pueden fragmentar en trozos más pequeños, así pues, se puede decidir si se toma un objeto o si se deja, pero no se puede tomar una fracción de un objeto. Supóngase además, que el objeto k tiene beneficio p_k y peso w_k, para $k = 1, 2, ..., N$. El problema consiste en averiguar qué objetos se han de insertar en la mochila sin exceder su capacidad, de manera que el beneficio que se obtenga sea máximo". Su formulación como problema de optimización es la siguiente:

$$\begin{array}{rll} \textit{máx} & \sum_{k=0}^{N-1} p_k x_k & \\ \text{sujeto a:} & \sum_{k=0}^{N-1} w_k x_k & \leq \ C \\ & x_k \in \{0,1\} \quad k \in \{0, \ldots, N-1\} & \end{array}$$

Para resolver el problema mediante la técnica de Ramificación y Acotación, se explora un árbol en cuya raíz no está fijado el valor de ninguno de los x_i, y en cada nivel sucesivo se va determinando el valor de una variable más, por orden numérico de las variables. Cada nodo que se explora tiene dos sucesores, dependiendo de si se introduce o no el siguiente objeto en la mochila. Siempre que se genera un nodo se calcula una cota superior y una cota inferior del valor de la solución que se puede obtener completando la capacidad parcialmente especificada, y utilizando estas cotas superiores para cortar ramas inútiles y guiar la exploración del árbol.

Se tomará como cota superior "la suma de beneficios de incluir objetos en la mochila hasta que la capacidad actual sea alcanzada y del objeto que haga que la capacidad se supere sólo se incluye la proporción de capacidad que encaje en la mochila". Sea $i \in \{0, 1\}$ el objeto que hace que se supere la capacidad, la formulación matemática de la cota superior U es la siguiente:

$$U = ptf + (p_i * (C_{res} - weigth))/w_i$$

$$\begin{array}{rl} \text{siendo:} & pft = \sum_{j=0}^{i-1} p_j, \\ & weigth = \sum_{j=0}^{i-1} w_j \\ & i \ el \ objeto \ que \ haga \ que \ se \ supere \ la \ capacidad. \end{array}$$

La cota inferior elegida es “la suma de beneficios de incluir objetos en la mochila hasta que la capacidad actual sea alcanzada, sin incluir el objeto que haga que se supere”, esto es:

$$L = \sum_{j=0}^{i-1} p_j.$$

Capítulo 2: Trabajos Relacionados

Desde 1995 se han desarrollado varias herramientas sobre implementaciones de esqueletos en paralelo para la técnica de Ramificación y Acotación. Este capítulo constituye una revisión bibliográfica de las mismas. La mayoría de estas herramientas se encuentran al hacer una búsqueda en Internet para el tópico “Ramificación y Acotación”. Cada sección del capítulo II está dedicada a la descripción de una de ellas, siendo el orden elegido para ello el cronológico.

La primera de las herramientas presentada es PPBB (Portable Parallel Branch-and-Bound Library). PPBB fue implementada y testeada en la Universidad de Paderborn en 1996. Esta librería está escrita en C y fue diseñada para ejecutarse sobre arquitecturas de multiprocesadores de memoria distribuida usando funciones de paso de mensaje, concretamente PVM. La primera versión de la librería fue testeada sobre Transputers

La segunda sección está dedicada a PUBB (Parallelization Utility for Branch-and-Bound algorithms). La herramienta PUBB fue desarrollada en la Universidad de Ciencias de Tokyo, Japan. La primera versión de PUBB fue desarrollada en 1995 en lenguaje C. Existe una versión renovada de 1997 con un algoritmo completamente rediseñado que no es una extensión de la vieja implementación. Esta nueva versión está implementada en lenguaje C++. PUBB se ejecuta sobre PVM y los experimentos han sido realizados sobre algunas redes de estaciones de trabajo.

En la tercera sección se estudia la librería BOB (a Branch-and-Bound Optimization liBrary) desarrollada en La Universidad de Versailles, Francia. Esta librería ha sufrido varios cambios desde 1995, cuando aparece la primera versión en lenguaje C. Los autores escriben una nueva versión de la librería llamada Bob++ en 1999 basada en C++. La parte distribuida de Bob++ ha sido implementada con PVM y portada a la máquina distribuida PARAGON. Hemos contactado con los autores y conocemos la existencia una versión en MPI que es ejecutable sobre un cluster de PCs.

PICO (Parallel Integer and Combinatorial Optimize) es la última de las librerías estudiadas en el capítulo II. Es una herramienta orientada a objetos desarrollada en la Universidad de Piscataway, New Jersey, y Sandia National Laboratories, Albuquerque, en 2000. PICO está estructurada como una librería de clases en C++ y la capa paralela se ha implementado usando MPI. Los resultados computacionales fueron obtenidos en un computador paralelo masivo Janus, que consta de miles de procesadores Pentium-II.

Con cada una de estas librerías se ha hecho un estudio de la interfaz de usuario y se ha ilustrado el uso de cada una de ellas con el Problema de la Mochila. Además se detallan para cada una de las librerías tanto las implementaciones secuenciales como las paralelas desarrolladas por sus autores.

En una sección adicional, se han añadido algunas herramientas más generales que aún no siendo específicas de la técnica de ramificación y acotación, se tiene conocimiento de su existencia.

Herramienta	C	C++	MPI	PVM	OpenMP	Orientación a objetos	Equilibrado de la carga	Disponibilidad
PPBB	X			X			X	X
PUBB	X	X		X		X	X	
Bob++	X	X		X				
PICO		X	X			X		

Table 0.1 Comparativa Entre Las Herramientas Estudiadas

La tabla 0.1 resume las principales características de cada una de las herramientas estudiadas. Las columnas 2 y 3 de dicha tabla representan los lenguajes de programación utilizados por las diferentes librerías; como se puede apreciar la mayoría de ellas están escritas en C++, excepto PPBB que fue escrita en C. Las columnas 4 y 5 representan las posibles herramientas de programación paralela basadas en paso de mensaje y la columna 6 representa el uso de OpenMP (herramienta de programación paralela basada en memoria compartida); PPBB, PUBB y Bob++ fueron implementadas usando PVM y únicamente PICO fue implementada bajo MPI, ninguna de ellas está disponible para OpenMP. La columna 7 representa cuáles de las herramientas siguen una metodología orientada a objetos. En la columna 8 se muestra qué librerías proporcionan estrategias de balanceo de carga. Finalmente, en la columna 9, se representa la disponibilidad de las distintas herramientas, donde se puede apreciar como únicamente PPBB está disponible.

Capítulo 3: La Librería MaLLBa

En el capítulo III de la memoria se presenta nuestra propuesta MaLLBa::BnB. El objetivo de este trabajo es desarrollar una metodología de trabajo para resolver problemas de Optimización Combinatoria mediante la técnica de Ramificación y Acotación. Para ello se ha desarrollado una herramienta orientada a objetos (instanciada sobre un conjunto de clases en C++), que hemos denominado MaLLBa::BnB.

Se define *esqueleto algorítmico* como un conjunto de procedimientos que constituyen el armazón con el que desarrollar programas para resolver un problema dado utilizando una técnica particular. Este software presenta declaraciones de clases vacías que el usuario ha de rellenar para adaptarlo a la resolución de un problema concreto.

Las entidades que intervienen en la instanciación de un esqueleto algorítmico MaLLBa son:

- Programador de Esqueletos. Persona que diseña e implementa el esqueleto. Es quien decide el conjunto de clases que el esqueleto proporciona y cuáles de ellas son requeridas.
- Programador de Soluciones. Persona que adapta el problema real al esqueleto rellenando los huecos que el Programador de Esqueletos le solicita.
- Usuario de Esqueletos. Persona que utiliza el esqueleto ya relleno para obtener una solución a un determinado problema. También puede combinar más de un esqueleto.

La librería MaLLBa se ha diseñado de forma que la tarea de la persona que adapta el problema real al esqueleto sea lo más simple posible. En un esqueleto MaLLBa se distinguen dos partes principales: una parte que implementa el *patrón de resolución* y que es completamente proporcionada por la librería, y una parte que el usuario ha de acabar de completar con las características particulares del *problema a resolver* y que será utilizada por el patrón de resolución.

En el diseño de los esqueletos se ha utilizado una metodología de programación orientada a objetos (OO). La facilidad de interpretación del esqueleto es una de las ventajas que aporta esta filosofía, además de las de modularidad, reusabilidad y modificabilidad. Existe una relación bastante intuitiva entre las entidades participantes en el patrón de resolución y las clases que ha de implementar el usuario.

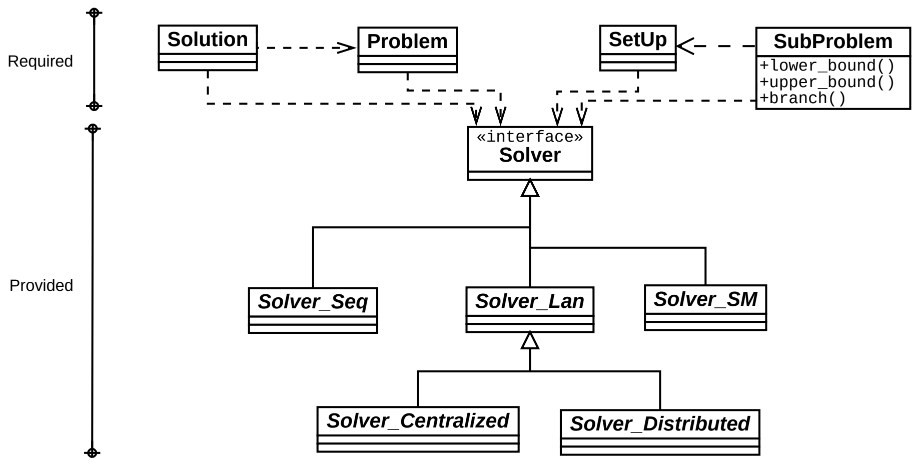


Fig. 0.1 Esquema UML de MaLLBa::BnB

La parte proporcionada por el esqueleto, esto es, los patrones de resolución, se implementan mediante clases en las que explícitamente se indica que son definidas por el esqueleto. A estas clases se les da el nombre de clases *proporcionadas* y aparecen en el código con el calificativo *provides*.

La parte que ha de completar el usuario con la instanciación de su problema particular se implementa mediante clases marcadas con el calificativo *requires*, se las denominará clases *requeridas*. La adaptación que ha de realizar el usuario consiste en representar mediante las estructuras de datos necesarias su problema e implementar (en función de dicha representación) las funcionalidades requeridas por los métodos de las clases. Estas clases serán invocadas desde el patrón de resolución particular (por que conoce la interfaz para dichas clases) de forma que, cuando la instanciación se ha completado, se obtienen las funcionalidades esperadas aplicadas al problema concreto del usuario. La figura 0.1 presenta la arquitectura de la herramienta MaLLBa::BnB.

En los párrafos que siguen se trata cada clase con detalle.

Clases Requeridas

La solución de un problema de optimización combinatoria en general consiste en un vector de enteros que cumple un número de restricciones y optimiza una función objetivo. La función objetivo debe ser maximizada o minimizada. Partiendo de esta descripción, se abstrae que en un esqueleto deben representarse mediante clases tanto el *problema* como la *solución*.

Las clases requeridas se utilizan para almacenar los datos básicos del algoritmo. Con ellas se representan el problema, la solución, los estados del espacio de búsqueda y la entrada/salida. Todos los esqueletos MaLLBa han de definir las siguientes clases:

- `Problem`: define la interfaz mínima estándar necesaria para representar un problema.
- `Solution`: define la interfaz mínima estándar necesaria para representar una solución.

Cada patrón de resolución requiere además un conjunto de clases propias de la técnica algorítmica.

Los algoritmos de Ramificación y Acotación dividen el área de soluciones paso a paso y calculan una cota del posible valor de aquellas soluciones que pudieran encontrarse más adelante. Si la cota muestra que cualquiera de estas soluciones tiene que ser necesariamente peor que la mejor solución hallada hasta el momento, entonces no es necesario seguir explorando esta parte del árbol. Por lo general, el cálculo de cotas se combina con un recorrido en anchura o en profundidad. Para representar este proceso en un esqueleto, se utilizará la clase `Subproblem`.

- La clase `Subproblem`: representa el área de soluciones no exploradas. Esta clase debe proporcionar las siguientes funcionalidades:
  - `initSubProblem(pbm)`: inicializa el subproblema de partida a partir del problema original.

- `branch(pbm, sps, sol)`: ha de proporcionar un algoritmo para dividir el área de soluciones factibles, esto implica, generar a partir del subproblema actual un subconjunto de problemas a ser explorados.
- `lower_bound(pbm,sol)`: este método calcula una cota inferior del mejor valor de la función objetivo que podría ser obtenido para un subproblema dado.
- `upper_bound(pbm,sol)`: esta función calcula una cota superior del mejor valor de la función objetivo que podría ser obtenido para un subproblema dado.

Clases Proporcionadas

Las clases proporcionadas son las que el usuario ha de instanciar cuando utilice un esqueleto, esto es, sólo ha de utilizarlas. Dichas clases son:

- `Setup`: agrupa los parámetros de configuración del esqueleto. Por ejemplo, en esta clase en el esqueleto de Ramificación y Acotación se especifica si la búsqueda en el área de soluciones será en profundidad, en anchura, primero-el mejor, etc.
- `Solver`: esta clase está comprendida por ella misma y sus hijas `Solver_Seq`, `Solver_Lan` y `Solver_SM`. Implementa la estrategia a seguir: Ramificación y Acotación, Divide y Vencerás, etc., y mantiene información actualizada del estado de la exploración durante la ejecución. La ejecución se lleva a cabo mediante la llamada al método `run()` de la clase `Solver_Seq`, `Solver_Lan` o `Solver_SM`.

Una de las aportaciones originales de nuestra propuesta es el uso combinado de esqueletos. Dado que algunos miembros del grupo de paralelismo de La Universidad de La Laguna habían implementado el esqueleto MaLLBa secuencial y paralelo para la técnica Divide-y-Vencerás MaLLBa::DnC hemos realizado algunas colaboraciones para integrar los esqueletos MaLLBa::BnB y MaLLBa::DnC. El algoritmo descrito por Martello y Toth [69] para resolver el Problema de la Mochila 0/1 requiere que los elementos a insertar estén ordenados previamente según la relación beneficio/peso dada por:

$$\frac{p_i}{w_i} \geq \frac{p_{i+1}}{w_{i+1}} \quad (i = 1, ..., N-1)$$

La idea, por tanto, consiste en aplicar el esqueleto Divide-y-Vencerás para ordenar los objetos según dicha relación, y posteriormente aplicar el esqueleto para la técnica de Ramificación-y-Acotación para resolver el problema. Dicha integración de esqueletos nos ha proporcionado algunas publicaciones presentadas en Congresos Internacionales [28, 26, 27].

Algoritmo Secuencial para el Esqueleto MaLLBa::BnB

```
1   $\mathcal{L} := \{\Pi_0\}$; $\mathcal{Q} := \{\Pi_0\}$;
2   $bs := -\infty$; $\mathcal{T} := \emptyset$;
3   while ($\mathcal{L} \neq \emptyset$) {
4       $\Pi_i := s(\mathcal{L})$;
5         if ($upper\_bound(\Pi_i) > bs$) {
6           if ($lower\_bound(\Pi_i) > bs$) {
7               $bs := lower\_bound(\Pi_i)$;
8               $\mathcal{T} := \{\sigma\}$;
9               // $\sigma$ satisfies $f(x, \sigma) = lower\_bound(\Pi_i)$;
10          }
11          else {
12              (brach) decompose $\Pi_i$ into $\Pi_{i1}, \Pi_{i2}, ..., \Pi_{ik}$
13              $\mathcal{L} := \mathcal{L} \cup \{\Pi_{i1}, \Pi_{i2}, ..., \Pi_{ik}\}$ - $\{\Pi_i\}$ ;
14          }
15          $\mathcal{Q} := \mathcal{Q} \cup \{\Pi_{i1}, \Pi_{i2}, ..., \Pi_{ik}\}$ ;
16      }
17      $\mathcal{L} := \mathcal{L}$ - $\Pi_i$;
18  }
19  return $bs$ // bs es el valor de la mejor solucion
20  return $\mathcal{S}$ // $\mathcal{S}$ es la mejor solucion
```

Fig. 0.2 Algoritmo Secuencial para la Técnica MaLLBa::BnB (Maximización).

La figura 0.2 muestra el algoritmo secuencial de MaLLBa::BnB para el caso de maximización. Se procede por repetición de la evaluación de los nodos vivos. La función de búsqueda s es quien selecciona el siguiente nodo vivo a ser evaluado, tal que, $s(\mathcal{L}) = \mathcal{L}$. Partiendo de un problema y un subproblema, y usando la técnica de Ramificación y Acotación trata de encontrar la mejor solución (bs). Se extrae el primer subproblema de la cola (línea 4), se le calcula su cota superior e inferior y en el caso de que mejore la mejor solución se actualiza ésta (líneas 5-10). Si el problema está resuelto, se continua con el proceso sacando problemas de la cola y comprobando la mejor solución (bs).

Si todavía no está resuelto se ramifica y el método branch es el encargado de insertar los nuevos subproblemas en la cola (líneas 12-13).

Algoritmos paralelos usando Paso de Mensaje

La primera versión paralela del esqueleto MaLLBa::BnB se ha implementado mediante Paso de Mensajes siguiendo un esquema Maestro/Esclavo. Esta estrategia consiste en lo siguiente: El procesador maestro posee toda la información sobre el espacio de estados, es decir, el número de subproblemas no ramificados del árbol enumerativo y la solución que se va obteniendo tras cada paso. El maestro asigna un subproblema a un procesador esclavo ocioso y recibe de éste los nuevos subproblemas generados. Un esclavo ocioso recibe un subproblema sólo si es probable que conduzca a una solución óptima. Simultáneamente, los esclavos que posean carga de trabajo evalúan los subproblemas recibidos y generan nuevos subproblemas si fuera necesario.

Esta estrategia proporciona ventajas significativas, tales como la facilidad de controlar y paralelizar algoritmos de ramificación y acotación de una manera más natural y directa.

Un esqueleto MaLLBa::BnB está compuesto de los siguientes procesos básicos:

- proceso maestro: Existe un único proceso *maestro* en el esqueleto que controla la actividad de un grupo de procesadores *Esclavos*. Este contiene una mejor solución inicial y el problema inicial. El proceso maestro extrae los subproblemas de la cola de ramificación global y los envía junto con la mejor solución a los esclavos ociosos (líneas 5-7). El maestro recibe de los esclavos sus mejores soluciones (línea 15). Para cada valor que mejore la mejor solución almacenada por el maestro, éste realiza una actualización de la misma (línea 17). La cola de subproblemas generada por cada esclavo con la mejor de todas las mejores soluciones se enlaza a la cola de subproblemas global almacenada por el maestro, de donde partirán los nuevos subproblemas a ser estudiados por los esclavos (línea 27). Finalmente, cuando el problema está resuelto, la mejor solución global se actualiza y todos los esclavos se marcan como ociosos. La figura 0.3 muestra el algoritmo para el proceso maestro.
- proceso esclavo: El número de procesos *esclavos* en una estación de trabajo se especifica en un parámetro de iniciación. El proceso esclavo recibe un problema y una mejor solución (línea 9). Este calcula la cota superior, la cota inferior y actualiza la mejor solución para ese subproblema (líneas 11-14). El esclavo envía la mejor solución al Maestro (línea 21) y en el

```
$\mathcal{P} := \{1, ..., p\}$; $\mathcal{I} := \mathcal{P}$;
$bs := -\infty$; $\mathcal{T} := \emptyset$;
while $((\mathcal{L} \neq \emptyset)$ and $(\mathcal{I} \neq \mathcal{P}))$ {
    while $((\mathcal{L} \neq \emptyset)$ or $(\mathcal{I} > 0))$ {
        $\Pi_i := s(\mathcal{L})$;
        $i := s(\mathcal{I})$; $\mathcal{I} := \mathcal{I}$ - $\{i\}$;
        $send(\Pi_i, bs, \mathcal{T}, i)$;
    }
    $receive(q, flag) / q \in \mathcal{P}$;
    while $(flag)$ {
        if $(flag =$SOLVED$)$ {
            $receive(q, bs, \mathcal{T})$;
        }
        if $(flag =$BnB$)$ {
            $receive(q, h, bstemp)$;
            if $(h > bs$ $)$ {
                $bs := bstemp$
                $send(\mathcal{L}\_request, q)$
                $receive(q, \Pi_{i1}, \Pi_{i2}, ..., \Pi_{ik})$
            }
            else
                $send(\mathcal{L}\_delete, q)$;
            $\mathcal{I} := \mathcal{I} \cup \{q\}$
        }
        $receive(q, flag) / q \in \mathcal{P}$;
    }
    $\mathcal{L} := \mathcal{L} \cup \{\Pi_{i1}, \Pi_{i2}, ..., \Pi_{ik}\}$
}
$send($END$, i)$; $\forall i \in \mathcal{P}$
```

Fig. 0.3 Algoritmo Centralizado: Maestro (Maximización)

caso de que éste le solicite la cola, ramifica el subproblema, almacena los nuevos subproblemas en una cola temporal y se la envía al Maestro (líneas 24-31). Finalmente, cuando el subproblema está resuelto, sólo le envía al maestro la mejor solución con una etiqueta *SOLVE* (línea 17). La figura 0.4 muestra el algoritmo para un proceso esclavo.

Está aproximación paralela se ha denominado *implementación centralizada*, puesto que el procesador maestro es quién distribuye el trabajo entre los procesadores esclavos. De los experimentos que se realizaron sobre el cluster de PCs de La Laguna en Linux se desprende la necesidad de mejorar esta implementación, dado que se invierte una gran parte del tiempo de ejecución únicamente en comunicaciones.

Se parte del algoritmo explicado anteriormente para desarrollar un algoritmo que mejora la primera versión paralela. Esta nueva versión es la que se denomina *implementación distribuida*. En este caso los procesos básicos son:

- proceso maestro: Existe un único proceso *maestro* en el esqueleto. El proceso maestro es el responsable de la coordinación entre las tareas. Para ello, cuenta con una estructura de datos donde registra el estado de ocupación de cada uno de los esclavos. Al comienzo de la ejecución todos los procesos esclavos se marcan como ociosos (línea 1). El subproblema inicial, el mejor valor de la función objetivo y el vector de la mejor solución obtenida hasta el momento se envían al primer esclavo ocioso (línea 4). Mientras existan esclavos libres el maestro recibe información de ellos y decide la siguiente acción a ejecutar dependiendo de si el problema está o no resuelto, si hay una solicitud de esclavos ociosos para colaborar en la resolución del problema o si los esclavos no tienen trabajo que realizar. En el caso de que el problema esté resuelto, el maestro recibe el mejor valor de la función objetivo y el vector solución y los almacena (línea 9). Cuando lo que recibe el maestro es una solicitud de esclavos libres, ésta viene acompañada del valor de la cota superior (línea 12). Si el valor de la cota superior es mayor que el valor actual de la mejor solución, la respuesta al esclavo incluye el número de esclavos libres que pueden ayudar a resolver el problema (línea 15). En otro caso, la respuesta indica que no es necesario trabajar en ese subárbol de búsqueda (línea 18). Cuando el número de esclavos libres es igual al valor inicial, el proceso de búsqueda finaliza y el Maestro notifica a todos los esclavos que dejen de trabajar (línea 27). La figura 0.5 muestra el algoritmo para el proceso maestro.

```
while ($true$) {
    $receive(q, flag)/q \in \mathcal{P}$;
    while ($flag$) {
      if ($flag =$END) {
        $receive(Master,$END$)$;
        return;
      }
      if ($flag =$PBM) {
        $receive(q, \Pi_i, bs_i, \mathcal{T}_i)$;
        $\mathcal{L} := \{\Pi_i\}$; $\mathcal{Q} := \{\Pi_i\}$; $bs := bs_i$; $\mathcal{T} := \mathcal{T}_i$;
        if ($upper\_bound(\Pi_i) > bs$) {
          if ($lower\_bound(\Pi_i) > bs$) {
            $bs := lower\_bound(\Pi_i)$;
            $\mathcal{T} := \{\sigma\}$; / $f(x, \sigma) = lower\_bound(\Pi_i)$;
          }
        }
        if (SOLVED) {
          $send(bs, \mathcal{T},$SOLVED$, Master)$;
        }
        else {
          $send(bs, upper\_bound(\Pi_i),$ BNB$, Master)$;
        }
        $receive(Master, \mathcal{L}\_request)$;
        if ($\mathcal{L}\_request$ ) {
          *(brach)* decompose $\Pi_j$ into $\Pi_{j_1}, \Pi_{j_2}, ..., \Pi_{j_k}$
          $\mathcal{L} := \mathcal{L} \cup \{\Pi_{j_1}, \Pi_{j_2}, ..., \Pi_{j_k}\}$ - $\{\Pi_j\}$;
          $\mathcal{Q} := \mathcal{Q} \cup \{\Pi_{j_1}, \Pi_{j_2}, ..., \Pi_{j_k}\}$;
          while ($\mathcal{L} \neq \emptyset$) {
            $\Pi_n := s(\mathcal{L})$; $\mathcal{L} := \mathcal{L} - \Pi_n$;
            $send(\Pi_n, Master)$;
          }
        }
      }
      $receive(q, flag)/q \in \mathcal{P}$;
    }
}
```

Fig. 0.4 Algoritmo Centralizado: Esclavo (Maximización)

```
$\mathcal{P} := \{1, ..., p\}$; $\mathcal{I} := \mathcal{P}$;
$bs := -\infty$; $\mathcal{T} := \emptyset$;
$i := s(\mathcal{I})$; $\mathcal{I} := \mathcal{I}$ - $\{i\}$;
$send(\Pi, i)$;
while ($\mathcal{I} \neq \mathcal{P}$) {
    $receive(q, flag) / q \in \mathcal{P}$;
    while ($flag$) {
      if ($flag =$SOLVED) {
        $receive(q, bs, \mathcal{T})$;
       }
       if ($flag =$BnB) {
         $receive(q, h, r)$;
         if ($h > bs$ ) {
             $\mathcal{I} := \mathcal{I} - \{p_1, \ldots, p_r\} / p_k = s(\mathcal{I})$;
             $send(bs, \{p_1, \ldots, p_r\}, q)$;
         }
         else
         $send($DONE$, q)$;
       }
       if ($flag =$IDLE) {
         $receive(q,$ IDLE$)$;
         $\mathcal{I} := \mathcal{I} \cup \{q\}$
       }
       $receive(q, flag) / q \in \mathcal{P}$;
    }
 }
 $send($END$, i)$; $\forall i \in \mathcal{P}$
```

Fig. 0.5 Algoritmo Distribuido: Maestro (Maximización)

- proceso esclavo: El número de procesos *esclavos* en una estación de trabajo se especifica en un parámetro de iniciación. Cada esclavo trabaja acotando los problemas que recibe (líneas 12-14). Éste genera nuevos subproblemas mediante llamadas a la función de ramificación (branch()) (líneas 19-20). El esclavo pide información sobre esclavos libres al Maestro (línea 21). Si no hay otros esclavos libres que le puedan ayudar en su tarea, el esclavo continúa trabajando localmente. En otro caso, extrae subproblemas de su cola local y los envía directamente a los otros esclavos que le hayan asignado por el maestro (línea 23-25). La figura 0.6 muestra el algoritmo para un proceso esclavo.

Una diferencia notable con respecto al algoritmos centralizado es que en este caso la carga de trabajo es distribuida entre los procesadores esclavos por ellos mismos, en lugar de que la distribuya el Maestro como ocurría en el primer caso. Con esto se consigue reducir notablemente el coste de las comunicaciones.

Algoritmo paralelo usando Memoria Compartida

El tercer esquema paralelo de trabajo propuesto en la memoria está implementado para memoria compartida. El algoritmo presentado en la figura 0.7 consiste en lo siguiente: en primer lugar se calcula y establece el número de "threads" P que intervienen. Se extraen a continuación P subproblemas de la estructura de datos y se asigna uno a cada "thread". En la región paralela cada "thread" asignado trabaja sobre su propio subproblema calculando las cotas inferiores y superiores. El valor de la mejor solución y el vector solución deben ser modificados cuidadosamente, puesto que sólo un "thread" puede cambiar estas variables en cada momento. El mismo cuidado especial debe tenerse en cuenta cuando un "thread" intenta insertar los nuevos subproblemas generados en la estructura de datos global compartida.

```
while ($true$) {
    $receive(q, flag)/q \in \mathcal{P}$;
    while ($flag$) {
      if ($flag$ =END) {
        $receive(Master$,END);
        return;
      }
      if ($flag$ =PBM) {
        $receive(q, \Pi_i, bs_i, \mathcal{T}_i)$;
        $\mathcal{L} := \{\Pi_i\}$; $\mathcal{Q} := \{\Pi_i\}$; $bs := bs_i$; $\mathcal{T} := \mathcal{T}_i$;
        while ($\mathcal{L} \neq \emptyset$) {
            $\Pi_j := s(\mathcal{L})$;
            if ($upper\_bound(\Pi_j) > bs$) {
              if ($lower\_bound(\Pi_j) > bs$) {
                $bs := lower\_bound(\Pi_j)$;
                $\mathcal{T} := \{\sigma\}$; / $f(x, \sigma) = lower\_bound(\Pi_j)$;
                $send(bs, \mathcal{T}$,SOLVED, $Master)$;
              }
              (brach) decompose $\Pi_j$ into $\Pi_{j_1}, \Pi_{j_2}, ..., \Pi_{j_k}$
              $\mathcal{L} := \mathcal{L} \cup \{\Pi_{j_1}, \Pi_{j_2}, ..., \Pi_{j_k}\}$ - $\{\Pi_j\}$;
              $\mathcal{Q} := \mathcal{Q} \cup \{\Pi_{j_1}, \Pi_{j_2}, ..., \Pi_{j_k}\}$;
              $send(bs, k$,BNB, $Master)$;
              $receive(Master, bs, \mathcal{P})$;
              while ($\mathcal{P} \neq \emptyset$) {
                $n := s(\mathcal{P})$; $\mathcal{P} := \mathcal{P} - \{n\}$; $\Pi_n := s(\mathcal{L})$; $\mathcal{L} := \mathcal{L} - \Pi_n$;
                $send(\Pi_n, n)$;
              }
            }
            $\mathcal{L} := \mathcal{L}$ - $\Pi_j$;
        }
        $send($IDLE, $Master)$;
      }
      $receive(q, flag)/q \in \mathcal{P}$;
    }
}
```

Fig. 0.6 Algoritmo Distribuido: Esclavo (Maximización)

```
$\mathcal{L} := \{\Pi_0\}$; $\mathcal{Q} := \{\Pi_0\}$;
$bs := -\infty$; $\mathcal{T} := \emptyset$;
$\mathcal{P} := \{1, ..., p\}$;
while ($\mathcal{L} \neq \emptyset$) {
   while ( $\mathcal{P} \neq \emptyset$) {
      $n := s(\mathcal{P})$; $\mathcal{P} := \mathcal{P} - \{n\}$; $\Pi_n := s(\mathcal{L})$; $\mathcal{L} := \mathcal{L} - \Pi_n$;
   }
   parallel for ($n \in \mathcal{P}$) {
      if ($upper\_bound(\Pi_n) > bs$) {
         if ($lower\_bound(\Pi_n) > bs$) {
            $bs := lower\_bound(\Pi_n)$;
            $\mathcal{T} := \{\sigma\}/f(x,\sigma) = lower\_bound(\Pi_n)$;
         }
         if ($upper\_bound(\Pi_n) \neq lower\_bound(\Pi_n)$ ) {
            (brach) decompose $\Pi_n$ into $\Pi_{n1}, \Pi_{n2}, ..., \Pi_{nk}$
            $\mathcal{L} := \mathcal{L} \cup \{\Pi_{n1}, \Pi_{n2}, ..., \Pi_{nk}\}$ - $\{\Pi_n\}$;
            $\mathcal{Q} := \mathcal{Q} \cup \{\Pi_{n1}, \Pi_{n2}, ..., \Pi_{nk}\}$ ;
         }
      }
   }
}
```

Fig. 0.7 Algoritmo para Memoria Compartida

Capítulo 4: Resultados computacionales

Con el objetivo de tomar una decisión sobre la filosofía de programación a seguir para implementar la herramienta MaLLBa se desarrolló un estudio de los tiempos requeridos por el mismo algoritmo de ramificación y acotación secuencial implementado usando un lenguaje imperativo y un lenguaje orientado a objetos. Después de algunas reuniones de coordinación de los miembros del proyecto, se optó por seguir una metodología orientada a objetos, dadas las ventajas que esta filosofía ofrece: modularidad, reusabilidad, modificabilidad, facilidad de interpretación de los esqueletos, etc. El lenguaje de programación elegido para la implementación de MaLLBa fue C++.

Con respecto a la herramienta paralela a usar para desarrollar el projecto MaLLBa previamente a tomar una decisión se realizaron algunos experimentos para medir el rendimiento de las comunicaciones. Los experimentos consistieron en la implementación del patrón de comunicaciones Ping-Pong entre dos máquinas en tres áreas diferentes usando PVM y MPI: dos máquinas de área local pertenecientes al cluster de La Laguna, dos máquinas del cluster de Barcelona y dos máquinas a una gran distancia: una del cluster de La Laguna y una del cluster de Barcelona. Dados los tiempos obtenidos se optó por implementar los esqueletos que hacen uso del paso de mensaje con MPI.

Las máquinas sobre las que se ejecutaron los experimentos usando los distintos esqueletos proporcionados MaLLBa para la técnica ramificación y acotación fueron las siguientes:

- Sunfire 6800 SMP, con la siguiente configuración:
  24 750 MHz UltraSPARC-III procesadores,
  48 Gbyte de memoria cada uno y
  120 Gbyte de espacio de disco por sistema.
  Esta máquina pertenece al EPCC (Edinburgh Parallel Computing Center) en La Universidad de Edimburgo [33].

- Origin 3000, cuya configuración es
  160 procesadores a 600 MHz MIPS R14000,
  1 Gbyte de memoria cada uno y
  900 Gbyte de disco.
  El CIEMAT (Centro de Investigaciones Energéticas, Medioambientales y Tecnológicas) ha permitido el acceso a estas máquinas [13].

- Un 344-procesador T3E-900/LC configurado con:
  344 450 MHz procesadores Alpha, cada uno con un pico de rendimiento

de 309 GFlops con una mezcla de 64 y 128 Mbytes de memoria.
También perteneciente al EPCC.

- Un cluster heterogeneo de PCs, con una configuración de
2 procesadores AMD Duron a 750 MHz,
4 procesadores AMD Duron a 800 MHz ,
7 procesadores AMD-K6 3D a 500 MHz , con
256 Mbyte de memoria y
32 Gbyte de disco duro cada uno.
Este cluster de PCs pertenece al grupo de paralelismo de la Universidad de La Laguna [65].

En el apartado de resultados computacionales se presentan las tablas y gráficas con los resultados obtenidos para las implementaciones MPI y OpenMP de los esqueletos paralelos aplicados al Problema de La Mochila. Se muestran también las comparativas entre los tiempos de ejecución para los casos en los que sólo se calcula la mejor solución del Problema de la Mochila y cuando se calcula además el vector de soluciones. Los tamaños del problema para los que se presentan los resultados son 50.000 y 100.000.

También se realiza un estudio usando las herramientas CALL y LLAC, que facilitan el análisis de complejidad de aplicaciones tanto secuenciales como paralelas. El análisis de complejidad de un algoritmo produce como resultado una "función de complejidad" que da una aproximación del número de operaciones que realiza. La herramienta CALL [5] permite anotar con dicha fórmula de complejidad el código C que implementa el algoritmo. Se desea estudiar el número de nodos visitados que, en este caso, es un valor distribuido entre los esclavos puesto que el Maestro no realiza labores de acotación. Por lo tanto, se anotará el código paralelo exactamente igual que el caso secuencial, cambiando sólo el lugar donde se incrementa el número de nodos visitados. Las directivas CALL que lo hacen se han añadido en los puntos en los que se extraen problemas de la cola local de cada esclavo ya sea para estudiarlo o para enviárselo a otro esclavo para que lo resuelva. La herramienta LLAC, por ser una extensión de R, proporciona la posibilidad de hacer un análisis de las muestras que se obtienen al ejecutar los experimentos generados con CALL. Con las mismas muestras se pueden realizar distintos tipos de representaciones para estudiar cuanto se ajustan la fórmula con la que habíamos anotado el programa y los resultados obtenidos.

Las ejecuciones secuenciales del problema de la mochila se ejecutaron sobre un procesador AMD-Duron a 800 MHz y 256 Mb de memoria. El experimento consistió en generar aleatoriamente diez problemas de la mochila con capacidad en el rango [500, 5000]. Presentamos en este apartado los resultados obtenidos en varias gráficas. De las gráficas se concluye que el número medio de nodos visitados se aproxima a una parábola y que a mayor número de objetos mayor dispersión en el número de nodos visitados. Se cree que este comportamiento está ligado al generador de problemas aleatorio que se está utilizando.

Otro parámetro interesante a estudiar en las implementaciones paralelas de los algoritmos de ramificación y acotación es el "equilibrado de la carga de trabajo" entre los distintos procesadores que intervienen en la ejecución del algoritmo. Representando gráficamente la distribución media de nodos visitados entre ocho procesadores de cinco ejecuciones de un problema de la mochila generado aletoriamente de tamaño 4000, se obtienen las siguientes conclusiones: el procesador cero no visita ningún nodo puesto que se trata del Maestro, el primero de los esclavos explora la mayor parte del espacio de búsqueda porque se trata de un procesador más rápido y, debido al tamaño del problema, no queda mucho trabajo para los últimos procesadores ociosos. Además se aprecia una gran diferencia con respecto al número de nodos visitados entre una ejecución y la siguiente.

Conclusiones (en Español)

En un gran número de problemas combinatorios, el tiempo empleado para obtener una solución usando un computador secuencial es muy alto. Esto es inaceptable en aplicaciones de tiempo real. Una forma de solventar este inconveniente consiste en utilizar la computación paralela. En un *computador paralelo*, varios procesadores colaboran para resolver simultáneamente un problema en una fracción del tiempo requerido por un sólo procesador. Esta idea no es nueva, sino que se ha utilizado de forma natural cada vez que es necesario realizar un trabajo de gran envergadura. Sin embargo, desde hace unos años esta idea es aplicable en Informática gracias al costo en declive de los procesadores y los avances en microelectrónica.

Entre los componentes claves necesarios para que sea posible la aplicación de la computación paralela están la arquitectura (o "hardware"), el sistema operativo y los compiladores de lenguajes de programación (o "software"). Sin embargo, el más importante de todos es el *algoritmo paralelo.* Ningún problema se puede resolver en paralelo sin un algoritmo paralelo. Del mismo modo que los algoritmos ocupan un lugar central en Ciencias de la Computación, los algoritmos paralelos son el núcleo de la computación paralela. El objetivo de este trabajo era el desarrollo de una metodología de trabajo para abordar la resolución de problemas de optimización combinatoria mediante la técnica de Ramificación y Acotación utilizando paralelismo. Partiendo de casos concretos se quería generalizar una forma de trabajar que diera lugar a la resolución de problemas diversos. Para ello se utilizó el concepto de *esqueleto*. El concepto de esqueleto paralelo que se implementó no es de tan bajo nivel de programación como el presentado por Cole [15]. Sin embargo, el nivel de programación es inferior al utilizado en el *Diseño de Patrones* [36].

Herramienta	C	C++	MPI	PVM	OpenMP	Orientación a objetos	Equilibrado de la carga	Facilidad de la interfaz
MaLLBa::BnB	X	X	X		X	X		X
PPBB	X			X			X	X
PUBB	X	X		X		X	X	
Bob++	X	X		X				
PICO		X	X			X		

Table 0.2 Comparativa entre las herramientas estudiadas y MaLLBa::BnB

En la memoria se realiza una revisión bibliográfica de los trabajos relacionados. Las herramientas que más se asemejan al trabajo desarrollado son: PPBB [92], PUBB [84], BOB [60] y PICO [32]. En la tabla 0.2 se resumen las principales características de las herramientas relacionadas estudiadas incluyendo nuestra propuesta MaLLBa::BnB. La primera columna enumera las herramientas analizadas. En las columnas segunda y tercera se resumen los lenguajes de programación usados para implementar estas herramientas. Las columnas cuarta, quinta y sexta especifican el uso de memoria distribuida con MPI o PVM, y el uso de memoria compartida con OpenMP en las implementaciones paralelas. La columna séptima muestra el uso apropiado de una metodología orientada a objetos. En la columna octava se especifican las herramientas que proporcionan estrategias de equilibrado de la carga de trabajo. Finalmente, la facilidad de uso de la interfaz de las herramientas es expresada en la columna novena.

En esta memoria se han presentado varias propuestas de esqueletos para la estrategia Ramificación y Acotación: un esqueleto secuencial, dos propuestas de esqueletos paralelos usando paso de mensaje y un esqueleto para memoria compartida. Enfatizamos el alto nivel programación ofrecido al usuario en los algoritmos, que le permite a los esqueletos ser fácilmente transformables para resolver otros problemas de optimización combinatoria. Cuando se usa MaLLBa::BnB como en cualquier otro sistema orientado a objetos, el usuario tiene que poner cada parte de código en la posición correcta. Una vez que las clases `Problema`, `Solución`, y `Subproblema` han sido modificadas, rellenando los datos básicos y definiendo las funcionalidades, el usuario obtiene un resolutor secuencial y tres resolutores paralelos sin esfuerzo de programación añadido. Una ventaja de nuestra propuesta es que el usuario no necesita tener conocimientos previos en paralelismo.

Como metodología de trabajo se proponen los siguientes pasos:

1.- representar las estructuras de datos que compondrán el *Problema*, el *Subproblema* y la *Solución*

2.- obtener el resolutor secuencial, obtener el resolutor paralelo y ver si con ellos se consiguen resultados satisfactorios

3.- en el caso de que los resultados no sean satisfactorios estudiar las propuestas paralelas para implementar una resolución AD-HOC.

El objetivo principal de la interface de usuario de MaLLBa::BnB es simplificar la tarea de los investigadores y usuarios que tienen que implementar algoritmos usando la técnica de Ramificación y Acotación. MaLLBa::BnB proporciona al usuario un valor añadido no sólo en términos de la cantidad de código a escribir, sino también modularidad y claridad conceptual. MaLLBa::BnB hace un uso equilibrado de la programación orientada a objetos. Proporciona los modelos necesarios mínimos para obtener una buena eficiencia computacional y, a la vez, hace el proceso de compilación seguro.

Además el sistema pone al alcance del usuario la posibilidad de generar y experimentar con nuevos algoritmos que permiten la integración de técnicas.

En este trabajo ha sido mostrado como es posible la integración de técnicas diferentes por medio del Problema de Mochila.

Los resultados computacionales obtenidos para un número grande de procesadores muestran que los algoritmos propuestos no son escalables. Esta no es una tendencia general en los algoritmos de ramificación y acotación. La carencia de escalabilidad de nuestros resultados puede ser una consecuencia de la sobrecarga de las comunicaciones. Una futura línea de trabajo para mejorar esta deficiencia será parametrizar el número de subproblemas que ha de resolver cada procesador antes de realizar algún tipo de solicitud al maestro.

Otra futura línea de investigación para mejorar la escalibilidad del algoritmo de paso de mensaje presentado consistirá en mejorar la gestión de la memoria. Actualmente, toda la información relacionada con un subproblema se agrupa en los subproblemas, que viajan entre procesadores. La optimización del algoritmo es posible haciendo que el subproblema comparta información común en toda la trayectoria de decisiones. Una solución natural, necesariamente, implica la distribución del vector de decisiones entre procesadores diferentes y la introducción de mecanismos para recuperar la información distribuida. En esta línea, se desarrollará un algoritmo totalmente distribuido donde no existe diferencia entre el procesador maestro y los procesadores esclavos. Para ello, se usarán dos punteros adicionales en cada subproblema para enlazar el nodo padre con sus dos nodos hijos. Aplicando la técnica de ramificación y acotación, para liberar un nodo del árbol de búsqueda, sus dos subproblemas hijos deben haberle especificado previamente que la mejor solución no se encontrará en las ramas del árbol que ellos generan. Para implementar este esquema la estructura de datos a seguir será la que se muestra en la figura 0.8.

En el caso de memoria compartida, el algoritmo presentado es muy simple y fácil de entender. Entre los trabajos futuros también se propone la implementación de un esqueleto para mezclar OpenMP y MPI. Este enfoque permitirá una mejor gestión de los recursos computacionales, especialmente en arquitecturas de híbridas de memoria distribuida-compartida. El esquema a desarrollar consiste en: cada multiprocesador está compuesto por dos tipos de procesos distintos: uno para realizar tareas de comunicación con los multiprocesadores vecinos y otro para administrar la cola de subproblemas compartida. La comunicación entre multiprocesadores se realiza mediante los procesos de comunicación usando la herramienta MPI. En este caso, cada multiprocesador comunica únicamente con los dos más cercanos (de forma circular), y no se permite la comunicación de todos los procesadores con el resto de procesadores. La figura 0.9 muestra esta futura línea de investigación.

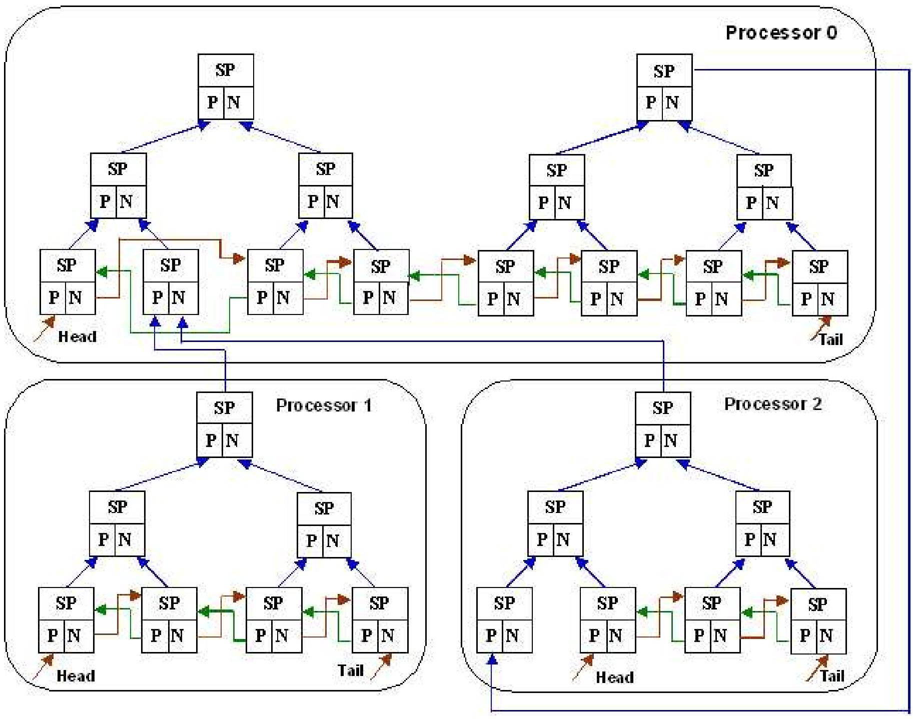


Fig. 0.8 Cola Totalmente Distribuida

Del análisis realizado se ha descubierto que el número de nodos visitados del espacio de búsqueda para los datos generados aleatoriamente es de orden n^2. Un objetivo a desarrollar es tratar de generar un problema con suficiente grano para obtener algún incremento en el rendimiento paralelo. También se está haciendo una búsqueda de problemas más apropiados para resolver usando la técnica de ramificación y acotación.

La elección del lenguaje de programación no ha sido la más adecuada en términos de herramientas disponibles; puesto que es difícil encontrar OpenMP para C++, Paraver para C++ y CALL-LLAC trabajan sobre código en C. Se ha solicitado cuenta en la máquina de Barcelona para tratar de ejecutar MaLLBa::BnB para el esqueleto de memoria compartida con PARAVER.

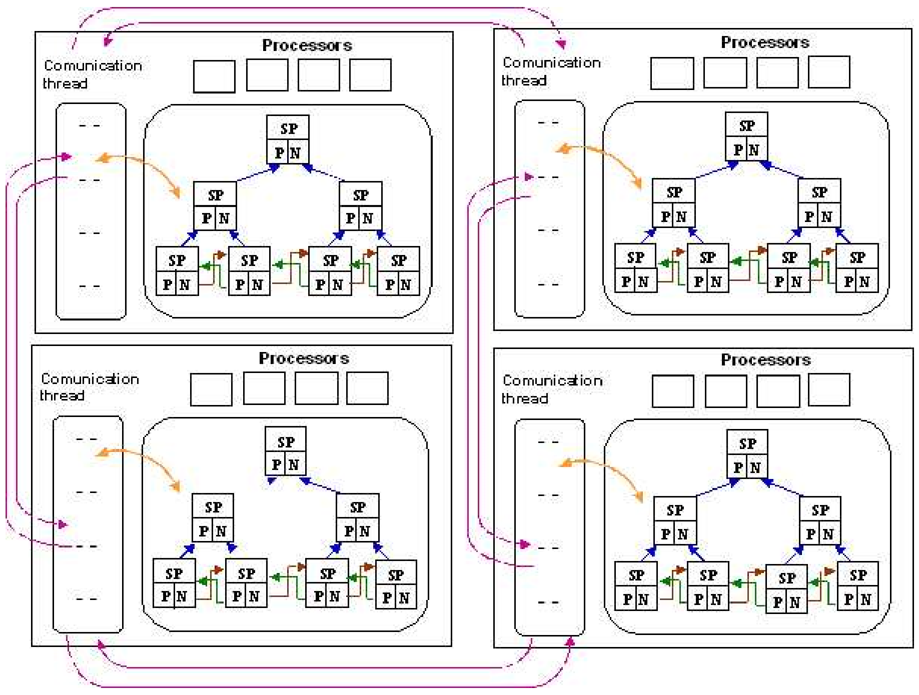


Fig. 0.9 Esquema OpenMP-MPI

Aportaciones (en Español)

Esta memoria incluye los resultados de diversos trabajos de investigación que han sido presentados en Congresos Internacionales, Nacionales y Revistas Internacionales en los campos de la Optimización Combinatoria y del Paralelismo.

En el capítulo III se presenta nuestra propuesta MaLLBa::BnB. El objetivo de esta tesis es desarrollar un entorno objeto orientado (instanciado sobre un conjunto de clases en C++) que proporciona las herramientas necesarias para la resolución de problemas de Optimización Combinatoria mediante el paradigma Ramificación y Acotación.

La librería MaLLBa:BnB está basada en la idea de esqueleto algorítmico. La librería MaLLBa le proporciona al usuario un interfaz único para cada esqueleto algorítmico, que es independiente de la plataforma en la que se ejecute. La participación del usuario final se limita a escoger un método de resolución y rellenar las características en el esqueleto algorítmico, y obtiene los resolutores tanto secuencial como paralelo sin tener que modificar el código. La herramienta proporciona un nivel alto de programación, que facilita el uso de redes de ordenadores bajo Linux, y multiprocesadores. Esta librería permite el uso de máquinas paralelas a usuarios no expertos en el paralelismo para resolver los problemas en el área de optimización. Se presentan dos implementaciones paralelas que hacen uso de MPI (Interfaz de Paso de Mensaje), la herramienta estandard en Programación de Paso de Mensaje. Esta elección permite adaptar de una manera fácil la herramienta a las diferentes plataformas paralelas. Además, se ha desarrollado una versión de la herramienta usando OpenMP, con el objetivo de portarla a plataformas paralelas de Memoria Compartida.

En el capítulo IV, se presentan los resultados computacionales obtenidos. Se ha realizado un cuidadoso diseño de los experimentos y los resultados obtenidos son estadísticamente estudiados para establecer el rendimiento de la librería.

En el capítulo V, se presentan las conclusiones obtenidas sobre la herramienta MaLLBa para la técnica de Ramificación y Acotación y algunas futuras líneas de investigación.

La implementación de varios problemas haciendo uso de MaLLBa::BnB se muestra en el Apéndice C.

Una de las novedades de la librería MaLLBa es que permite la integración de técnicas diferentes, que habían sido desarrolladas separadamente pero no la combinación de ellas. Del trabajo coordinado de algunos miembros del Grupo de Paralelismo de la Universidad de La Laguna en la integración de las técnicas Ramificación-y-Acotación y Divide-y-Vencerás se realizan las siguientes aportaciones:

- La integración para el caso secuencial de ambos esqueletos titulada "Resolución del Problema de la Mochila 0/1 usando esqueletos Divide-y-Vencerás y Ramificación-y-Acotación" fue presentado en "5th International Conference on Operations Research" en La Habana, 2002, que ha sido publicado por la revista Investigación Operacional, pp. 4-13, Vol. 25, No. 1, 2004.

- El esquema paralelo centralizado para el esqueleto de Ramificación-y-Acotación se integra con el esqueleto Divide-y-Vencerás paralelo. Con el título "Branch-and-Bound and Divide-and-Conquer Parallel Skeletons: an Integrated Approximation", se presenta esta contribución en el "International Symposium on Combinatorial Optimization" en Paris, 2002.

- El esquema mejorado para el paradigma Maestro/Esclavo, que denominamos esquema distribuido, fue presentado en el "Eleventh Euromicro Conference on Parallel, Distributed and Network Based Processing", en Genoa, 2003. Con el título "Parallel Skeletons for Divide-and-Conquer and Branch-and-Bound Techniques" ha sido publicado en el libro de actas del congreso por IEEE Computer Society.

En el capítulo III se presentan además las implementaciones de los esqueletos paralelos para paso de mensaje y memoria compartida.

Un esquema para memoria compartida usando OpenMP fue presentado en “Eighth International WorkShop on High-Level Parallel Programming Models and Supportive Environments”, en Nice 2003. El artículo titulado “A comparison between MPI and OpenMP Branch-and-Bound Skeletons”, fue publicado en el libro de actas del congreso por IEEE Computer Society.

La formulación matemática de los algoritmos para los esqueletos de Ramificación-y-Acotación fue presentada en “Fifth International Conference on Parallel Processing and Applied Mathematics (PPAM2003)” en Czestochowa, 2003. El artículo titulado “Parallel Branch-and-Bound Skeletons: Message Passing and Shared Memory implementations” será publicado en el libro de actas del congreso por Springer Verlag Series Lecture Notes in Computer Science.

Un ejemplo del uso completo de la Librería MaLLBa::BnB fue presentado en “Parallel Computing 2003 (Parco2003)”, en Dresden 2003, con el título “MPI and OpenMP implementations of Branch-and-Bound Skeletons”. El libro de actas del congreso será publicado por Elsevier Science.

Una recopilación de las aportaciones a estas tres Conferencias Internacionales será publicado bajo el título “Comparing MPI y OpenMP las implementaciones del 0-1 Problema de Mochila” en la revista “Parallel and Distributed Computing Practices”.

En las “IX Jornadas de Enseñanza Universitaria de la Informática”, Cádiz en 2003, se presentó un análisis de los resultados computacionales usando las herramientas CALL y LLAC con el título “Complejidad Algorítmica: de la Teoría a la Práctica”. Este trabajo es el resultado de la coordinación con algunos miembros del Grupo de Paralelismo de La Universidad de La Laguna, quienes desarrollaron ambas herramientas.

El trabajo desarrollado ha sido parcialmente financiado por tres proyectos: El Proyecto MaLLBa (TIC1999-0754-C03) titulado “Entornos Geográficamente Distribuidos: Librería de Optimización Combinatoria”, agrupa investigadores de tres instituciones españolas y ha sido financiado por CICYT (Comisión Interministerial de Ciencia y Tecnología). Los miembros de proyecto incluyen personal del “Departamento de Lenguajes y Ciencias de la Computación” (Universidad de Málaga), “Departamento de Estadística, Investigación Operativa y Computación (Universidad de La Laguna), y “Department de Llenguatges i Sistemes Informátics” (Universidad Politécnica de Cataluña).

El projecto “MAGOS: Librería Paralela para la Integración de Técnicas Divide-y-Vencerás y Ramificación-y-Acotación y su aplicación en la resolución

del Problema de Corte" está financiado por el Gobierno de las Islas Canarias y su número de referencia es: PI-2001/60.

También hemos recibido subvención del "European Community-Access to Research Infrastructure action of the Improving Human Potential Programme" con número de referencia HPRI-CT-1999-00026 (TRACS Program at EPCC-Edinburgh Parallel Computing Center). También hemos usado los computadores del CEPBA (Centro Europeo de Paralelismo de Barcelona). Finalmente, el CIEMAT (Centro de Investigaciones Energéticas, Medioambientales y Tecnológicas) nos ha permitido acceder a sus computadores.

Como resultado de la coordinación de estos proyectos y el uso de estos computadores se presentaron contribuciones a varios congresos. En 2000, el artículo titulado "Algoritmos Exactos de Optimización Combinatoria: Una Aproximación Geográficamente Distribuida" fue presentado en "XI Jornadas de Paralelismo", en Granada. En 2001, se presentó el artículo titulado "MALLBA: Towards a Combinatorial Optimization Library for Geographically Distributed Systems" en las "XII Jornadas de Paralelismo", en Valencia. Ambos fueron publicados en las actas del congreso. Finalmente, "MALLBA: A Library of Skeletons for Combinatorial Optimisation" fue presentado en Euro-Par, Paderborn 2002. Los proceedings del congreso fueron publicados por Springer-Verlag Series Lecture Notes in Computer Science.

En la actualidad, estamos desarrollando el proyecto TRACER, titulado "Técnicas Avanzadas de Optimización para Problemas Complejos", que agrupa a investigadores de cinco Universidades Españolas. Los miembros del proyecto incluyen personal de las tres Universidades que desarrollaron MaLLBa y miembros de otras dos universidades: La Universidad de Extremadura y La Universidad Carlos III. Este trabajo ha sido cofinanciado por la CEE (FEDER) y por el Ministerio de Ciencia y Tecnología a través del Plan Nacional de I+D+I número TIC2002-04498-C05-05.

Acknowledgements

I would like to thank my supervisor Coromoto León Hernández for the incalculable amount of time she has spent during these years and the extraordinary patience she has had teaching me many of the tools needed in Parallel Computing. I am also really grateful for her great help and useful advice in developing and writing this memory, which would probably have been impossible without her support.

Also, I want to thank to all members of *Departamento de Estadística, Investigación Operativa y Computación* that have helped me somehow or other, and particularly to the members of the Parallel Computing Group of La Laguna University with whom I have collaborated. My express gratitude to Andrés Rodríguez González for his help throughout the years we have shared the office, the job and attended many social events.

Finally, I want to mention my family, in particular my brother Pablo and my friends for their kindness and support.

Preface

Combinatorial Optimization is one of the branches of Mathematical Programming which has achieved remarkable progress in the last two decades. It is closely related to other disciplines such as Operations Research, Computer Science and Combinatorial Mathematics. A way of approaching the resolution of the combinatorial optimization problems is through the enumeration techniques or exact techniques. The need for reducing the number of subproblems studied take us towards techniques such as Branch-and-Bound or Dynamic Programming. Branch-and-Bound algorithms are general methods applied to various combinatorial optimization problems, some of them belong to the NP-hard problems class. These algorithms are search-based enumeration techniques that enumerate implicitly the entire solutions space. In order to perform this enumeration efficiently, many improvements of the algorithms have been proposed so far, which focus on the structure of each problem. Unfortunately, in real life applications, many cases of unsolvable size are encountered (optimal solution cannot be found in a reasonable amount of time). However, it is possible to improve the techniques that control the parallelization of Branch-and-Bound algorithms with the objective of increasing the solvable size of problem instances. Since research on Branch-and-Bound algorithms began, these algorithms have been considered one of the suitable tools for parallel processing.

This thesis focuses on Branch-and-Bound skeletons. In Chapter I, we present the basic concepts and parallel tools we will use along this memory.

Several parallel tools for the Branch-and-Bound paradigm have been developed. PPBB (Portable Parallel Branch-and-Bound Library) proposes an implementation in the C programming language. PUBB (Parallelization Utility for Branch-and-Bound algorithms), BOB, PICO (An Object-Oriented Framework for Parallel Branch-and-Bound) and COIN (Common Optimization Interface for Optimization Research) are developed in C++ and in all cases a hierarchy of classes is provided and the user has to extend it to solve his/her particular problem. In Chapter II, an in depth study of the characteristics of each one of these tools is accomplished.

In chapter III, our MaLLBa::BnB proposal is presented. The aim of this thesis is to develop an object oriented framework (instantiated onto a C++ library of classes) that provides the necessary tools for the resolution of problems by means of the Branch-and-Bound paradigm. The library is generic and based on skeletons. The user is provided with the possibility of solving problems, both sequentially and in parallel form without having to modify its code. Also, the library provides a unique interface for each algorithmic skeleton which is platform independent. The participation of the final user is limited to choosing a resolution method and to characterize the algorithmic skeleton. The tool supplies a high level of programming, which facilitates the use of networks of computers under Linux, and also personal computers multiprocessor. This library allows the use of parallel machines of low cost by nonexpert users in parallelism, to solve problems in the area of optimization and resource decision making. Two parallel implementations are made using MPI (Message Passing Interface), the standard tool in Message Passing Programming. This choice gives an easy way to adapt the tool to different parallel platforms. Besides, a version of the tool is developed using OpenMP, in order to cover parallel platforms with Shared Memory. In chapter IV, the obtained computational results are presented. A careful experiments design is made and the obtained results are statistically studied to establish the performance of the library. The data structure used to implement the sequential and parallel schemes is shown in Appendix A.

In chapter V, we present the conclusions obtained about MaLLBa tool for Branch-and-Bound technique and some future research lines. Some examples of the implementation of MaLLBa::BnB for different Problems are shown in Appendix C.

A novelty of MaLLBa library is the integration of techniques, for which libraries have been developed separately but not combined. From the coordinated work between some members of the Parallel Computing Group of La Laguna University in the integration of the Divide-and-Conquer and Branch-and-Bound techniques, some papers were presented in International Conferences. A sequential approach to the integration of both skeletons titled "Resolución del Problema de la Mochila 0/1 usando esqueletos Divide-y-Vencerás y Ramificación-y-Acotación" was presented in 5th International Conference on Operations Research in La Havana, 2002, which will be published in the journal Investigación Operacional. A centralized parallel scheme of Branch-and-Bound skeleton is integrated with a parallel Divide-and-Conquer skeleton. With the title "Branch-and-Bound and Divide-and-Conquer Parallel Skeletons: an Integrated Approximation", it was presented in the International Symposium on Combinatorial Optimization in Paris, 2002. The improved scheme of the previous Master/Slave paradigm, that we call distributed scheme, was presented in the Eleventh Euromicro Conference on Parallel, Distributed and Network Based Processing, in Genoa, 2003. With the title "Parallel Skeletons for Divide-and-Conquer and Branch-and-Bound Techniques" was published in the proceeding by IEEE Computer Society.

A shared memory scheme using OpenMP was presented in Eighth International WorkShop on High-Level Parallel Programming Models and Supportive Environments, Nice 2003. The paper titled "A comparison between MPI and OpenMP Branch-and-Bound Skeletons", was published in the proceeding by IEEE Computer Society.

The mathematical formulation of the algorithms for the Branch-and-Bound skeletons were presented in Fifth International Conference on Parallel Processing and Applied Mathematics (PPAM2003) in Czestochowa, 2003. The paper titled "Parallel Branch-and-Bound Skeletons: Message Passing and Shared Memory implementations" will be published in the conference proceedings by Springer Verlag Series Lecture Notes in Computer Science.

An example of the complete use of MaLLBa::BnB library is presented in Parallel Computing 2003 (Parco2003), in Dresden 2003, with the title "MPI and OpenMP implementations of Branch-and-Bound Skeletons". Conference proceedings will be published by Elsevier Science.

A summary of these three International Conferences will be published under the title "Comparing MPI and OpenMP implementations of the 0-1 Knapsack Problem" in the Parallel and Distributed Computing Practices journal.

An analysis of the computational results using the tools CALL and LLAC was presented with the title "Complejidad Algorítmica: de la Teoría a la

Práctica" in "IX Jornadas de Enseñanza Universitaria de la Informática", Cádiz in 2003. This work is the result of the coordination with some members of the Parallel Computing Group of La Laguna, developers of both tools.

This developed work has been partially supported by three projects: The MaLLBa Project (TIC1999-0754-C03) titled "Geographically Distributed Environments: Combinatorial Optimization Library". The MaLLBa project brings together researchers from three Spanish institutions funded by CICYT (Comisión Interministerial de Ciencia y Tecnología). The project members include personnel from the "Departamento de Lenguajes y Ciencias de la Computación" (University of Málaga), "Departamento de Estadística, Investigación Operativa y Computación (University of La Laguna), and "Department de Llenguatges i Sistemes Informátics" (Polytechnic University of Catalonia), which coordinates the project.

The project "MAGOS: Parallel Library for Integration of Divide-and-Conquer and Branch-and-Bound Techniques and its application in the resolution of the Nesting Problem" is founded by the Government of the Canary Islands and its reference number is: PI-2001/60.

We have also received the support from the European Community-Access to Research Infrastructure action of the Improving Human Potential Programme contract No. HPRI-CT-1999-00026 (TRACS Program at EPCC-Edinburgh Parallel Computing Center). We have also used the computers of CEPBA (Centro Europeo de Paralelismo de Barcelona). Finally, the CIEMAT (Centro de Investigaciones Energéticas, Medioambientales y Tecnológicas) allowed us access to their computers.

As result of the coordination of these projects and the use of these computers some contributions to conferences were obtained. In 2000, "Algoritmos Exactos de Optimización Combinatoria: Una Aproximación Geográficamente Distribuida" was presented in "XI Jornadas de Paralelismo", in Granada. In 2001, "MALLBA: Towards a Combinatorial Optimization Library for Geographically Distributed Systems" was presented in "XII Jornadas de Paralelismo", in Valencia. Both of them appear in the proceedings of the conferences. Finally, "MALLBA: A Library of Skeletons for Combinatorial Optimisation" was presented in Euro-Par, Paderborn 2002. The proceedings of the conference were published by Springer-Verlag Series Lecture Notes in Computer Science.

At present, we are developing the TRACER project, titled "Advanced Optimization Techniques for Complex Problems", that joins researchers from five Spanish Universities. The project members include staff from the three Universities that developed MaLLBa plus another two Universities: the University of Extremadura and Carlos III University. The co-ordinated project

TRACER aims at developing the baseline research foundations of algorithms and advanced optimization techniques. This project has been supported by the EC (FEDER) and the Spanish “Ministerio de Ciencia y Tecnología” with the “I+D+I” (Research, Development and Innovation) contract number: TIC2002-04498-C05-05.

1 Introduction

1.1 DEFINITIONS

Combinatorial optimization is one of the branches of Mathematical Programming which has accomplished remarkable progress in last two decades. It is closely related to other disciplines such as Operations Research, Computer Science and Combinatorial Mathematics. Its applications are found in a wide spectrum of fields; Engineering (Electrical, Electronics, Civil, Mechanical, Chemical and System), as well as Economics, Social Sciences, Biology, Telecommunications and others. The importance of combinatorial optimization is being enhanced not only by its theoretical progress but also by the recent innovation of computer technology, because these enable us to solve large scale problems more efficiently and easily. This thesis tackles Combinatorial Optimization from the point of view of Computer Science, not from the Operations Research point of view. We are specifically interested in the resolution of combinatorial optimization problems.

A `combinatorial optimization problem` asks to find a solution maximizing (or minimizing) an objective function over a combinatorial set. Among typical examples are optimization problems related to graphs and networks, scheduling problems to find optimal orders of n jobs, problems to select optimal subsets of given finite sets, and problems to find optimal locations of n objects.

We use the notation introduced by Ibaraki in [46], where, formally, a combinatorial optimization problem is defined as a t-uple $\Pi = (I, S, f, g)$ satisfying:

- I is the set of instances of Π. If $x \in I$ we say that x is an *instance* (or an input) of Π.
- Given an instance $x \in I$, $S(x)$ denotes the *set of feasible solutions* of x.
- For any instance $x \in I$ and any feasible solution $\sigma \in S(x)$, $f(x, \sigma)$ represents a real value, the measure (or cost or fitness) of σ with respect to Π and x. The function f is called the *objective function.*
- $g \in \{\max, \min\}$. The goal of Π is to find a feasible solution that optimizes f according to g: given an input $x \in I$, determine an *optimal solution* $\sigma' \in S(x)$ such that $f(x, \sigma') = g\{f(x, \sigma) \mid \sigma \in S(x)\}$.

The word `algorithm` takes on a special significance in computer science, where it has come to refer to a precise method usable by a computer for the solution of a problem. This is what makes the notion of an algorithm different from words such as process, technique or method. An algorithm [52] is composed of a finite set of steps, each of which may require one or more operations. After a finite number of operations, it terminates. The possibility of a computer carrying out these operations requires that certain constraints be placed on the type of operations an algorithm can include. For example, each operation must be definite, meaning that it must be perfectly clear what should be done. Another important property each operation should have is that it be effective; each step must be such that it can, at least in principle, be done by a person using pencil and paper in a finite amount of time. An algorithm produces one or more outputs and may have zero or more inputs which are externally supplied.

The main object of combinatorial optimization is to find better algorithms to solve problems like this. Often one has to find the best element of some finite set of feasible solutions. This set is not listed explicitly but implicitly depends on the structure of the problem. Therefore an algorithm must exploit this structure. In practice, it is often required to solve a given problem even if it is a theoretically difficult one. Within the limit of its inherent complexity, therefore, is the desire to develop an algorithm of the maximum efficiency by exploiting all the possible problem structures useful in generating the computation. In general, to solve a combinatorial optimization problem, all possibilities have to be explored in a search tree. The simplest technique consists of the enumeration of all possible solutions. This way of working is called Brute Force. However, other approaches to deal with the resolution of problems such as: Divide-and-Conquer, Branch-and-Bound and Dynamic Programming have been developed.

The Divide-and-Conquer algorithm can be defined as follows [44]: Given a function to compute on n inputs, this strategy suggests splitting the inputs into k subsets, $1 < k \leq n$ yielding k subproblems. These subproblems must be solved and then a method must be found to combine sub-solutions into a solution as a whole. If the subproblems are still relatively large, then the Divide-and-Conquer strategy may be reapplied. Often the subproblems resulting from a Divide-and-Conquer design are of the same type as the original problem. Now smaller and smaller subproblems of the same kind are generated, eventually producing subproblems that are small enough to be solved without splitting.

However, the explicit enumeration of all possible solutions is not computationally feasible even for problems of moderate size. This means that it is imperative to equip the algorithms with a capability to identify those solutions not yielding optimal solutions, and to exclude them from the explicit enumeration. Different types of enumerative algorithms result depending upon how such exclusion is carried out, with which the computational burden can be drastically reduced. Dynamic Programming is an algorithmic technique that can be used when the solution to a problem is viewed as the result of a sequence of decisions. Dynamic Programming often drastically reduces the amount of enumerations by avoiding the enumeration of some decision sequences that cannot possibly be optimal. The idea of Dynamic Programming algorithms is to break a large problem down (if possible) into incremental steps so that, at any given stage, optimal solutions are known to sub-problems. When the technique is applicable, this condition can be extended without having to alter previously computed optimal solutions to subproblems. Eventually the condition applies to all of the data and, if the formulation is correct, this together with the fact that nothing remains untreated gives the desired answer to the problem as a whole. In conclusion, a feasible solution is taken into account as a sequence of decisions occurring in stages and the total cost is the sum of the cost of individual decisions [10].

The Branch-and-Bound method tries to reduce the number of feasible solutions by systematic exploration of the solution area. This method divides the solution area step by step (branch steps) and computes a bound for the value of the objective function for all elements of the partition (bound step). The computation of the bound makes it possible to cut off parts of the solution area. The quality of the computed bound has a strong influence on the effectiveness of the whole method [46]. In the following section we study this technique in deep.

It has been observed that many algorithm techniques can be characterized and classified by their adherence to one or more of a number of generic patterns of computation and interaction. Skeletal programming proposes that these patterns be abstracted and provided as a tool for programmers, with specifications which transcend architectural variations, but with implementations which recognize these, to enhance performance. In this way it promises to address many of the traditional issues within the software engineering process such as:

- it will simplify programming by raising the level of abstraction,
- it will enhance portability and re-use by absolving the programmer of responsibility for detailed realization of the underlying patterns,
- it will improve performance by providing access to carefully optimized, architecture specific implementations of the patterns,
- it will offer scope for static and dynamic optimization, by explicitly documenting information on algorithmic structure which would be impossible to extract from equivalent unstructured programs.

To document these common techniques used by functional programming the concepts of `pattern` and `skeleton` must be defined. In its simplest form, a pattern is a three-part rule that expresses a relation to a programming context, the set of competing concerns, or forces, that occurs repeatedly in that context, and a stereotypical software configuration that resolves these forces favorably. The context denotes the state of a system: the components already present and global constraints on the result. The set of forces constitutes a problem to be solved, and the stereotypical configuration solves the problem. In practice, a pattern also usually explains why this solution suffices and describes how to implement the solution in code. The value of a pattern lies largely in its ability to explain. Producing a pattern requires that the author state explicitly the context in which a technique applies and the design issues involved in implementing a solution. These elements of a pattern provide significant benefits to the reader, who can learn both the technique and explore when and how to use it. The author of a pattern also benefits, as the pattern form encourages a level of explicitness not always present in other written forms.

1.2 Branch-and-Bound TECHNIQUE

Branch-and-Bound algorithms are general methods applied to various combinatorial optimization problems that belong to the NP-hard problems class. In this section we modify the definition of combinatorial optimization problems given in the previous section to introduce the concept of `subproblem`. This allows us to tackle the resolution of these problems by means of the Branch-and-Bound technique.

A *subproblem* Π_i is a tuple $\Pi_i = (I, S_i, f, g)$ where $S_i(x)$ is a subset of the underlying space I.

Given a *subproblem* Π_i, it can be broken down into $\Pi_{i1}, \Pi_{i2}, ..., \Pi_{ik}$ by a branching operation where $S_i = \bigcup_{j=1}^{k} S_{ij}$. Thus any feasible solutions $\sigma \in S_i$ belong to some S_{ij} and conversely any $\sigma \in S_{ij}$ belong to S_i. Let $\mathcal{Q}$ denote the set of *subproblems* currently generated. A *subproblem* $\Pi_i \in \mathcal{Q}$ that is neither broken down nor tested yet is called *live*. The set of *live subproblems* are denoted by $\mathcal{L}$. For each tested subproblem in $\mathcal{Q}$ its *lower bound* and *upper bound* are computed. The greatest lower bound so far obtained is called the *best solution value* or *incumbent value* and is denoted by bs. The solution bs is called the *best solution* and is stored in $\mathcal{T}$. The algorithm proceeds by repeating the test of lives subproblems. The selection of a live subproblem for the next test is performed by a *search function* s, such that, $s(\mathcal{L}) = \mathcal{L}$.

To apply the Branch-and-Bound method to a combinatorial optimization (i.e maximization) problem, we follows the following steps:

Step 1 (initialization): $\mathcal{L} := \{\Pi_0\}$, $\mathcal{Q} := \{\Pi_0\}$, $bs := -\infty$ and $\mathcal{T} := \emptyset$ (empty set).

Step 2 (search): Go to Step 9 if $\mathcal{L} = \emptyset$. Otherwise go to Step 3 after selecting $\Pi_i := s(\mathcal{L})$ for the test.

Step 3 (update): If $lower\_bound(\Pi_i) > bs$, let $bs := lower\_bound(\Pi_i)$ and $\mathcal{T} := \{\sigma\}$, where σ is a feasible solution of Π_i carrying out $f(x, \sigma) = lower\_bound(\Pi_i)$. Go to Step 4.

Step 4: Go to Step 8 if $\Pi_i \in G$ (the set of partial problems Π_i solved in the course of computing the $upper\_bound(\Pi_i)$). Otherwise go to Step 5.

Step 5 (upper bound test): Go to Step 8 if $upper\_bound(\Pi_i) \leq bs$. Otherwise go to Step 6.

Step 6: Go to Step 8 if there exists a $\Pi_k (\neq \Pi_i) \in \mathcal{Q}$ such that $f(\Pi_k) \geq f(\Pi_i)$. Otherwise go to Step 7.

Step 7 (branch): Break down Π_i into $\Pi_{i1}, \Pi_{i2}, ..., \Pi_{ik}$ and let $\mathcal{L} := \mathcal{L} \cup \{\Pi_{i1}, \Pi_{i2}, ..., \Pi_{ik}\}$ - $\{\Pi_i\}$, $\mathcal{Q} := \mathcal{Q} \cup \{\Pi_{i1}, \Pi_{i2}, ..., \Pi_{ik}\}$. Return to Step 2.

Step 8 (terminate Π_i): Return to Step 2 after letting $\mathcal{L} := \mathcal{L}$ - Π_i.

Step 9 (termination): Stop. bs is equal to $f(\Pi_0)$, and $\mathcal{T}$ stores an optimal solution of Π_0, if $bs > -\infty$. If $bs = -\infty$, Π_0 is unfeasible.

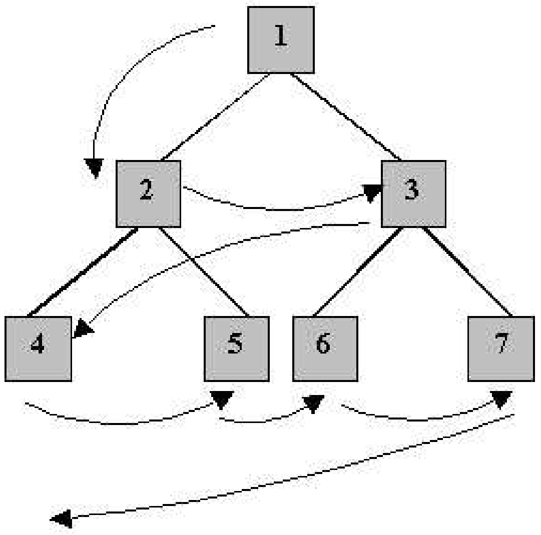


Fig. 1.1 Breadth-First Search

A Branch-and-Bound algorithm always converges in finite steps regardless of the search function s. However, its efficiency and the required storage space are highly dependent on the search function s. We introduce the following important search functions used in practice and their characteristics:

- *Breadth-first search* expands the search space on a level by level basis. A priority function that always chooses to expand the problem with the smallest depth value will lead to a breadth-first expansion of the search space. An advantage of a breadth-first search is that it guarantees to find the solution of minimum depth in the tree. In a problem where solutions may be found at different depths this may be one of the optimization criteria.

- *Depth-first search* attempts to generate a solution by performing search levels down the tree. After expanding a given problem, the algorithm attempts to expand one of its newly generated offsprings. One way of

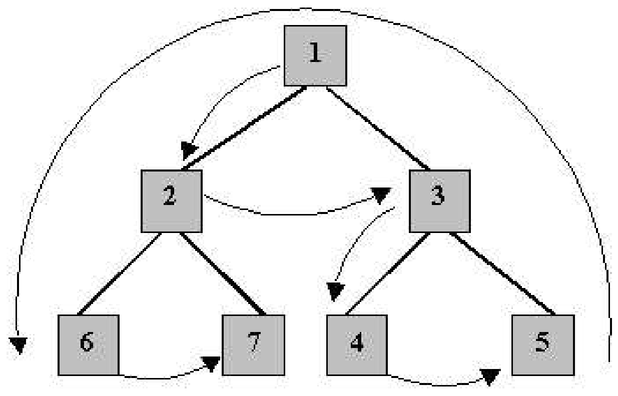


Fig. 1.2 Depth-first search

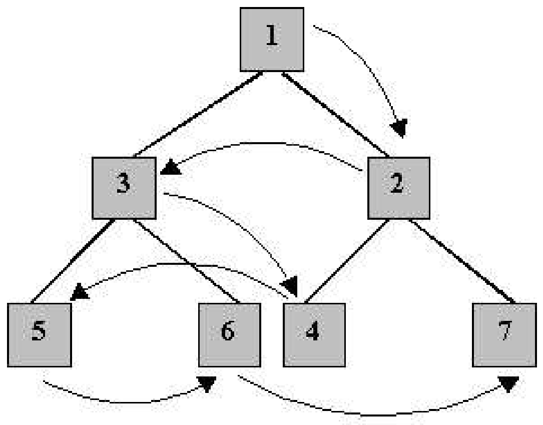


Fig. 1.3 Best-First Search

implementing this is to always select the problem-state of greatest depth in the tree for expansion. An advantage of a depth-first search is that it often has the smallest storage requirements of any search strategy. This strategy also attempts to generate an initial solution quickly so that this can be used for pruning sections of the tree.

- *Best-first search* orders the problems for expansion according to their bound values. Problems with the best bound values are chosen to be expanded before others. An advantage of the best-first approach is that, on a single processor, provided the bound values are unique, it will never expand a problem-state which has not been expanded by other methods.

In general terms, it can be said that the breadth-first searches explore the nodes in the same order in which they are created, it means that they use a queue (FIFO list: first-in-first-out) to store the nodes that have been generated but have not been examined yet (figure 1.1). The depth-first searches explore the nodes in inverse order to that of their creation, employing a stack (LIFO list: last-in-first-out) to store the nodes that are being generated but have not been examined yet (see figure 1.2). The best-first search explore the nodes according to their best bound values (see figure 1.3). In figures 1.1, 1.2 and 1.3 the numbers of the nodes represent the order of creation of the subproblems and the arrows indicate the way.

1.3 A CASE STUDY: KNAPSACK PROBLEM

In this section we introduce the definition of the 0/1 Knapsack Problem that will be used to study the functioning of the different libraries analyzed in chapter II and the library we propose in chapter III.

The 0-1 Knapsack Problem has been studied extensively during the past decades. Several exact algorithms for its resolution can be found in the literature and for this reason appears in many real applications with practical importance. This problem is considered NP-hard.

Consider the classical 0-1 Knapsack Problem where a subset of N given items has to be introduced in a knapsack of capacity C. Each item has a profit p_i and a weight w_i and the problem is to select a subset of items whose total weight does not exceed C and whose total profit is a maximum. Assume that all input data are positive integers. Introducing the binary decision variables x_i, with $x_i = 1$ if item i is selected, and $x_i = 0$ otherwise, the problem can be formulated as follows:

$$\begin{array}{rc} & max \;\; \sum_{i=1}^{N} p_i x_i \\ subject \;\; to: & \sum_{i=1}^{N} w_i x_i \leq C \\ & x_i \in \{0,1\}, i \in \{1,...,N\} \end{array}$$

Exact algorithms for knapsack problems are mainly based on two approaches: Branch-and-Bound and Dynamic Programming. This work focuses on the Branch-and-Bound (MaLLBa::BnB) paradigm, as mentioned in the previous section. This technique determines the optimum-cost solution of a problem through a selective exploration of a solution tree. The internal tree nodes correspond to different states of the solution search and the "leaves" correspond to feasible solutions. The first MaLLBa::BnB algorithm to solve the Knapsack Problem appeared in the seventies. Among others, we should mention the algorithms by Horowitz and Shani [43], and Martello and Toth [68]. These algorithms are based on a depth-first enumeration, in order to limit the search space.

The Branch-and-Bound algorithm described by Martello and Toth [69] to solve this problem has been implemented using the MaLLBa::BnB skeleton (see next chapter). The algorithm requires the elements to be ordered in ascending order according to the weight/profit relation given by:

$$\frac{p_i}{w_i} \geq \frac{p_{i+1}}{w_{i+1}} \quad (i = 1, ..., N-1)$$

The bounds used in the implementation are defined as follows:

The *lower_bound* is calculated by including objects in the knapsack until the maximum capacity is reached:

$$\sum_{i=k}^{s} w_i x_i \leq C; \quad x_i \in \{0,1\}, i, k, s \in \{1, ..., N\}$$

$$lower\_bound = \textstyle\sum_{i=k}^{s} p_i x_i$$

In a similar way the *upper_bound* is calculated, but in this case a portion of the the last object considered is included to fit the capacity.

$$C_r = \sum_{i=k}^{s} w_i x_i \leq C; \qquad x_i \in \{0,1\}, i, k, s \in \{1, ..., N\}$$

$$upper\_bound = \textstyle\sum_{i=k}^{s} p_i x_i + \frac{p_{s+1} \times (C - C_r)}{w_{s+1}}$$

As an example [69] we consider a knapsack with a capacity $C = 102$ and the number of objects $N = 8$ with the following weights and benefits:

$$\begin{aligned} p_k &= \{15, 100, 90, 60, 40, 15, 10, 1\} \\ w_k &= \{2, 20, 20, 30, 40, 30, 60, 10\} \end{aligned}$$

Figure 1.4 shows the trace of the algorithm for the resolution of the Knapsack Problem in this example. To solve the problem by Branch-and-Bound a tree in whose root the value of none of the x_i is fixed, and where in each successive level the value of a farther variable is determined by numerical order of the variables has to be explored. Each node that is explored produces two farther nodes, depending on whether the following object in the knapsack is introduced or not. If a node is generated, the upper bound and a lower bound of the solution value are calculated, which can be obtained completing the partially specified capacity, and these bounds are used to cut useless branches and to guide the exploration of the tree.

The optimum solution vector is $x = (1, 1, 1, 1, 0, 1, 0, 0)$ and the value of the objective function is $z = 280$.

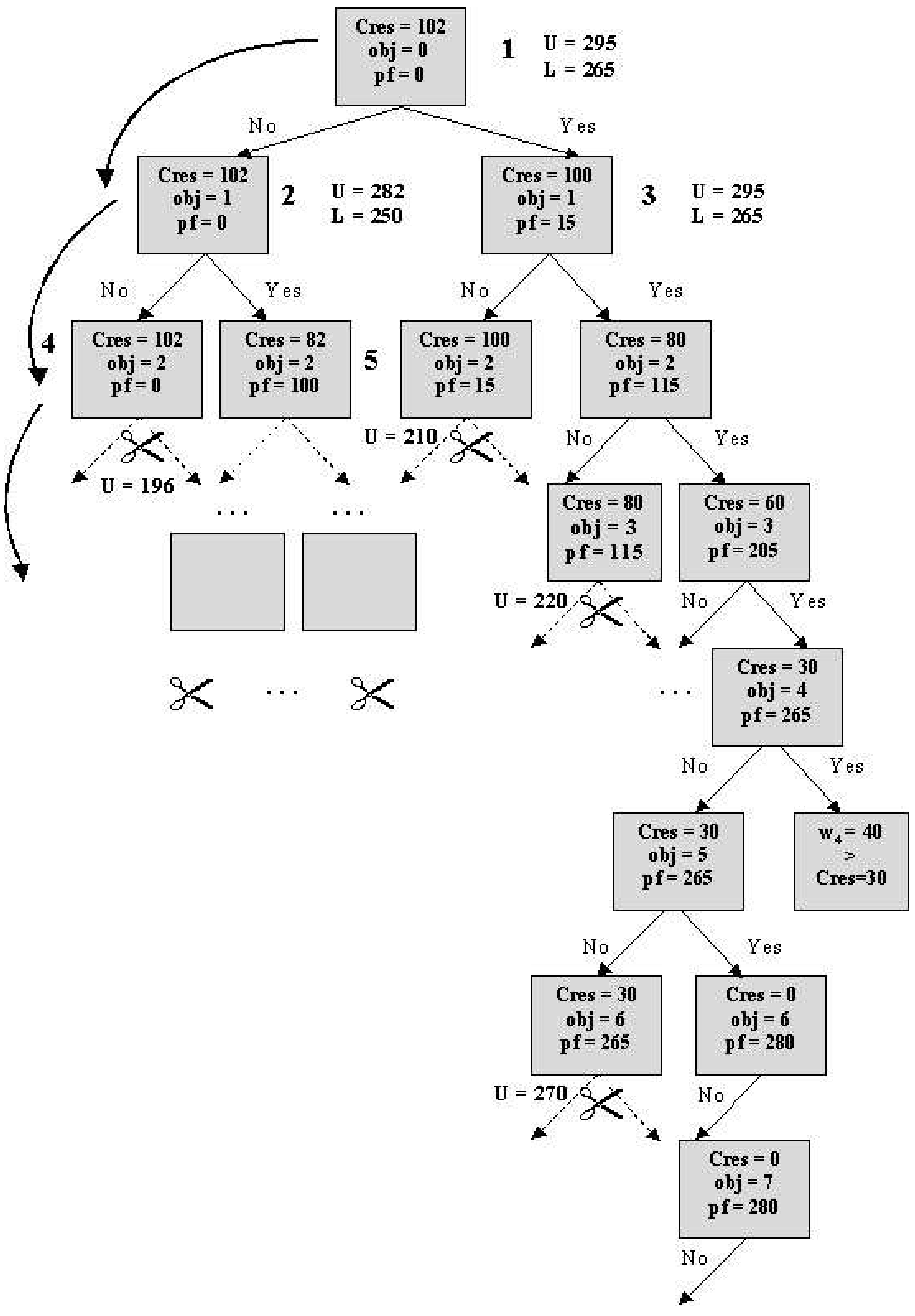


Fig. 1.4 Search Tree of an Example of Knapsack Problem

1.4 PARALLEL PROGRAMMING DEFINITIONS

In this section we introduce the definitions related to parallel programming that will be used in following chapters. It seems that whatever the computational speed of current processors, there will be applications that require still more computational power. One way of increasing the computational speed, a way that has been considered for many years, is by using multiple processors operating together on a single problem. The overall problem is split into parts, each of which is performed by a separate parallel processor. Writing programs for this form of computational method is known as `parallel programming` [2]. The computing platform, a `parallel computer`, could be a specially designed computer system containing multiple processors or several independent computers interconnected in some way. The approach should provide a significant increase in performance. The idea is that n computers could provide up to n times the computational speed of a single computer, no matter what the current speed of the computer, with the expectation that the problem would be completed in $1/n_{th}$ of the time. Of course, this is an ideal situation that is rarely achieved in practice. Problems often cannot be divided perfectly into independent parts and interaction is necessary between the parts, both for data transfer and synchronization of computations. However, substantial improvement may be achieved, depending upon the problem and the amount of parallelism in the problem. Apart from obtaining the potential for increased speed on an existing problem, the use of multiple computers/processors often allows a larger or a more precise solution of the problem to be solved in a reasonable amount of time. A related factor is that multiple computers often have more total RAM memory than a single computer, enabling problems that require larger amounts of RAM memory to be tackled.

Parallel programming requires a suitable computing platform, which is described as either a single computer with multiple internal processors or multiple interconnected computers. A natural way to extend the single processor model is to have multiple processors connected to multiple memory modules, such that each processor can access any memory module in a so-called `shared memory configuration`, as shown in figure 1.5. The connection between the processors and memory is through some form of interconnection network. A shared memory multiprocessor system employs a single address space, which means that each location in the whole main memory system has a unique address and this address is used by each processor to access the location. Programming a shared memory multiprocessor involves having executable code stored in the memory for each processor to execute. The data for each program will also be stored in the shared memory and hence each program could access all the data if needed.

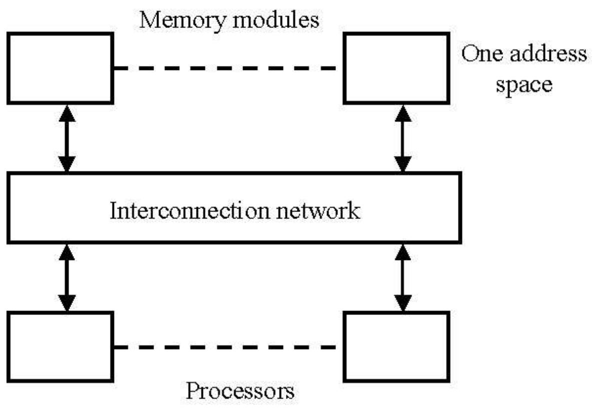


Fig. 1.5 Traditional Shared Memory Multiprocessor Model

The shared memory multiprocessor is a specially designed computer system. An alternative form of multiprocessor system can be created by connecting complete computers through an interconnection network. Each computer consists of a processor and local memory that is not accessible by other processors. In a multicomputer, the memory is distributed among the computers and each computer has its own address space. A processor can only access a location in its own memory. The interconnection network is provided for processors to send messages to other processors. These messages can include data that other processors may require for their computations. Such multiprocessor systems are usually called `message passing multiprocessors` or simply `multicomputers`; as shown in figure 1.6. Message-passing systems are characterized by their interconnection network topology, which describes how processors are connected. The most common topologies are the ring, the tree, the mesh, and the hypercube [42].

We make a further distinction between distributed computer systems that are homogeneous and those that are heterogeneous. This distinction is useful only for multicomputers. In a `homogeneous multicomputer`, there is essentially only a single interconnection network that uses the same technology everywhere. Likewise, all processors are the same and generally have access to the same amount of private memory. In contrast, a `heterogeneous`

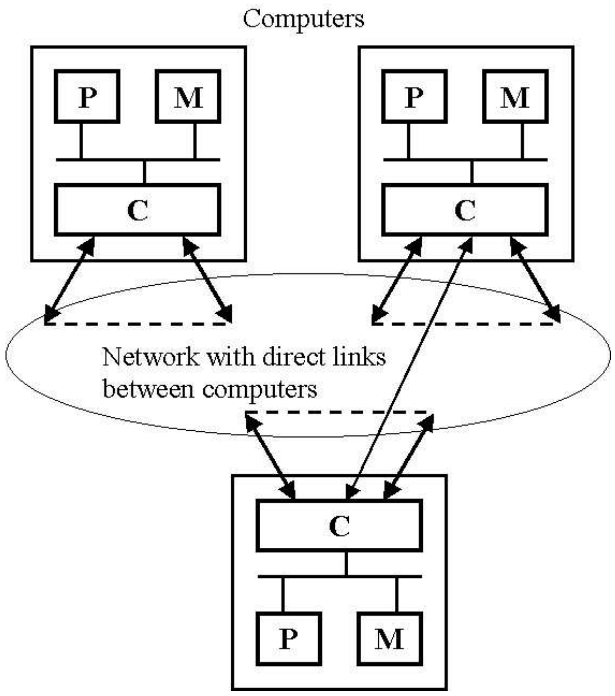


Fig. 1.6 Message-Passing Multicomputer Model

multicomputer system may contain a variety of different, independent computers, which in turn are connected through different networks.

Computer architecture strongly influences the design of parallel Branch-and-Bound algorithms and parallelism at the hardware level must be distinguished from the parallelism at the software level. To describe parallelism at the hardware level we need to explain some basic definitions [39, 94] as follows.

Key issues in network design are the network bandwidth, network latency, and the cost indicated by the number of links in the network. The bandwidth is the number of bits that can be transmitted in unit time, given as bits/sec. The network latency is the time taken for message transfer through the network. The communication latency is the total time taken to send the message, including the software overhead and interface delays. Synchronization refers to the presence or absence of a global clock used to synchronize operations among processors. When there is only one clock, we speak of a synchronous `system`, while in the presence of several clocks, typically one per processor, the system is called `asynchronous`. Communication between processors can be asynchronous or synchronous. The characteristic feature of `asynchronous communication` is that a sender continues working immediately after it has submitted its message for transmission. This means that the message is stored in a local buffer at the sending host, or otherwise at the first communication server. With `synchronous communication`, the sender is blocked until its message is stored in a local buffer at the receiving host, or actually delivered to the receiver. The strongest form of synchronous communication is when the sender is blocked until the receiver has processed the message.

`The grain` indicates the amount of data each processor in the system can handle. In fine-grained systems, each processor can handle only a small amount of data, corresponding to scalar or small vector operations. At the other extreme, coarse-grained systems are characterized by the possibility of simultaneous treatment of large amounts of data.

Considering the number of processors, massive parallel systems are made up of a large number of processors (in the order of thousands). Fine-grained systems are usually massively parallel, while coarse-grained ones generally have less processors, say in the order of tens (but the situation is changing rapidly, and now some coarse-grained systems with more than one thousand processors exist). Note that tightly coupled shared-memory systems are generally restricted to a small number of processors, usually not more than twenty, due to the difficulty of implementing simultaneous access to a common memory without provoking bottlenecks. With respect to the total number of generated subproblems, this measure may be used in all types of algorithms, but it may not represent a fair evaluation of the total amount of work performed by the algorithm for two main reasons. First, the work performed may vary significantly from one subproblem to another. Second, it does not take into account additional work introduced by parallelism, such as the sharing of work units among processes and the communication of an upper bound. Another problem related to this measure is that it may vary significantly with different executions when the algorithm has a nondeterministic behavior, particularly if it is an asynchronous one.

The following definitions will be used to analyze the computational results.

The `speedup measure` attempts to evaluate the improvement in time performance when more than one processor is used. Let $T(p)$ denote the time to solve a given problem on $p \geq 1$ processors. The `speedup` and the `efficiency` are then defined as $S(p) = T(1)/T(p)$ and $E(p) = S(p)/p$, respectively. The major difficulty with this definition consists of determining which algorithms should be used to measure the times, and how these should be measured. One definition for $T(1)$ is the time required by the best sequential algorithm. To evaluate the speedup of a given parallel algorithm based on this definition, time is measured with respect to the same parallel architecture for both the sequential and the parallel algorithms. However, the best sequential algorithm for all instances may not be known, as is often the case with problems solved by Branch-and-Bound algorithms. One may then use as $T(1)$ the time obtained by a good sequential algorithm, or the time required by the parallel algorithm running on one processor. These are the methods of choice when studying the speedup performance of a parallel Branch-and-Bound algorithm. In particular, a good sequential Branch-and-Bound algorithm can be one in which the branching and bounding operations, as well as the elimination rules, are defined in similar way to the parallel algorithm.

So far, we have divided a problem into a fixed number of processes that are to be executed in parallel. Each process performs a known amount of work. In addition, it is assumed that the processes are simply distributed among the available processors without any discussion on the effects of the types of processors and their speeds. However, it may be that some processors will complete their tasks before others and become idle because the work is unevenly distributed or some processors operate faster than others. Ideally, we want all the processors to be operating continuously on tasks that would lead to the minimum execution time. Achieving this goal by spreading the tasks evenly across the processors is called `load balancing`.

In the wider world of software engineering the concept of `skeleton` introduced by Cole in [14] appears. The gist of the idea is that useful patterns of parallel computation and interaction can be packaged up as 'framework/second order/template' constructs (i.e. parameterized by other pieces of code), perhaps presented without reference to explicit parallelism, perhaps not. Such constructs are known as `skeletons` because they have structure but lack detail.

The Longman dictionary of contemporary English defines skeleton as: Something forming a structure on which more is built or added: *the steel skeleton of a new skyscraper | I've written the skeleton of my report, but I have to fill in the details.*

To apply the parallelism of skeletons different tasks are performed:

- Map: In a map technique a vector data item appearing in the input stream of a map is split into components and each component is submitted to one of the worker processes computing a function. Each worker process applies the same function to the data items received and delivers the result into the output stream. The emitter process takes care of (sub)task-to-worker scheduling, while the collector process takes care of rebuilding the vector with the output data items and of delivering the new vector into the output data stream. The resulting process network looks like figure 1.7.

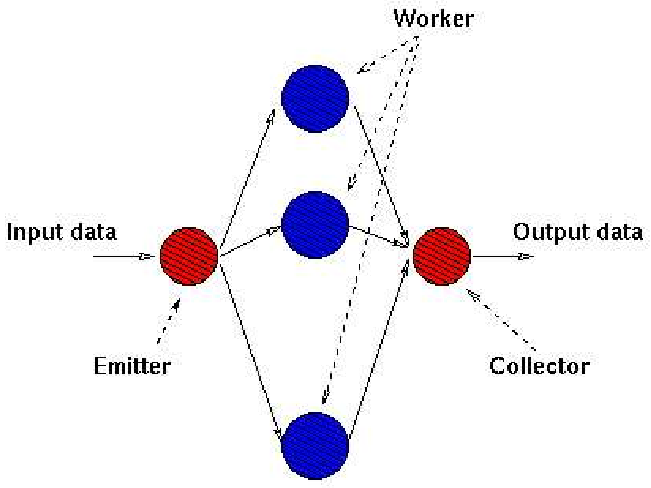


Fig. 1.7 Map Technique

- Farm: In a farm technique a sequence of data appearing into the input stream of a farm is submitted to a set of worker processes. Each worker process applies the same function to the data items received and delivers the result into the output stream. The emitter process takes care of task-to-worker scheduling, while the collector process takes care of reordering the output data items with respect to the input ordering and of delivering them into the output data stream. The resulting process network looks like figure 1.8.

- Pipeline: In the pipeline technique, a sequence of data appearing onto the input stream of a pipe is submitted to the first pipeline stage. The output of this stage is submitted to the second stage and so on and so on until the output of the (n-1) stage is submitted to the last stage. Eventually, the last stage delivers its own output onto the pipeline output channel. The resulting process network looks like the figure 1.9.

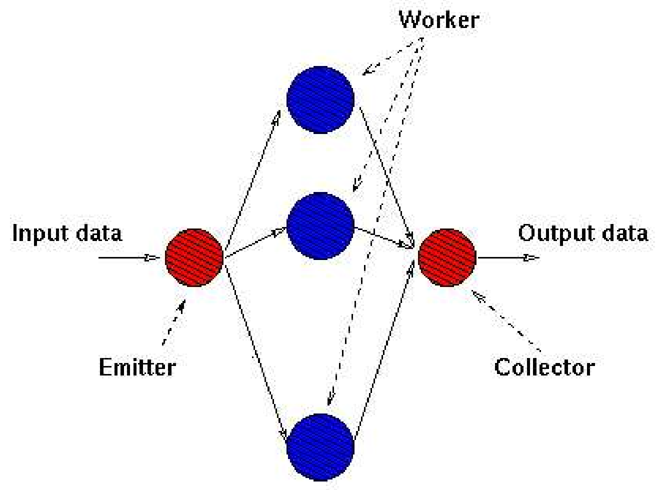


Fig. 1.8 Farm Technique

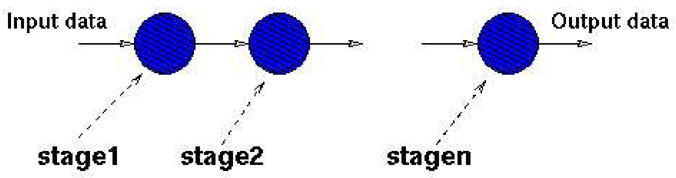


Fig. 1.9 Pipeline Technique

In our proposal we implement a parallel skeleton based on the farm technique, also called master-slave. This parallel implementation will be made using MPI, the standard tool in message passing programming, presented in next section. This choice provides an easy way to adapt the tool to different parallel platforms.

1.5 MPI TOOL

MPI stands for Message Passing Interface [88]. The objective of the Message Passing Interface (MPI) effort is to develop a widely used standard for writing message-passing programs. As such the interface should establish a practical, portable, efficient, and flexible standard for message passing. Message passing is a paradigm used widely on certain classes of parallel machines, especially those with distributed memory.

MPI provides a large number of routines for message passing and associated operations. However, programs can be written using a very small subset of the available functions. We will mention just the fundamental functions. All MPI routines start with the prefix `MPI_` and the next letter is capitalized. Generally, routines return information indicating the success or failure of the call.

Before any MPI function call, the code must be initialized with `MPI_Init()`, and after all MPI function calls, the code must be terminated with `MPI_Finalize()`. Initially, all processes are enrolled in a universe called `MPI_COMM_WORLD`, and each process is given a unique rank, a number from 0 to $n - 1$, where there are n processes. In MPI terminology, `MPI_COMM_WORLD` is a communicator that defines the scope of a communication operation, and processes have ranks associated with the communicator. A communicator is a communication domain that defines a set of processes that are allowed to communicate between themselves.

MPI assumes communication takes place within a known group of processes. Each group is assigned an identifier. Each process within a group is also assigned a local identifier. A (groupID, processID) pair therefore uniquely identifies the source or destination of a message, and is used instead of a transport-level address. There may be several, possibly overlapping groups of processes involved in a computation and they may all be executing at the same time.

At the core of MPI are messaging primitives to support most of the forms of communication. The routines we will need are classified into two groups:

Blocking Routines. In MPI, blocking routines (send and receive) return when they are locally complete. The local completion condition for a blocking send routine is that the location used to hold the message can be used again or altered without affecting the message being sent. A blocking send will send the message and return. This does not mean that the message has been received, just that the process is free to move on without adversely affecting the message. Essentially the source process is blocked for the minimum time required to access the data. A blocking receive routine will also return when it is locally complete, which in this case means that the message has been received by the destination location and the destination location can be read.

```
MPI_Send (buf,       // address of send buffer
          count,     // number of items to send
          datatype,  // datatype of each item
          dest,      // rank of destination process
          tag,       // message tag
          comm       // communicator
         )


MPI_Recv (buf,       // address of receive buffer
          count,     // maximum number of items to receive
          datatype,  // datatype of each item
          dest,      // rank of source process
          tag,       // message tag
          comm,      // communicator
          status     // status after operation
         )
```

The following routine is used to test for a message in the nonblocking case:

```
MPI_Probe (int source,         // source process rank
           int tag,            // message tag
           MPI_Comm comm,      // communicator
           MPI_Status *status  // status returned
          )
```

Nonblocking Routines. A nonblocking routine returns immediately; that is, it allows the next statement to execute, whether or not the routine is locally complete. The nonblocking send, `MPI_Isend()`, where `I` refers to the word immediate, will return even before the source location is safe to be altered. The nonblocking receive, `MPI_Irecv()`, will return even if there is no message to accept. The formats are

```
MPI_Isend (buf, count, datatype, dest, tag, comm, request)
MPI_Irecv (buf, count, datatype, source, tag, comm, request)
```

These routines need to know whether the particular operation has been completed, which is determined by accessing the `request` parameter. Nonblocking routines provide the ability to overlap communication and computation, which is essential when communication delays are high.

The following routine is used to test for a message in the blocking case:

```
MPI_Probe (int source,        // source process rank
           int tag,           // message tag
           MPI_Comm comm,     // communicator
           int *flag,         // true if there is a message (returned)
           MPI_Comm *status   // status returned
          )
```

The semantics of MPI communication primitives are not always straightforward, and different primitives can sometimes be interchanged without affecting the correctness of a program. More on MPI can be found in [90, 94].

1.6 OPENMP TOOL

OpenMP C/C++ Application Program Interface (API) [72] is a collection of compiler directives, library functions, and environment variables that can be used to specify shared-memory parallelism in C and C++ programs. The goal of this specification is to provide a model for parallel programming that allows a program to be portable across shared-memory architectures from different vendors.

Before describing the functioning of OpenMP, we will define the following terms that are used in this section:

`Thread.` An execution entity having a serial flow of control, a set of private variables and access to shared variables.

`Team.` One or more threads cooperating in the execution of a construct.

`Directive.` A C or C++ `pragma` followed by the `omp` identifier, other text, and a new line. The directive specifies program behavior.

`Parallel region.` Region of a program that is to be executed by multiple threads in parallel.

`Master thread.` The thread that creates a team when a parallel region is entered.

`Private.` A private variable names a storage block that is unique to the thread making the reference.

`Shared.` A shared variable names a single storage block. All threads in a team that access this variable will access this single block of storage.

`Barrier.` A synchronization point that must be reached by all threads in a team. Each thread waits until all threads in the team arrive at this point.

There are explicit barriers identified by directives and implicit barriers created by the implementation.

A program written with the OpenMP C/C++ API begins execution as a single thread of execution called the `master thread`. The master thread executes in a serial region until the first parallel construct is encountered. In the OpenMP, the `parallel` directive constitutes a parallel construct. When a parallel construct is encountered, the master thread creates a team of threads, and the master becomes master of the team. Each thread in the team executes the statements in the dynamic extend of a parallel region, except for the work-sharing constructs. Work-sharing constructs must be encountered by all threads in the team in the same order, and the statements within the associated structured block are executed by one or more of the threads.

If a thread modifies a shared object, it affects not only its own execution environment, but also those of the other threads in the program. The modification is guaranteed to be complete, from the point of view of one of the other threads, at the next sequence point, only if the object is declared to be volatile. Otherwise, the modification is guaranteed to be complete after the first modifying thread, and then (or concurrently) the other threads.

Upon completion of the parallel construct, the threads in the team synchronize at an implicit barrier, and only the master thread continues execution.

Directives are based on `#pragma` directives defined in the C and C++ standards. The syntax of an OpenMP directive is specified as follows:

```
#pragma omp directive-name [clause[ [,] clause]...] new-line
```

Each directive starts with `#pragma omp` to reduce the potential for conflict with other pragma directives with the same names. The following directive defines a parallel region:

```
#pragma omp parallel [clause[ [,] clause]...] new-line
    structured block
```

Inside a parallel region, variables can be either shared or private, so the previous clause can be : $private\ (variable - list)$ or $shared\ (variable - list)$.

Another clause is the $num\_threads\ (integer - expression)$ which determines the number of threads that are requested. However, subsequent parallel regions are not affected by it.

A work-sharing construct distributes the execution of the associated statement among the members of the team that encounter it. The work-sharing directives do not launch new threads. The sequence of work-sharing constructs must be the same for every thread in a team. Of all OpenMP defined work-sharing constructs we only describe the `for` directive here. It identifies an iterative work-sharing construct that specifies that the iterations of the associated loop will be executed in parallel. The iterations of the `for` loop are distributed across threads that already exist in the team executing the parallel construct to which it binds. The syntax of the `for` construct is:

```
    #pragma omp for [clause[ [,] clause]...] new-line
        for-loop

being the canonical shape of {\tt for} loop:
        for (init-expr; var logical-operation b; incr-expr)
```

We have only described the essential concepts we will use in chapter III, more information about OpenMP can be found at [72].

2
Related Works

Several tools of parallel skeleton implementations for Branch-and-Bound algorithms have been developed since 1995. This chapter is a survey of parallel Branch-and-Bound tools, most of which can be found by searching in Internet. Each section of this chapter is dedicated to the description of one tool. The sections are ordered in chronological form.

In section 2.1 the PPBB (Portable Parallel Branch-and-Bound Library) is presented. PPBB was designed, implemented, and tested at the University of Paderborn in 1996. This library is written in C and designed to run on distributed memory multicomputer architectures using the given message passing functions, specifically in PVM. The first version of the library was tested on Transputers.

Section 2.2 is dedicated to PUBB (Parallelization Utility for Branch-and--Bound algorithms). The PUBB utility was developed at the Science University of Tokyo, Japan. The first version of PUBB was developed in 1995 in C language. There is a renewed version in existence from 1997 with a completely redesigned algorithm which is not an extension of the old implementation. This new version is implemented in C++ language. PUBB runs on PVM and the experiments have been performed on some workstation networks.

The BOB (a Branch-and-Bound Optimization liBrary) library from the University of Versailles, France is studied in section 2.3. This library has suffered many changes from 1995, when the first version in C language appears. The authors wrote a new version of the library called Bob++ in 1999 implemented in C++. The distributed part of Bob++ works on PVM and has been used on the distributed machine PARAGON. We have also contacted the authors who confirm of an MPI draft version running on PC clusters.

PICO (Parallel Integer and Combinatorial Optimize) is the last library studied in section 2.4. It is an object-oriented framework developed at Rutgers University of Piscataway, New Jersey, and Sandia National Laboratories, Albuquerque, in 2000. PICO is structured as a C++ class library and the parallel layer is implemented using the MPI. The computational results are obtained on the Janus massive parallel computer, which consist of thousands of Pentium-II processors.

Section 2.5 covers other general tools which are not restricted to Branch--and-Bound. Finally, in the last section we present a summary table of the features of each analyzed library.

In each section we begin explaining the user interface of the studied library and we illustrate the use of it with the 0/1 Knapsack problem. Then we detail the sequential and the parallel implementation.

2.1 PPBB - LIBRARY

The Portable Parallel Branch-and-Bound Library (PPBB-Library) [92] was developed at the University of Paderborn in Germany at the beginning of the nineties. It was the first attempt to write a tool which parallelizes sequential Branch-and-Bound algorithms. At that moment it was an original idea. A programmer who wanted to run his sequential algorithm on a parallel system, only needed to use the library. He or she didn't need either knowledge about the hardware architecture or the parallelization mechanisms.

PPBB library was designed to run on any distributed memory multicomputer architecture using the given message passing functions. The first version of the library was tested on Transputers [91, 93]. It is available under PVM and is written in C. After filling in a form to request a library version of PPBB-Lib in the web page: http://www.uni-paderborn.de/fachbereich/AG-/monien/SOFTWARE /PPBB/ppbblib.html, we received the library by email. We have not been able to install it.

For most advanced users, who are experts on parallelism the library can be used for implementing and testing load balancing strategies while the loads are provided by a particular Branch-and-Bound application.

2.1.1 User Interface

Using the PPBB library, the essential parts of a parallel implementation of Branch-and-Bound algorithms are included in six main components: branching procedure, bounding procedure, methods to reduce the solution space, functions for data input and output (I/O), queue management for the subproblems and load distribution. The first three components are application-dependent, and are programmed by users as a `application process`. The last three components are encapsulated in a `communication process` of PPBB_LIB. An application process is coupled with a communication process through a so called Branch-and-Bound `interface`. To adapt a Branch-and-Bound algorithm to the library, the user has to extract the functions for subproblem and I/O handling from the original algorithm and has to replace them by library functions. PPBB functions are classified under four main groups: General Functions, I/O Management, Queue Management and Load Balancers. Figure 2.1 shows in detail which ones compose each group.

The pseudo C code, presented in figure 2.2, shows an example for a Branch-and-Bound algorithm with an objective function to be maximized adapted to the library interface. Specifically, we implement the application of the PPBB library interface to solve the 0/1 Knapsack Problem. We will make reference this figure in order to show the use of the different functions.

General Functions. This group includes functions for the initialization of the library, process identification, and time measurement. The available functions are the following:

- `ppbb_Init():` Initializes and starts the library. Every application has to call `ppbb_Init` at the beginning of the `main` function to start up the processes and communication structures of the library. The function `ppbb_Init` has to be called before any other function of PPBB_Lib can be used (see line 6).
- `ppbb_End():` Ends the library. When the Branch-and-Bound algorithm has ended its computation on the problem instance the application process has to call the function `ppbb_End` to end all library functions. The

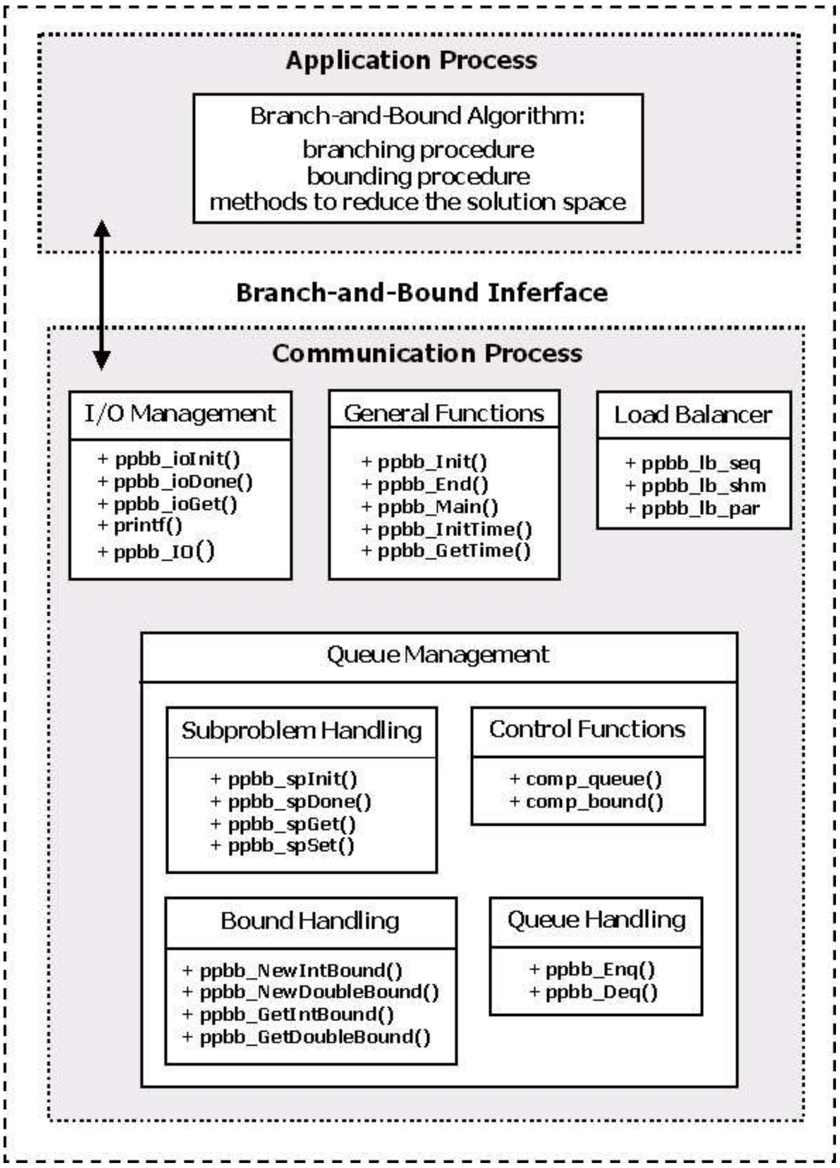


Fig. 2.1 Scheme of PPBB Library Interface

```
int main (int argc, char *argv[]) {
  io_t *info;  problem_t *prob;
  knp_subproblem *knps;  subproblem_t *subprob;
  double bound, bestsol;
  /* step 1: get data and initialization */
  ppbb_Init (&argc,argv,"knap",OBJ_MAX,DOUBLE,knap_io,
             comp_queue,comp_bound,NULL,NULL);
  ppbb_NewDoubleBound (bound);
  info = ppbb_IO (ppbb_ioInit (IC_READ,READ_PROBLEM,
                  strlen(filename)+1,filename));
  prob = ppbb_ioGet_data (info);
  init_knp_data();
  /* step 2: solve the problem */
  ppbb_InitTime();
  if (ppbb_Main())
  { main process starts computing, generates n subproblems;
    for n-1 subproblems do
      subprob = ppbb_spInit (0,INT,upper_bound,0.0,
                          sizeof(knp_subproblem),knps,0,0);
      ppbb_Enq (subprob);
  } else {  other processes request for subproblem
    subprob = ppbb_Deq(0);
  }
  while (subprob != NULL) {
    compute subproblem;
    generate new subproblems;
    for each generated subproblem do {
      compute lower bound;
      if lower bound < upper bound {
        new_subprob = ppbb_spInit (0,INT,upper_bound,0.0,
                    sizeof(knp_subproblem),knps,
                    ppbb_spGet_depth(subprob)+1,0);
        ppbb_Enq (new_subprob);
    } }
    ppbb_spDone (subprob);
    subprob = ppbb_Deq (0);
  }
  time = ppbb_GetTime ();
  /* step 3: print solution */
  if (ppbb_Main ())
     printf ("Computing time:%.3f seconds\n", time);
  ppbb_IO(ppbb_ioInit(IC_WRITE,PRINT_SOLUTION,sizeof(double),bestsol));
  ppbb_End ();
  return 0;
}
```

Fig. 2.2 Pseudo Code of PPBB main Function

call of `ppbb_End` has to be the last command executed by the application (see line 43).

- `ppbb_Main():` Determines the main process (line 15).
- `ppbb_InitTime():` Initializes distributed time (line 14). The function has to be called by all application processes or none, because all processes are synchronized.
- `ppbb_GetTime():` Measures time (line 38).

I/O Management. While running the Branch-and-Bound algorithm, the input and output of data has to take place at a special point on the parallel system. For routing different kinds of I/O data through the parallel system a standardized data format is necessary. The so-called `I/O structure` is shown in figure 2.3.

The following functions can be used to work with the `I/O structures`:

- `ppbb_ioInit():` Generates an I/O structure (line 9).
- `ppbb_ioDone():` Deletes an I/O structure.
- `ppbb_ioGet_<varname>():` Reads a variable of an I/O structure (line 11).
- `printf():` Is used for screen outputs (line 41).
- `ppbb_IO():` Has to be used for the input and the output of data (lines 9 and 42).

The additional I/O process manages the I/O of data by calling the function of collected I/O functions, which is called `knp_io` (see 2.4). This function is passed from the application process to the i/o process during initialization by `ppbb_Init`. The I/O process runs `knp_io` every time the function is invoked by a `ppbb_IO` statement of any process. The I/O process is an additional process for input and output connected to the communication process with identifier zero. It exclusively controls the input and output of data between the parallel system and the external world. Figure 2.6 presents the relation between the different processes in a distributed computing environment.

```
typedef struct io {
  /* external data items: modified by application */
  int  type;  /* type of I/O operation that could be
              IC_READ: reading data
              IC_WRITE: writing data
              IC_ERROR: writing error messages
              IC_DEBUG: writing debug information
              IC_COLLECT: collects information of all processes */
  int  func;  /* to identify different input or output
              functions (see an example in knp_io) */
  int  datasize;
  void *data;

  /* internal data items: modified by ppbb */
  int   pid;
} io_t;
```

Fig. 2.3 I/O Structure for Routing I/O Data

```
io_t *knp_io (io_t *info) {
  type = ppbb_ioGet_type (info);
  func = ppbb_ioGet_func (info);

  switch (type) {
    case IC_READ:
      if func = 1
      { call function which reads N_object, Capacity, and
        weight and profit of each object from a file }
      break;
    case IC_WRITE:
      if func = 1
      { call function which writes best solution of
        Knapsack problem to a file }
      break;
    case IC_ERROR: break;
    case IC_DEBUG: break;
    case IC_COLLECT: break;
} }
```

Fig. 2.4 Function `knp_io`

```
# define OBJECTS 8;

typedef struct {
  int N_objects;
  int Capacity;
  int weight[OBJECTS];
  int profit[OBJECTS];
} problem_t;

typedef struct {
  int CRest;    /* current capacity */
  int obj;      /* actual depth in the search tree */
  int profit;   /* current profit */
} knp_subproblem;

typedef struct subproblem {
  /* external dataitems: modified by application */
  int   type;   /* identifies different subproblem types */
  bound_t bound;    /* bound of this subproblem */
  int   datasize; /* number of bytes allocated for *data */
  void  *data;    /* pointer to the subproblem  */
  int   depth;    /* actual depth in the search tree  */
  int   priority; /* load balancing criteria */

  /* internal dataitems: modified by ppbb  */
  int   pid;     /* ID of the immediate sending process */
  int   channel;  /* channel on which subproblem arrives */
  int   origin;   /* ID of generating process */
  int   destcnt;  /* number of sends since generation */
} subproblem_t;
```

Fig. 2.5 Data Formats of the problem_t and subproblem_t Structures Adapted to 0/1 Knapsack Problem

Queue Management. For routing subproblems through the parallel system, a standardized format is required. Therefore, the library provides the so called `subproblem structure` in which subproblems have to be inserted before they are sent. This structure is shown in figure 2.5.

The library provides the following functions to handle the subproblem structure:

- `ppbb_spInit():` Generates a subproblem structure (see line 18).
- `ppbb_spDone():` Deletes a subproblem structure (line 35).
- `ppbb_spGet_<varname>():` Reads a variable of a subproblem structure (line 30).
- `ppbb_spSet_<varname>():` Sets a variable of a subproblem structure.

After reading the problem instance, every application process can start the execution of the Branch-and-Bound algorithm. The most practical situation is to identify one process that starts the computation on the problem instance and to wait with the other processes for the first subproblem instead of producing the same subproblem instances with several processes running the same algorithm on the problem instance. Waiting for a subproblem means requesting a subproblem from the queue management. Subproblems are provided by the queue management to the Branch-and-Bound application at every request as long as there are subproblems available. For the exchange of subproblems between the application and the queue management the following functions are provided:

- `ppbb_Enq():` Passes a subproblem structure to the queue management (lines 20 and 33).
- `ppbb_Deq():` Requests a subproblem structure from the queue management (line 36).

If any application process finds a new best solution, this produces a new bound which has to be broadcast to all other processes as quickly as possible in order to avoid computing on bad subproblems. The broadcasting of the bound is done by the library, but the application has to inform the library that a new bound has been found. The functions provided to handle the bound are:

- `ppbb_NewIntBound():` Announces an integer bound to the library.

- `ppbb_NewDoubleBound():` Announces a double bound to the library (line 8).
- `ppbb_GetIntBound():` Brings the actual integer bound from the library.
- `ppbb_GetDoubleBound():` Brings the actual double bound from the library.

To make sure the actual bound is always used, the application has to call the function `ppbb_GetIntBound` or `ppbb_GetDoubleBound` every time it operates with the bound value.

The subproblems can be stored in different kinds of queues. The load balancers provided by the library use a priority queue to store the subproblems. The user has to provide a comparative function which determines the sorting order of the subproblems in the queue. This function is called the `comp_queue`. With help of this function the Branch-and-Bound application can determine which subproblem is returned at the next `ppbb_Deq` call.

If a new bound has been found the queue management has to update the queues, this means deleting all subproblems in the queues whose bounds are worse than the new bound. The user has to provide the `comp_bound` function that compares the bound of a subproblem with the bound which represents the best solution and so implement the updating of the queue.

Load Balancers. PPBB_Lib provides standard load balance functions for the sequential, shared-memory and parallel versions: `ppbb_lb_seq()`, `ppbb_lb_shm()` and `ppbb_lb_par()` respectively.

2.1.2 Parallel Implementation

Parallelization used in the library is based on a dynamic load distribution of the subproblems in the parallel system. Figure 2.6 presents an overview of the PPBB-LIB architecture in a distributed computing environment. The Branch-and-Bound algorithm adapted to the library runs parallel on several application processes. All application processes work on the distributed subproblems, that is to say on one problem instance. Every application process is supported by one communication process to which it is connected by the Branch-and-Bound interface. Both the application and communication process together is called a node and has its identification number id. The node with id 0 is called the superior node. The communication process of the superior node is connected to an additional I/O process and an additional monitor process.

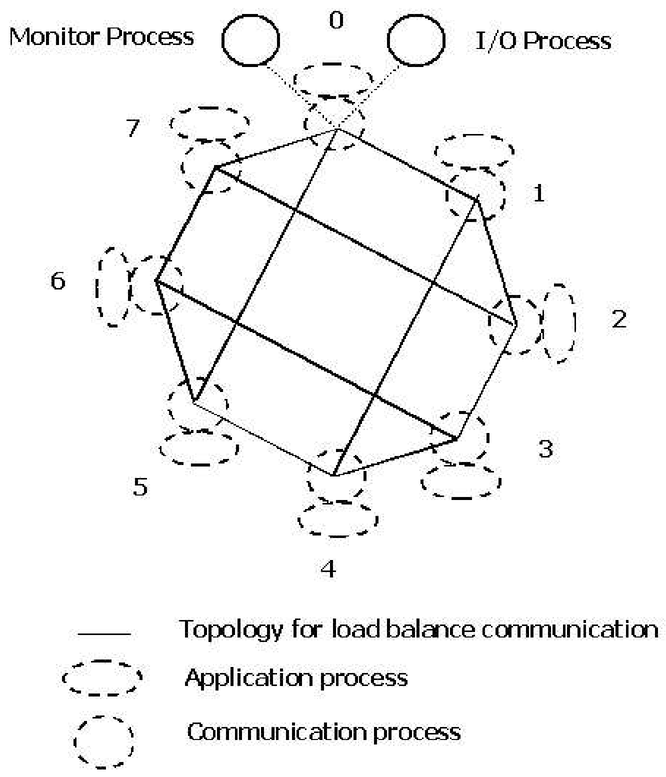


Fig. 2.6 The Parallel Process System of the PPBB-Library

The communication processes are connected by a topology defined by the load balancers provided by the library. This topology is used for exchanging load balancer messages and transfer subproblems for load balancing. Lines in figure 2.6 represent the communication channels between processors. In addition, all communication processes are connected by a ring, which can be used by the communication kernel for termination detection and I/O data transport. The I/O process runs the extracted and collected I/O functions of the original Branch-and-Bound algorithm and provides the I/O data for all other processes.

The monitor process exists only if the library version which provides monitoring is used. With the help of the monitor process the monitoring can be achieved independently of the application and communication processes used. All relevant data are collected by each process. This has only a small influence on the performance of the system. Some information can be displayed on screen or written to info files. The user interacts with the monitoring and decides what information he wants to see at what time.

The authors propose a number of load distribution strategies incorporated within PPBB in the papers [20, 66, 95]. These strategies are:

- `The RAND and the ACWN`

  With the `RAND` policy, a load distribution operation is performed once a subproblem is generated, and the newly generated subproblem is then sent out to a randomly selected neighbor. Using this policy, no local load information needs to be maintained nor any load information needs to be sent to other processors.

  The `ACWN` policy differs from the `RAND` in the destination choice when a newly generated subproblem is required to be migrated out. It always selects the least loaded nearest neighbor as the recipient of the subproblem. A processor that receives a subproblem keeps it in its local heap if it finds its own load is less than that of its least loaded neighbor. With the `ACWN` policy, each processor is required to maintain its local information and adjacent processors are needed to exchange this information periodically.

- `The PRAND and the PACWN`

  In both `RAND` and `ACWN` strategies, only those newly generated subproblems are allowed to migrate. In Branch-and-Bound computations, subproblems have large variants in terms of their computational requirement. It is known that the computational requirement of a subproblem is closely relevant to its lower bound. The `RAND` and `ACWN` are thus modified in such a way that the second best subproblem in the local heap is selected for migration. These two new strategies with different priorities are named `PRAND` and `PACWN` respectively.

- `The LADE and the LADF`

  Both the `LADE` and `LADF` strategies follow the same idea of local averaging in each load distribution operation, but differ in the way averaging is performed. In the `LADE` policy, a processor in need of load distribution balances its workload with one of its neighbors, while in the diffusion policy, the processor manages to balance its workload with its surrounding neighbors around. Since the computational load of a processor is mainly determined by the weight and the size of its local heap, a load balancing operation of the `LADE` and `LADF` comprises two steps: balancing the heap weighted subproblems and then balancing the heap sizes through the migration of the least weighted subproblems.

2.2 PUBB - LIBRARY

PUBB (Parallelization Utility for Branch-and-Bound algorithms) is one of the generalized software tools developed at the Science University of Tokyo, Japan. The first version of PUBB was developed in 1995 and was an implementation of the central control scheme with a single subproblem pool. PUBB provides a skeleton of a parallel Branch-and-Bound algorithm and has several interfaces for user defined routines. For the development of applications PUBB also provides a skeleton of a sequential Branch-and-Bound algorithm and the interfaces for user defined routines are exactly the same as those used in the parallel method. PUBB possesses the advantage that it is easy to translate from an existing sequential to a parallel algorithm.

The 1997 version of PUBB [84] is a completely redesigned algorithm and not an extension of the old implementation presented in [85]. This time the system has been rebuilt to accommodate distributed control with multiple subproblems pools. All program codes are written in C++ and use the PVM library.

The user manual and the available codes of the version in C of PUBB can be downloaded from the following web page address: http://al.cs.tuat.ac.jp/∼yshi-nano/pubb/. The information about the C++ implementation appears in the article [87]. It is not possible to download the C++ version of the library.

2.2.1 User Interface

The authors of PUBB propose in [87] a model of a generalized system for parallel Branch-and-Bound algorithms based on an object-oriented paradigm. They consider the preparation of base classes for general Branch-and-Bound algorithms and the preparation of a skeleton for using the base classes as a generalized system. The parts of classes that depend on solving algorithm for each specific problem are written as derived classes. The model is composed of the following six classes: Problem Manager, InitData, Solution, Subproblem, Load Balancer and Solver (see figure 2.7). All objects are created from derived classes for solving algorithm for each specific problem.

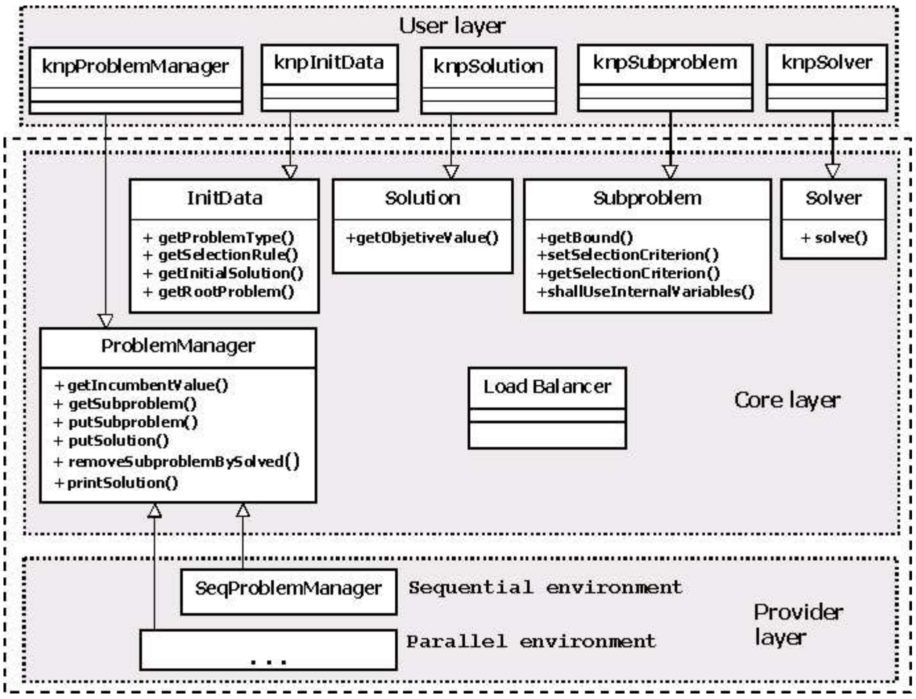


Fig. 2.7 UML Scheme of PUBB

In order to describe the skeleton, we show the interface related program codes of derived classes adapted to the Knapsack Problem (knp) as an example. For presentation of the skeleton, the main function and the derived class of the `Problem Manager` for sequential Branch-and-Bound algorithms are developed in Figure 2.8. The derived class for `Solver` is presented in Figure 2.9. In figure 2.8 the construction of the `Problem Manager` is carried out in (line 10 and line 3-7) and the result output corresponds to the `printSolution` call of the `Problem Manager` (line 13). Other parts of general Branch-and-Bound algorithms correspond to the codes of the `solve` function (line 12) of the `Solver` class. In figure 2.9 the restrictions on the usage of the member functions of the `Problem Manager` are that `getSubproblem` needs to be called at the beginning of an evaluation (line 10) and `removeSubproblemBySolver` (line 19) is to be called at the end of an evaluation. The number of calls of `putSubproblem` is the number of branches from the subproblem obtained from the `Problem Manager` at the beginning of an evaluation. Some low-level parallelism can

```
class knpProblemManager: public ProblemManager {
public:
  knpProblemManager(char *pcInputFileName, ...) {
    pInitData = new knpInitData(pcInputFileName);
    pBestSolution = pInitData->getInitialSolution();
    pRootProblem = pInitData->getRootProblem();
  }
};

int main( int argc, char* argv){
  knpProblemManager* pkPm = new knpProblemManager(...);
  knpSolver* pSolver = new knpSolver();
  pSolver->solve(pkPm);
  pkPm->printSolution();
  delete pSolver;
  delete pkPm;
  return 0;
}
```

Fig. 2.8 The main Function and the Definition of knpProblemManager

```
class knpSolver: public Solver {
public:
  void solve( ProblemManager* );
private:
};

void knpSolver::solve( ProblemManager* pPm ){
  knpProblemManager* pkPm = ( knpProblemManager* )pPm;
  knpInitData* pIDat = ( knpInitData* )pkPm->getInitData();
  while( knpSubproblem* pkSp
         = ( knpSubproblem* )pkPm->getSubproblem() ){
    if ( pkSp->shallIUseInternalVariables()
         == Subproblem::YOU_SHOULD_NOT_USE_IT ){
      // Initialize internal variables
    } else {
      // use internal variables of the previous calculation
    }
    // Routines for an evaluation
    . . .
    pkPm->removeSubproblemBySolved( pkSp );
  }
}
```

Fig. 2.9 Definition of knpSolver

be implemented (line 17). Implementations of the `Problem Manager` are the greater part of a generalized system development. `The Problem Manager` can handle *transported objects* as abstract data type and also its internal architecture (single-pool, multiple-pool, and so on) is hidden from the `Solver`. Hence program codes for problem depending implementations can be exchanged from one parallelization architecture depending implementation to another without any modification.

In some parallelization architecture, it may be necessary for `Problem Manager` to map on several processors and all of its functions are realized by several modules. In such a case, the designer of a generalized system needs to construct other class types which are derived from the base class of the `Problem Manager` with respect to the mapping. Even if such a mapping changes internal architecture of the `Problem Manager`, all services provided in the base class can be inherited. The main functions should be provided by the generalized systems, because the number of main functions containing the written codes needs to be changed depending on the architecture. As in the case of the `Problem Manager` composed of several modules, the module on which procedures of data input and results output are performed depends on the architecture. The derived class of the `Problem Manager` is closely related to the parallelization architecture used and it should be written mechanically. Therefore, the main functions and the derived class of the `Problem Manager` should be generated by the system.

Next we detail each class and the functions that compose each of them in the PUBB Library.

Problem Manager (PM). This object is an abstraction of the manager for solving problems with Branch-and-Bound algorithms and contains all the necessary information needed by the algorithm. Therefore, this object determines the rule by which the next subproblem to be solved should be selected. This rule is called the *selection rule*. In this object, elimination rule that recognizes and eliminates subproblems that cannot yield an optimal solution to the original problem, is also applied when an improvement of a solution value is notified. The following public member functions are produced by the base class:

- `getIncumbentValue():` returns the incumbent value. In the parallelization, the incumbent value should remain globally best in the `Problem Manager`.
- `getSubproblem():` returns a `Subproblem` to evaluate (line 10 in figure 2.9).

- `putSubproblem():` puts a created `Subproblem` into a `Problem Manager`.
- `putSolution():` puts a created `Solution` into the `Problem Manager`.
- `removeSubproblemBySolved():`removes an evaluated `Subproblem` from the `Problem Manager` (line 19 in figure 2.9) .
- `printSolution():` outputs a solution by calling print member function of `Solution` (line 13 in figure 2.8).

InitData. This object is an abstraction of all initialization data for Branch--and-Bound algorithms. It provides problem instance data, an initial solution and a root problem. When this object is constructed, it reads data for the target problem and is also initialized by the target problem. The following public member functions are declared in the base class and carried out in the derived class:

- `getProblemType():` returns whether the problem is maximization or minimization.
- `getSelectionRule():` returns the value which indicates a selection rule.
- `getInitialSolution():` returns a `Solution` object which is a initial solution (line 5 in figure 2.8).
- `getRootProblem():` returns a `Subproblem` object which is a root problem (line 6 in figure 2.8).

Solution. This object is an abstraction of the solution. With the help of the `InitData` object, this object represents the solution. When this object is constructed, an objective value is passed as an argument of a base class constructor and is set in this object. The following public member function is carried out in the base class.

- `getObjetiveValue():` returns the objective value of this object.

The following public member function is declared in the base class and realized in the derived class.

- `print():` displays a solution by editing the contents of this object.

Subproblem. This object is an abstraction of the subproblem. With the help of the `InitData` object, this object represents the subproblem. When this object is constructed, a bound value is passed as an argument of a base class constructor and is set in this object. The following public member functions are realized by the base class.

- `getBound():` returns the bound value of this object.
- `setSelectionCriterion():` sets a selection criterion value on this object.
- `getSelectionCriterion():` returns the selection criterion value.
- `shallUseInternalVariables():` returns whether this object is a direct result of the Subproblem evaluated in previous calculation of the same Solver or not (line 12 in figure 2.9).

Load Balancer. This class spans the corresponding `Solver`. The authors define *transported objects* as the generic name for `InitData`, `Solution` and `Subproblem`. In parallelized implementation, since several `Solver` objects should exist on a system, *transported objects* may be transferred between processors. The `Load Balancer (LB)` makes decisions over load balancing [82].

Solver. This object is an abstraction of the solving algorithms for a subproblem. The algorithms include the *bounding rule*, that computes a bound value on the optimal solution of a subproblem, the *branching rule* that determines how to divide a subproblem into subproblems, and the *elimination rule*. The following public member function is declared in the base class and take place in the derived class.

- `solve():` performs a sequential Branch-and-Bound algorithm (line 12 in figure 2.8).

There is one `Problem Manager (PM)` in the system, while there are many `Load Balancers (LBs)` and `Solvers` in it. There is a one to one correspondence between `LB` and `Solver`, and the corresponding pairs run on the same workstation (see figure 2.10). Subproblems are evaluated in the `Solver`. In PUBB, the evaluation of a subproblem can be defined as follows: 1) obtain a subproblem from the system, 2) compute a bounding value (UPPER_BOUNDING), 3) check whether the subproblem can be eliminated or not, 4) select branching variables and generate new subproblems if necessary (BRANCHING), and 5) remove the current subproblem.

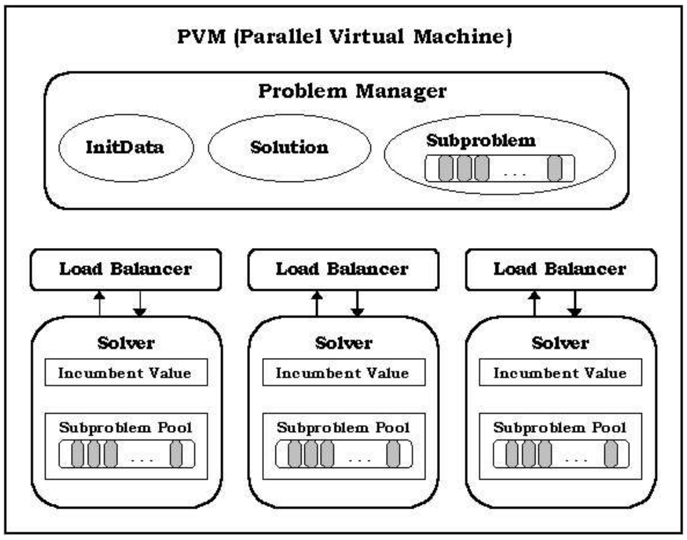


Fig. 2.10 Instances of Objects in the PVM

The data structure of a subproblem pool consists of three index heaps [84]: the selection ordered heap which is ordered by the selection criterion value, the best bound ordered heap and the worst bound ordered heap. Each heap element has only pointers. Basic heap operations such as insert or remove are performed based on the value of the subproblem data. The pruning procedure removes subproblems from the subproblem pool by worst bound order as long as the bound value of the subproblem on the top of the worst bound heap is worse than the incumbent value. The PUBB supports depth first, breadth first, best bound first, hybrid and some heuristic value ordered selection rules.

2.2.2 Parallel Implementations

PUBB has three running modes with respect to the control schemes used in [82, 84]. The central control scheme is applied to a `Master-Slave (MS)` mode, the distributed control scheme is applied in the `Fully- Distributed (FD)` mode and the mixed control scheme is applied in the `Master-Slave` to `Fully-Distributed (MStoFD)` mode.

Master-Slave (MS) model. PUBB in the `MS` mode has a single subproblem pool and the tree search is controlled by the centralized scheme, as shown in figure 2.11. To begin with `Problem Manager (PM)` has to be initiated. The `PM` reads the problem instance data and recognizes the target problem type (minimization or maximization) and the selection rule. Subsequently, an initial solution and a root problem are obtained by calling the user defined functions. Initialization of the `PM` ends when the root problem is put into the subproblem pool. Then, the `PM` waits for requests.

When a `Load Balancer (LB)` starts, it generates a `Solver` which notifies the completion of the initialization via the corresponding `LB`. Then, the `PM` checks for available `Solvers` and manages their operation. After the initialization, the `PM` works as the master and the `Solvers` as slaves. As long as the subproblem pool of the `PM` is not empty and there is more than one idle `Solver`, the `PM` selects subproblems and assigns them to the `Solvers`. When a `Solver` receives a subproblem, it evaluates it. If an improved solution is found, the `Solver` sends it to the `PM` and broadcasts the objective value to the PVM group of `LBs`. Each `LB` receives the objective value and sends it to the corresponding `Solver`. If new subproblems are generated, they are also sent to the `PM`.

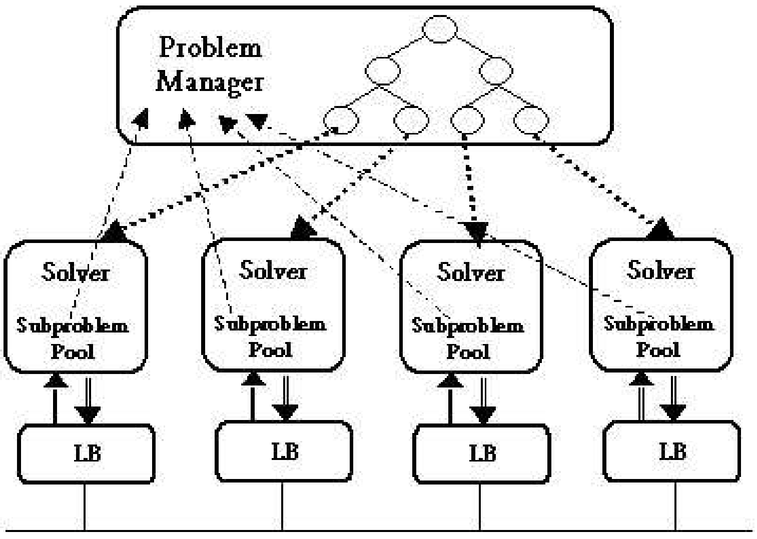


Fig. 2.11 Scheme of Master-Slave Mode

Fully Distributed (FD) model. PUBB in the `FD` mode has multiple subproblem pools and the tree search is controlled by a fully distributed scheme. In the `Solver`, a sequential Branch-and-Bound algorithm is performed obtaining subproblems from a local subproblem pool and putting newly generated subproblems into it. In order to avoid imbalance of workload among the `Solvers`, each `Solver` may send or receive subproblems at the end of each evaluation. When the evaluation is completed, the `Solver` sends its own local pool states (the number of subproblems, their best bound and so on) to the corresponding `LB`. Hence, each `LB` recognizes the state of the corresponding `Solver`, the `LB` searches the partner to send or receive subproblems among other `LBs`. The `LB` usually begins to search when the number of subproblems in the corresponding `Solver` is less than a parameter. Subproblems are transferred between `Solvers` only when the number of subproblems into the pool of some `Solver` is less than that of the parameter or is full to its capacity. A sender side `Solver` having many subproblems is selected and the selected sender `Solver` transfers one subproblem for every one that it evaluates. Figure 2.12 shows this scheme.

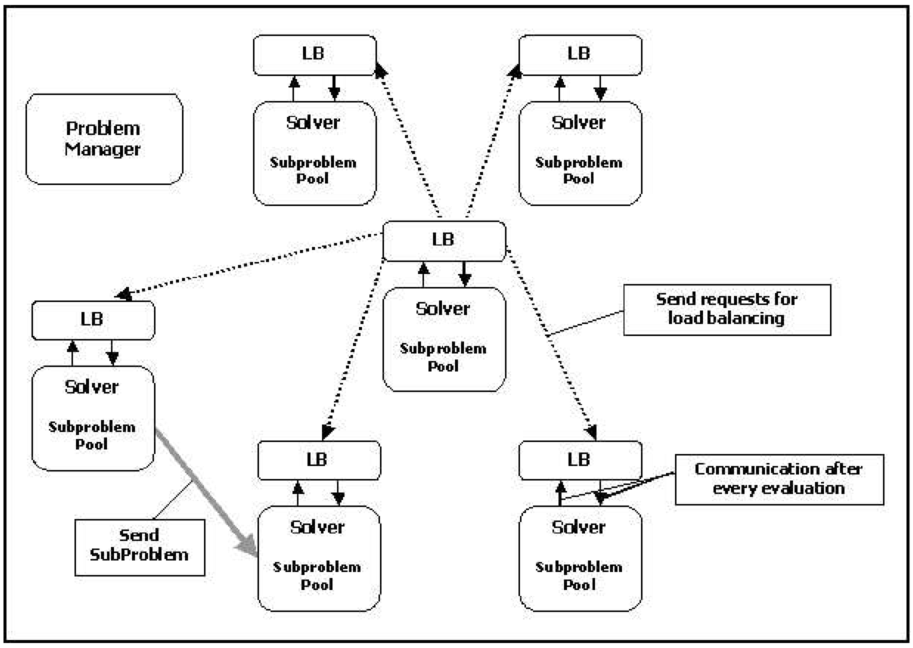

Problem Manager
LB
Solver
Subproblem Pool
Send requests for load balancing
Communication after every evaluation
Send SubProblem

Fig. 2.12 Scheme of Fully Distributed Mode

Master-Slave to Fully-Distributed (MStoFD) model. PUBB starts execution with the same initialization sequence and internal architecture as in the MS mode. When the number of subproblems maintained by the PM becomes greater than a certain value specified as a parameter, the internal architecture is modified to that of the FD mode. As soon as this condition is detected, the PM broadcasts the message of the change architecture to the group of LBs and each Solver receives the message via the corresponding LB. Upon receiving the message, the Solver sends an acknowledgment message to the PM and starts placing recently generated subproblems into the local pool. When the PM recognizes that all the Solver states have changed, it selects a subproblem from the pool using best bound order and distributes it to a different Solver one by one using a repeating cyclic scheme.

2.3 BOB - LIBRARY

BOB (a Branch-and-Bound Optimization liBrary) is a library implemented by University of Versailles, France. The current BOB library was designed for several years from 1995 [60, 61, 18, 17]. It has the double goal of allowing, on the one hand, the Combinatorial Optimization community to implement their applications without worrying about the architecture of the machines and benefiting from the advantages provided by parallelism and on the other hand BOB offers the Parallelism community a set of benchmark Combinatorial Optimization applications to test its parallelization methods and/or tools. The main limitation of BOB is the position of an algorithm in the architecture design. First, only Branch-and-Bound is proposed and it is difficult to add a new algorithm without hacking the source code of the library. Second, it is impossible to combine different algorithms and thus different parallelizations in the same code. This is the case for example in a Branch-and-Price algorithm where this is mainly a Branch-and-Bound in which each subproblem uses other search algorithms like for example shortest path constraint. For these reasons, the authors began writing a new version of the library called Bob++ in 1999. The most visible change is the use of the C++, an object oriented language [59]. The first version was written in C. Bob++ library is designed for sequential and parallel Branching (Branch-and-Bound, Branch-and-Price and Branch-and-Cut) algorithms. An algorithm is represented by an Object. A user can choose one of the algorithms proposed in the library or define a new one.

The BOB library is founded on the notion of Global Data Structure which makes the parallelization methods independent from the applications, and vice-versa. The authors describe for this global priority queue different implementation models according to the architecture of machines (serial, parallel with shared or distributed memory). The serial part of BOB does not cause any portability problems since it uses the standard environment of UNIX. However, this is not the case of the parallel implementation. BOB needs, on a shared memory machine, the use of mutual exclusion primitives. The distributed part of BOB works on PVM, but in order not to limit BOB to this library of message exchanges, it can work on a set of macros, which could correspond to PVM calls or functions implementing PVM calls at the top of a particular library. This is used for the primitives of sending/receiving messages. With respect to launching processes, the code can be written without being forced to simulate the PVM calls for handling processes.

The only available information about Bob++ is generated by doxigen and can by found in the web page: http://www.prism.uvsq.fr/ blec/Research/BOBO-/index.html. But the library is not available.

2.3.1 User Interface

The Bob++ library is composed by the following two main parts, as shows figure 2.13:

- The core library, which includes the options, the `BobMain` class and the data structures that store objects like nodes and solutions.
- The hierarchy of the different possible algorithms that are called User Algorithms. In this work, we will reference only the part corresponding to the Branch-and-Bound.

An example of the use of the specific classes to solve the 0/1 Knapsack Problem (KNP) using the Branch-and-Bound method is shown in figures 2.14 and 2.15.

2.3.1.1 The core of the library. This module of Bob++ is formed by various groups of classes.

The machine architecture manager. The class `BobParaManager` manages the architectural aspect of the machine (Communication library, Threads,...). The public methods are: BobParaManager(), GetPid(), Init(), Run(), BobAlgo *Curr() and Stop(),and the protected method is AddAlgo(BobAlgo *al).

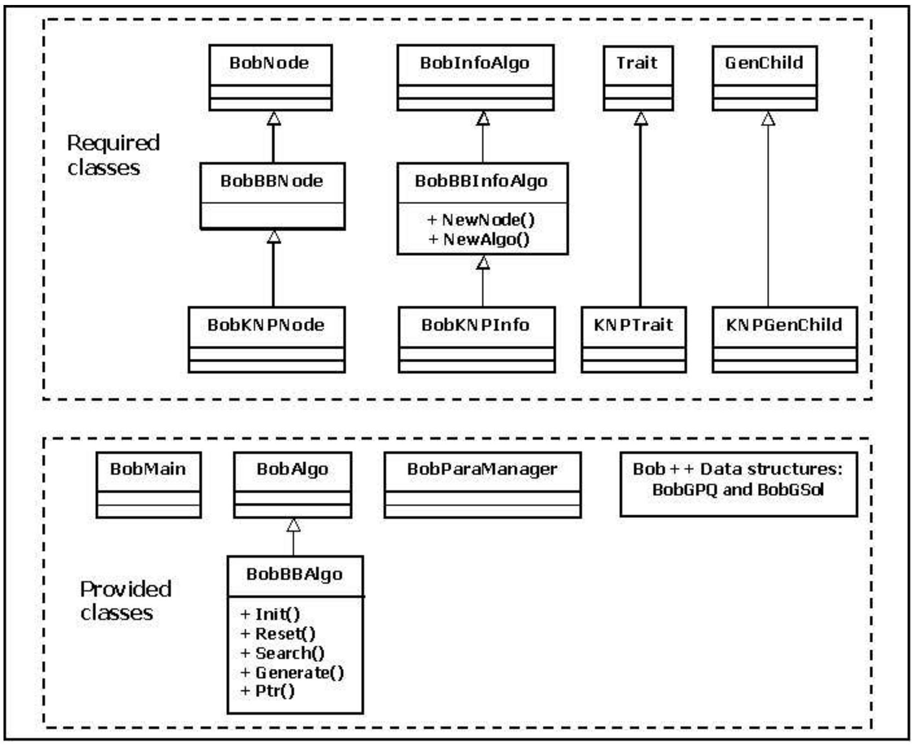


Fig. 2.13 UML Scheme of Bob++

The authors comment that at this time, information about only two `BobParaManagers` exist: the sequential BobParaManager `seq` and the BobParaManager for shared memory machine `smp`, but the latter one is not written with doxigen.

BobMain. The `BobMain` is the entry point for the application from the user point of view. The `BobMain` class stores all Bob++ core aspects the `BobInfoAlgo` classes of an application, the `BobParaManager` (the machine architecture manager), the GUI, the Option, and so on. An object of this class is defined and initialized by the user with the name `Bobmain`.

The user has to initialize the specific method `BobMain::AppRegister()`. An example of how to do this in the case of Knapsack Problem is the following:

```
class KNPNode;
class KNPInfo;
class KNPGenChild;

class KNPTrait {
public:
   typedef KNPNode Node;
   typedef KNPInfo Info;
   typedef BobBBAlgo<BobKNPTrait,BobBBPriAlloc> Algo;
   typedef KNPGenChild GenChild;
};
```

Fig. 2.14 Definition of KNPTrait

```
template <class Trait,class PriAllocator>
class BobBBAlgo : public BobSchedAlgo<Trait,PriAllocator>,
                  public BobEvalSolAlgo<Trait> {
public:
   typedef typename Trait::Node TheNode;
   typedef typename Trait::Info TheInfo;
   typedef typename Trait::Algo TheAlgo;
   typedef typename Trait::GenChild TheGenChild;
   . . .
   virtual void Search(TheNode *n) {
      if ( n->IsSol() ) {
         TheNode *ni = new TheNode(n);
         SolUpd(ni);
         return;
      }
      if ( SolIsBetter(n) ) {
         BobSchedAlgo<Trait,PriAllocator>::Search(n);
      } else StPrun();
   };
   /// the new generate method
   virtual bool Generate(TheNode *n) {
      if ( n->IsSol() ) {
         TheNode *ni = new TheNode(n);
         SolUpd(ni);
         return false;
      }
      return SolIsBetter(n);
}  };
```

Fig. 2.15 Definition of BobBBAlgo

```
void BobMain::AppRegister() {
   KNPBobInfoAlgo *InfoAlg = new KNPInfoAlgo();
   RegisterAlgo(&InfoAlg,0);
}
```

Bob++ Data structures. The data structures are used in Bob++ to store a set of `BobNodes`. The two main data structures are `BobGPQ` and `BobGSol`. The Global priority Queue `BobGPQ` is used to store the pending work that Bob++ has to distribute to the BobNodes. The `BobGSol` is used to store solutions.

2.3.1.2 The user hierarchy. The Bob++ library is based on the basic idea that its own problem resolution algorithm is written by the derivation of the one of the proposed algorithms. All of these *User Algorithms* derive from the basic `BobAlgo` and related classes `BobInfoAlgo` and `BobNode`. An Exact Algorithm in Bob++ is mainly divided into 5 classes:

BobAlgo. This class represents the basic class the BoB algorithms can execute. This class can not be executed. Real algorithms inherit this class. It stores the run method which performs the search. It encapsulates the different data structures: the pool of pending nodes `BobGPQ` which is the Global priority Queue class and stores `BobNodes`, the pool of exploring nodes (if needed), the best known solution, and so on. In a parallel environment, an instance of this class is executed on each processor data, which is distributed or shared by the Global Data Structures.

BobNode. It represents a node of the search space. The only feature implemented in a base `BobNode` is the reference counter. The reference counter stores the number of times that a node is in a Data Structure.

BobInfoAlgo. This class includes all the information needed by an algorithm to create one of them.

Trait. Prototype class to fix the user types for an application: Node, Algo, Info and GenChild.

GenChild. This class must be implemented to create the child nodes of a parent node.

The classes needed to implement a Branch-and-Bound algorithm are the following:

BobBBNode. This class represents a node of the search tree, in this case a subproblem in a Branch-and-Bound algorithm. Its public method `BobBBNode()`.

BobBBInfoAlgo. This class includes all the information needed by the Branch-and-Bound algorithm to schedule it. Its public methods are: `NewNode()`, `NewAlgo()` and `BobBBInfoAlgo()`.

BobBBAlgo. Public Methods are: `BobBBAlgo()`, `Init()`, `Reset()`, `Search()` where the GenChild::operator() class calls for each new generated node, and `Generate()`.

2.3.2 Parallel Implementation

Processes exploring nodes get their workload by executing an algorithm participating in the management of the `global priority queue` (`GPQ`). The notion of `GPQ` encapsulates two components: the parallel management algorithm (concurrent access protocol, load balancing, etc), and the local or shared priority queues, based on the sequential algorithms of priority queues.

Bob++ library is based on the notion of global priority queue. The authors also introduce the notion of `global lower/upper bound` (`GL/UB`). The value of `GL/UB` must be known by all the processes. Whenever it is updated, this value should be distributed to all the other processes in order to avoid redundant work. Quite often the parallel management of `GL/UB` is linked to the `GPQ` algorithm. Another component managed by the library is the notion of priority. It could be either by simple evaluation or by node depth evaluation. The architectures of the machines are classified in BoB according to three families: serial, parallel with shared memory and parallel with distributed memory. In serial architecture, the notion of the global priority queue makes use of a particular algorithm in the local priority queue (heap, search tree, and so on). The authors propose three models for each parallel architecture chosen concerning the classification of global priority queues proposed in BoB.

- Asynchronous: Processes execute the operations on local or shared priority queues. They do not cooperate to execute one or many operations. The `GPQ` can be seen as a priority queue with concurrent access (shared memory machines) or as a set of local priority queues with a queue per process and a load balancing strategy (distributed memory machines). Figure 2.16 shows the asynchronous model.

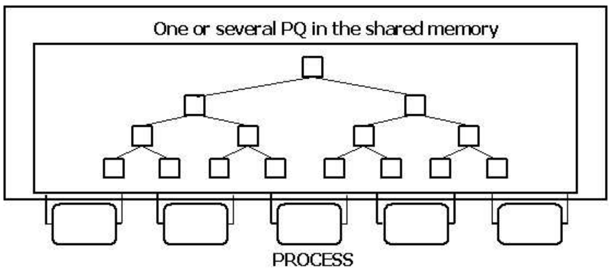


Fig. 2.16 Asynchronous Model

- Synchronous: Processes cooperate to execute a parallel operation, as simultaneous insertions or deletions. The cooperation can exist in each operation. In the delete operation in particular, processes can handle the deletion of nodes with the highest priority in the GPQ. Figure 2.17 shows the synchronous model.

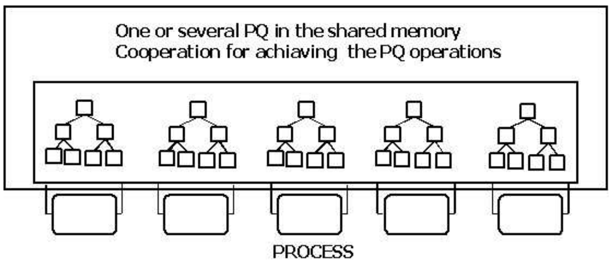

One or several PQ in the shared memory
Cooperation for achiaving the PQ operations
PROCESS

Fig. 2.17 Synchronous Model

- Farm of processes: In this model, one process (server) is specialized in the management of the GPQ. The client processes do the search and communicate with this server to obtain workload (nodes). Figure 2.18 shows the farm model.

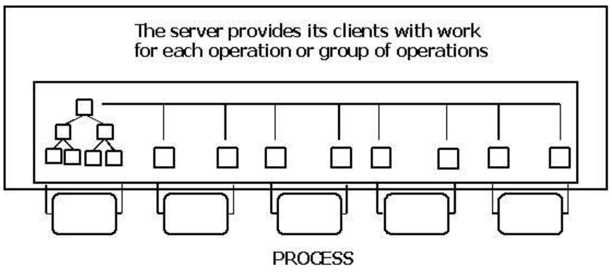

The server provides its clients with work
for each operation or group of operations
PROCESS

Fig. 2.18 Farm Model

These parallelizations are different from each other not only in the function of the architecture of the target machine but also in the function of the semantics attached to the delete operation. In certain cases, the deleted node is not the best-valued node in the whole GPQ.

2.4 PICO - LIBRARY

PICO (Parallel Integer and Combinatorial Optimize) [32] is another C++ framework for implementing general parallel Branch-and-Bound algorithms. This tool was developed at Rutgers University of Piscataway, New Jersey, and Sandia National Laboratories, Albuquerque, in 2000. It consists of two layers, the serial layer and the parallel layer. The serial layer provides an object-oriented mode of describing Branch-and-Bound algorithms, with essentially no reference to parallel implementation. For users uninterested in parallelism, or simply in the early stages of algorithm development, the serial layer allows Branch-and-Bound methods to be described and run in a familiar serial development environment. The parallel layer contains the code necessary to create parallel versions of serial applications. To parallelize a Branch-and-Bound application developed with the serial layer, the user simply defines new classes derived from both the serial application and the parallel layer. The parallel layer is designed using a distributed-memory computation model, which requires message passing to communicate information between processors, and was implemented using the MPI standard. A fully-operational parallel application only requires the definition of a few additional methods for these derived classes, principally to tell PICO how to pack application-specific problems and subproblem data into MPI message buffers, and later unpack them.

The PICO library is not available, so we have not been able access to the implementation of examples.

2.4.1 User Interface

A feature of PICO is that its working is based on `states`. Each subproblem progresses through six states: `boundable`, `beingBounded`, `bounded`, `beingSeparated`, `separated` and `dead`, as illustrates in figure 2.19.

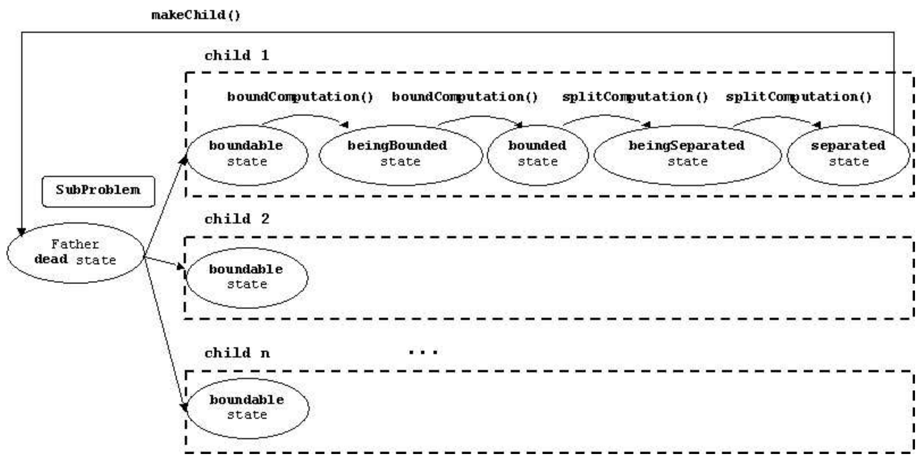


Fig. 2.19 States Transition of a PICO Subproblem

A subproblem always comes into existence in a `boundable` state, meaning that it has associated a bound value, typically inherited from the parent subproblem. Once PICO starts work on bounding a subproblem, its state becomes `beingBounded`, and when the bounding work is finished, the state becomes `bounded`. Once a problem is in the `bounded` state, PICO may decide to branch it, leaving the subproblem in `beingSeparated` state. When separation is complete, the state becomes `separated`, at which point the children of the subproblem may be created. Once the last child has been created the subproblem is in the `dead` state.

Using as an example the binary knapsack problem, figure 2.20 shows the structure of classes of the PICO serial and parallel layers. The `binaryKnapsack` class, derived from `branching`, describes the capacity of the knapsack and the possible items to be placed in it. The class `binKnapSub`, derived from `branchSub`, describes the status of the knapsack items at nodes of the branching tree (i.e. included, excluded, undecided). Each object in a subproblem class `binKnapSub` contains a pointer back to the corresponding instance of the global class `binaryKnapsack`. Through this pointer, each subproblem object can find global information about the Branch-and-Bound problem. To turn a serial application into a parallel application, two new classes must be defined. The first is derived from `parallelBranching` and `binaryKnapsack` (the global class serial application). This is the global parallel class and is called `parallelBinaryKnapsack`. For each problem instance, the information in the global parallel class is replicated on every processor. The second class to be defined is the parallel subproblem class `parBinKnapSub` derived from `binKnapSub` and `parallelBranchSub`. As with the serial subproblems, each instance of `parBinKnapSub` has a pointer to `paralleBinaryKnapsack` that allows it to locate global problem information.

To define a serial Branch-and-Bound algorithm, a PICO user extends two fundamental classes in the serial layer (`branching` and `branchSub`), that are derived from a common base class `picoBase` containing common symbol definitions and run-time parameter definitions. Figure 2.20 shows these heritages.

branching. This class stores global information about a problem instance. The user must define the following methods in this class:

- `readIn()`: Read problem instance data from the command line and/or data file.
- `blankSub()`: Construct an empty subproblem.

branchSub. This class stores data about each subproblem in the Branch-and-Bound tree and contains methods that perform generic operations on subproblems.

- `setRootComputation()`: turns a blank subproblem into the root problem.
- `boundComputation()`: moves the subproblem to the `bounded` state, updating the data bound to reflect the computed value. This method is allowed to pause an indefinite number of times, leaving the subproblem in the `beingBounded` state, but any subproblem will become bounded after some finite number of applications of `boundComputation`.

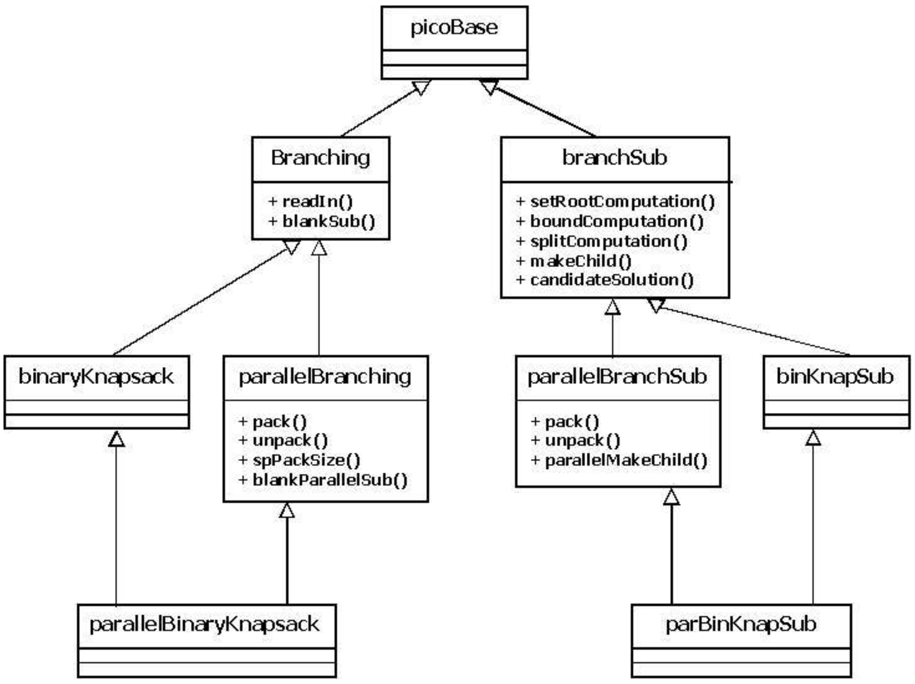


Fig. 2.20 Applying PICO to the Knapsack Problem

- `splitComputation():` The job of this method is similar to the bound-Computation method, but it manages the separation process. It branches a subproblem into the `bounded` or `beingSeparated` state.

- `makeChild():` creates a single child of the subproblem it is applied to, which must be in the separated state.

- `candidateSolution():` returns TRUE if a bounded subproblem does not need further separation.

The parallel layer is formed by two fundamental classes: `parallelBranching` and `parallelBranchSub`, which have the same function as `branching` and `branchSub`, respectively, except that they perform a parallel search of the Branch-and-Bound tree. Both are derived from a common, static base class `parallelPicoBase`, whose function is similar to `picoBase`. Each of `parallel-`

`Branching` and `parallelBranchSub` is derived from the corresponding class in the serial layer.

parallelBranching. Required virtual methods of `parallelBranching` are:

- `pack():` this is responsible for packing global problem information of a specific application into a buffer suitable for message transfer between processors, using the MPI datatype.
- `unpack():` this is responsible for unpacking the same data from an MPI receive buffer.
- `spPackSize():` estimates the maximum number of bytes required to buffer the data from a specific application for a single subproblem.
- `blankParallelSub():` constructs an empty subproblem in a parallel setting.

parallelBranchSub. Required virtual methods of `parallelBranchSub` are:

- `pack():` this is used to pack subproblem data from a specific application into a buffer.
- `unpack():` this is used to unpack subproblem specific application data from a buffer.
- `parallelMakeChild():` constructs a single child from the current subproblem, which must be in the `separated` state. Similar to `makechild` but returns an object of `parallelBranchSub` type.

2.4.2 Parallel Implementation

The processors of the parallel layer of PICO are organized into clusters, each with a hub processor and one or more worker processors. The hub processor serves as a master taking decisions about work allocation, whereas the workers are the slaves, doing the work of bounding and separating subproblems. The degree of control that the hub has over the workers may vary according to the number of run-time parameters, and may not be as tight as a classic master-slave. Moreover, the hub processor has the option of simultaneously functioning as a worker.

Each PICO worker maintains its own pool of active subproblems and all workers use the same pool type. Currently there are three kinds of pool used by PICO: heap sorted by subproblem bound, stack and FIFO queue. If the user specifies the heap pool, PICO will follow a best-first search order; specifying the stack pool results in a depth-first order, and specifying the queue results in a breadth-first order. Each worker processes its pool in the same general manner as the serial layer: it extracts subproblems from the pool and passes them to a search bounding protocol until the pool is empty.

The authors define a `subproblem token` as the only information needed to identify a subproblem, locate it in a secondary pool that resides in the worker, and schedule it for execution. Workers periodically send messages to their controlling hub processor containing blocks of released subproblem tokens with data about the workload in the worker's subproblem pool and other status information. The hub processor maintains a pool of subproblem tokens received from workers. Each time it appreciates a change in workload status from one of its workers, the hub reevaluates the work distribution in the cluster. The hub tries to make sure that each worker has a sufficient quantity of subproblems. Workload quantity evaluation is via a run-time parameter; if a worker appears to have fewer subproblems in its local pool than that indicated by the parameter, the hub judges it deserving of more subproblems. Of the workers that deserve work, the hub designates the one with fewest subproblems as being the most deserving, unless this number exceeds the parameter of workload quantity; in this case, the workers are ranked in inverse order of the best subproblem bound in their pools.

As long as there is a deserving worker and the token pool in the hub is not empty, the hub extracts a subproblem token from its pool and sends it to the most deserving worker. The message sending the subproblem goes directly to the worker that originally released the subproblem, and when that worker receives the token, it forwards the necessary information to the target worker. If the parameter of the workload quantity is set larger than 1, it helps to reduce worker idleness by given each worker a buffer of subproblems to keep it busy while the messages are in transit or the hub is attending other workers.

As each processor has to perform a certain degree of multitasking, PICO defines a thread of control for each required task on a processor, and manages these threads through a scheduler module.

At any given time, each thread is in one of three states: ready, waiting or blocked. Only threads in the ready state are allowed to run. Threads in the waiting state are waiting for the arrival of a particular kind of message, as identified by an MPI tag. The scheduler periodically tests for message arrivals, and changes thread states from waiting to ready as necessary. Threads in the blocked state are waiting for some event other than a message. The scheduler periodically polls these threads by calling their ready virtual methods; when a blocked threads ready method starts returning TRUE, the scheduler changes it back to the ready state.

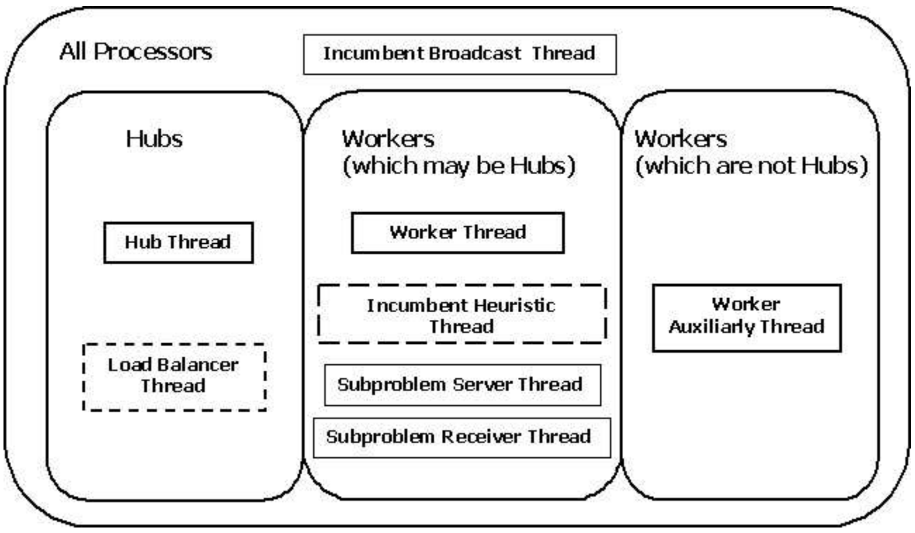

All Processors
Incumbent Broadcast Thread
Hubs
Hub Thread
Load Balancer Thread
Workers (which may be Hubs)
Worker Thread
Incumbent Heuristic Thread
Subproblem Server Thread
Subproblem Receiver Thread
Workers (which are not Hubs)
Worker Auxiliarly Thread

Fig. 2.21 Threads Used by the PICO Core

The threads used by PICO belong to two categories: `message-triggered threads` and `compute threads`. A `message-triggered thread` spends most of its time waiting for messages. When a message with the right tag arrives, the scheduler changes the thread state from waiting to ready. After that, it tries to run soon after they become ready. Once it runs, the thread processes the message, issues a nonblocking receive for another message, changes its state to waiting and returns. `Compute threads` are usually in the ready state, but may be in the blocked state if they have exhausted all their available work. The compute threads are the worker thread and the optional incumbent heuristic thread, and the message-triggered threads are the hub thread, the incumbent broadcast thread, the subproblem server thread, the subproblem receiver thread, the worker auxiliary thread and the load balancer thread. Figure 2.21 groups together all threads used by the PICO Library.

The tasks to accomplish for each of these threads are:

- `The worker thread`. It is present in every worker processor. It extracts subproblems from the local worker pool and sends them to the search handler. If the local pool is empty, the worker thread put state down as blocked. This thread is also responsible for pruning the local subproblem pool and server pool on its processor.

- `Incumbent heuristic thread`. It is an optional compute thread and its task is to search for better incumbent solutions.

- `The hub thread`. It runs on hub processors and listens for messages with the tag, hubTag, from any worker in the system containing workload status information, tokens of subproblems or acknowledgments of receipt of subproblems dispatched from the hub.

- `The incumbent broadcast thread`. It runs on all processors and listens for incumbent broadcast messages. Each processor stores the best objective value known as incumbent and the rank of the processor that generated that value. The incumbent broadcasting procedure organizes all processors into a balanced tree rooted at the initiating processor. The broadcast message contains the objective value of the newly-found incumbent and the processor number of the tree root.

- `The subproblem server thread`. It runs on all workers and listens for work dispatch messages from the hubs. These messages contain a subproblem token and the processor rank of a worker to which the corresponding subproblem should be delivered. The task of this thread is to deliver the full information about the specified subproblem to the worker in question.

- `The subproblem receiver thread`. It is present in all workers and listens for messages with the tag, deliverSPTag. Upon receipt of such a message, the data structures for the subproblem is recreated and it is marked as delivered. If the subproblem cannot be found, the thread inserts it into the local worker pool.

- `The worker auxiliary thread`. This exists only in workers that are not hubs. It listens for messages which are sent by hubs to their workers. Each of these messages can contain one of three possible signals: load information signal, termination check signal o termination signal.

- `The load balancer thread`. It runs on all hub processor. It listens for various kinds of messages, depending on what phase of the load balancing procedure it is currently in. This thread organizes the load balancing scheme and is responsible for terminating the search computation.

2.5 OTHER RELATED TOOLS

This section covers other related tools, like the e-skel Library, A Skeleton Library and COIN-OR library. e-skel library and A Skeleton Library constitute two standard skeleton libraries that try to standardize the skeleton concept. The goal is to add such higher level collective operations to the MPI programmer's toolbox. On the other hand, COIN-OR library is an environment for implementing Branch, Cut and Price algorithms with similar intentions to the previous described libraries.

e-skel Library

eSkel (Edinburgh Skeleton library) [15] is a library of C functions and type definitions which extend the standard C binding to MPI with skeletal operations. Its conceptual model is that of SPMD distributed memory parallelism, inherited from MPI, and its operations must be invoked from within a program which has already initialized an MPI environment. It is most readily understood as an extension to MPI's set of collective operations: both MPI's MPI_Reduce and eSkel's Farm are collective functions which coordinate the execution of instances of a given operation (which the author calls an activity) and associated communications, within the context of a given MPI communicator and process group. The development of eSkel is at a preliminary stage and the task of building a convincing set of examples on top of a robust implementation is the subject of this work.

A Skeleton Library

A Skeleton Library [54, 55] is a tool implemented at the University of Munster, Germany in 2002. This library integrates task and data parallel constructs within an API for C++. Data parallelism is based on a distributed data structure or several of them. This data structure is manipulated by operations (like map and fold) which process it as a whole and which happen to be implemented in parallel form internally. These operations can be interwoven with sequential computations working on non-distributed data. In fact, the programmer views the computation as a sequence of parallel operations. Conceptually, this is almost as easy as sequential programming. The skeleton library offers pipelines, farms and parallel composition. In a farm, a farmer process accepts a sequence of inputs and assigns each of them to one of several workers; the parallel composition works in a similar way to the farm, but each input is forwarded to every worker. Each parallel skeleton task has the same property as an atomic process, namely it accepts a sequence of inputs and produces a sequence of outputs. This allows the parallel skeleton tasks to be arbitrarily nested. In an application, each stage could internally use data parallelism resulting in a two tier model, where the computation is first structured by task parallel skeletons like the pipeline and where atomic task parallel computations can be parallel data. The Skeleton Library has been implemented on top of MPI.

COIN-OR - Library

SYMPHONY (Single- or Multi-Process Optimization over Networks) [57, 75] is an environment for implementing Branch, Cut, and Price algorithms. SYMPHONY is a state-of-the-art solver which is designed to be completely modular and is easy to port to various problem settings. All library subroutines are generic (their implementation does not depend on the problem-setting). To develop a full-scale, parallel Branch, Cut, and Price (BCP) algorithm, the user has only to specify a few specific functions of the problem such as preprocessing and separation. SYMPHONY communicates with the user's routines through well-defined interfaces and performs all the normal functions of branch and cut: tree management, LP solution, cut pool management, as well as interprocess communication. Although there are default options, the user can also assert control over the behavior of SYMPHONY through a huge numbers of parameters and optional subroutines. SYMPHONY can be built in a variety of configurations, ranging from completely sequential to fully parallel with independently functioning cut generators, cut pools, and Linear Programming solvers. SYMPHONY is a framework written in C. The distributed version currently runs only in an environment supported by the PVM message passing protocol. The same source code can be compiled for shared-memory architectures using any OpenMP compiler.

COIN/BCP co-named COIN-OR (Common Optimization INterface for Operations Research) [75] is a framework written in the same spirit as SYMPHONY in C++. Because of their generic, object-oriented designs, both are extremely flexible and can be used to solve a wide variety of discrete optimization problems. However, these frameworks have somewhat limited scalability. The goal of the research team is to address these scalability issues by designing a more general framework called the Abstract Library for Parallel Search (ALPS).

ALPS is a C++ class library upon which a user can build parallel algorithms for performing tree search. To support the implementation of parallel BCP (branch, constrain, and price), they have designed two additional C++ class libraries built on top of ALPS. The first, called the Branch, Constrain, and Price Software (BiCePS) library, implements a generic framework for relaxation-based Branch-and-Bound. In this library, they make very few assumptions about the nature of the relaxations, i.e., they do not have to be linear programs. The second library, called the BiCePS Linear Integer Solver (BLIS), implements LP-based Branch-and-Bound algorithms, including BCP.

2.6 COMPARISON TABLE OF ANALYZED TOOLS

Tool	C	C++	MPI	PVM	OpenMP	Object oriented	Automatic load balancing	Availability
PPBB	X			X			X	X
PUBB	X	X		X		X	X	
Bob++	X	X		X				
PICO		X	X			X		

Table 2.1 Comparison between the Studied Tools

The table 2.1 shows a summary of the main features of each one of libraries analyzed in previous sections. Columns 2 and 3 represent the programming language; as we can appreciate, most of the libraries are written in C++, except PPBB that was written in C . Columns 4 and 5 represent parallel programming based on message passing and column 6 is based on shared memory; PPBB, PUBB and Bob++ were implemented using PVM and only PICO was implemented under MPI, none of them is available on OpenMP. Column 7 shows which of them present an oriented object methodology. Column 8 shows which of the libraries provide load balancing strategies. Finally, in the column 9, it can be appreciated that only the PPBB Library is available.

3
MaLLBa::BnB - *Library*

3.1 INTRODUCTION

This chapter presents our proposal, a skeleton to solve optimization problems with the Branch-and-Bound technique (MaLLBa::BnB). The implementation of the skeleton has been made in C++. Sequential code and parallel code of the invariant part of the solvers is provided for this paradigm. The user must fill a blueprint in order to solve his/her problem. The classes that compose this blueprint serve to establish the relation between the main solver and the problem. Once the user has represented the problem, he/she obtains a parallel solver without any additional effort. The skeleton provides modularity for the design of exact algorithms, which supposes a great advantage with respect to the direct implementation of the algorithm, not only in terms of code re-usability but also in methodology and clarity of concepts.

MaLLBa::BnB is part of the MaLLBa project [3, 4, 5]. In the accomplishment of MaLLBa the Spanish Universities of La Laguna, Polytechnic of Catalonia and Málaga have participated. The MaLLBa project is an effort to develop an integrated library of skeletons for combinatorial optimization (including exact, heuristic and hybrid techniques) dealing with parallelism in a user-friendly and, at the same time, efficient manner. Its three target environments are sequential computers, LANs of workstations and WANs (local and wide area networks, respectively). The main features of MaLLBa are: integration of all the skeletons under the same design principles, facility to switch from sequential to parallel optimization engines, and cooperation among solvers to provide more powerful hybrid skeletons, ready to use on different machines. Clusters of PC under Linux are currently supported, and the resulting software architecture is flexible and extensible (new skeletons can be added, alternative communication layers can be used, etc.).

In MaLLBa each resolution technique is encapsulated into a skeleton. At present, the following skeletons are available: Branch-and-Bound (BnB), Divide-and-Conquer (DC), Dynamic Programming (DP), Hill Climbing, Metropolis, Simulated Annealing (SA), Tabu Search (TS) and Genetic Algorithms (GA). Exact techniques have been integrated to solve particular problems such as DC+BnB. Moreover hybrid techniques have been implemented combining the previous skeletons, e.g., GA+TS, GA+SA.

To our knowledge, there is no other work that considers, together, all these classes of algorithms and problems, and that also extends the analysis to LAN environments.

In this chapter we first present the user interface of MaLLBa in Section 3.2. The required classes in a MaLLBa skeleton are presented in Section 3.3 and an example of instantiation of these classes is presented in Section 3.4. Section 3.5 shows the classes provided by a MaLLBa skeleton. In Section 3.7 we present the strategy used to implement the parallel skeleton, and in Sections 3.7.1, 3.7.2 and 3.7.3 we present the centralized, distributed and shared memory schemes that we implement to parallelize the MaLLBa::BnB skeleton, respectively.

3.2 MaLLBa USER INTERFACE

In the MaLLBa skeletons world the following users appear (see figure 3.1):

- Skeleton programmer. A person who designs and implements the skeleton. He or she decides the set of classes the skeleton offers and requires.
- Skeleton filler. A person who adapts the real problem and the heuristic to the skeleton classes filling the details left by the skeletons programmer.
- Final user. Uses the fulfilled skeletons to get an approximated solution for an instance.
- Skeleton combinator.

The skeletons are specially targed at two different objectives:

- To provide the final user with a friendly environment sharing their common knowledge.

- To simplify the programming of the skeleton filler.

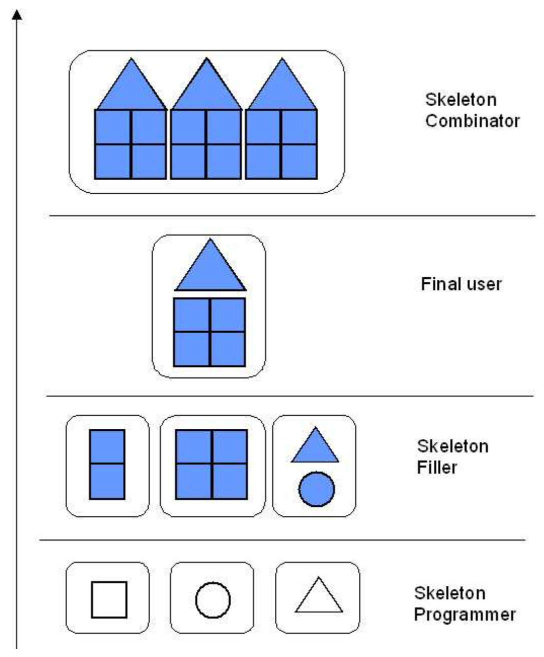


Fig. 3.1 Skeleton Users

In a MaLLBa skeleton two principal parts are distinguished: One that implements the *resolution pattern* provided by the library and a part which the user has to complete with the particular characteristics of the *problem to solve* and that will be used by the resolution pattern [5]. There is an intuitive relationship between the participating entities in the resolution pattern and the classes which will be implemented by the user.

The part provided by the skeleton, that is, the resolution pattern, is implemented through classes. These classes are denominated *provided classes* and appear in the code with the qualifier *provides*. The part which the user fills in

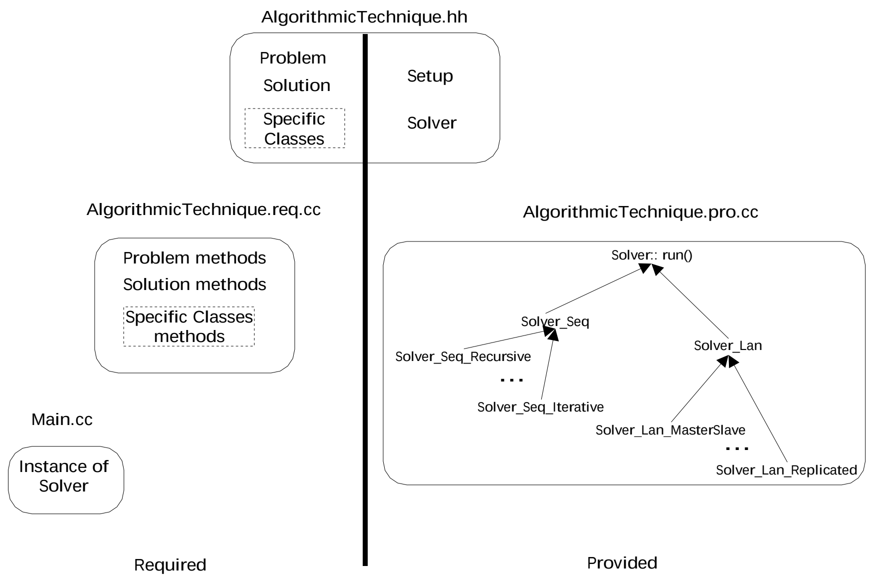


Fig. 3.2 MaLLBa Skeletons Interface

with their particular problem is implemented through classes labelled with the qualifier *requires*, and will be named *required* classes.

The adjustment that has been accomplished by the user consists of two steps. First, the problem has to be represented through data structures and then, using them, the user has to implement the required functionalities of the classes. These functionalities will be invoked from the particular resolution pattern (because the interface for such classes is known) so that, when the application has been completed, the expected functionalities applied to the particular problem are obtained.

Figure 3.2 presents the structure of MaLLBa skeletons. In the file with extension "`.hh`" the required and provided classes are declared. The file "`.pro.cc`" contains the C++ implementations of the resolution patterns, while in the file "`.req.cc`" the user has to implement the required functionalities. As a particular case, figure 3.3 shows a UML scheme of MaLLBa for the Branch-and-Bound technique.

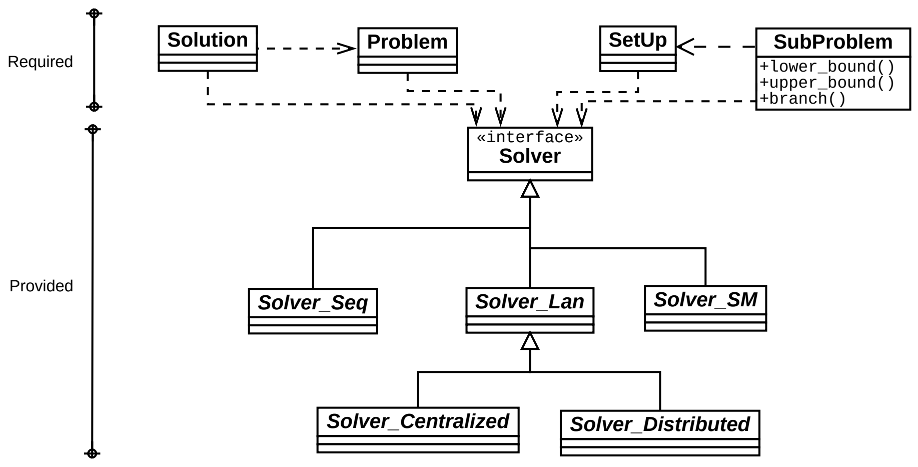


Fig. 3.3 UML Scheme of MaLLBa::BnB

3.3 REQUIRED CLASSES

Required classes are used to store the basic data of the algorithm. The problem, the solution, the states of the search space and the input/output are represented by them. All MaLLBa skeletons have to define the following classes:

- `Problem`: defines the minimal standard interface to represent a problem.
- `Solution`: defines the minimal standard interface to typify a solution.

Furthermore each resolution pattern requires a set of specific classes depending of the algorithmic technique.

Branch-and-Bound algorithms divide the solutions space step by step and calculate a bound value of those solutions that could be found forward. If the bound shows one of these solutions is worse than the best solution found until then, it is not necessary to continue exploring this part of the solution space. The class `Subproblem` will be used to represent this process in a MaLLBa::BnB skeleton (figure 3.4). Furthermore the user must specify in the definition of the `Problem` class whether the problem to solve is a maximization or minimization problem.

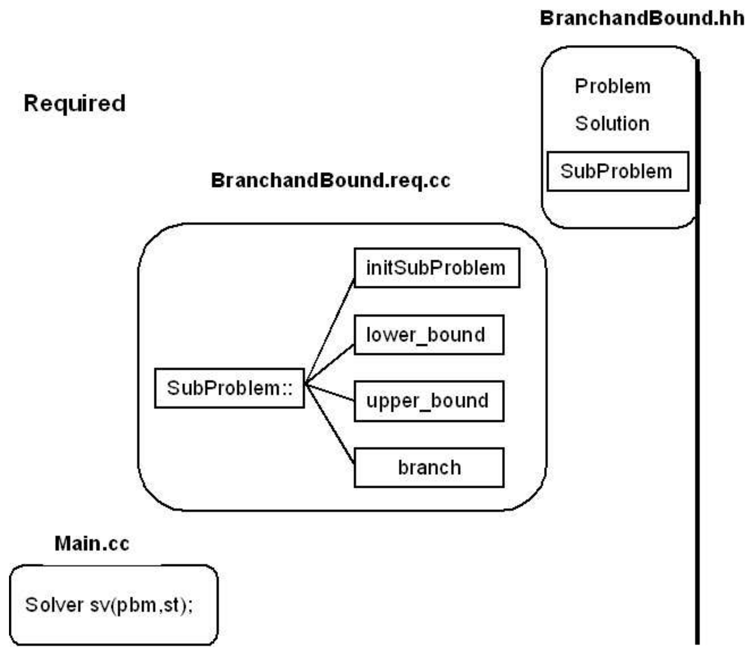


Fig. 3.4 Required Classes for Branch-and-Bound skeleton

- The class `Subproblem`: represents the area of non explored solutions. This class must provide the following functionalities:

  - `initSubProblem(pbm)`: generates the first subproblem from the original problem.

  - `branch(pbm, sps)`: it generates from the current subproblem the subset of subproblems to be explored.

  - `lower_bound(pbm,sol)`: this method calculates a lower bound of the objective function for a given problem.

  - `upper_bound(pbm,sol)`: this function calculates an upper bound of the objective function that would be obtained for a problem.

3.4 AN EXAMPLE: THE 0-1 KNAPSACK PROBLEM

This section shows the interface of required classes of the MaLLBa::BnB skeleton [25, 29]. The instantiation of the skeleton classes for a concrete problem consists of two steps: (i) choosing the data types for representing the members of the class (in a `.hh` file) and, (ii) implementing the methods of the class according to the chosen data types (in a `.req.cc` file).

The required class `Problem` represents an instance of the problem to be solved. The internal implementation of the class only needs to create, serialize and obtain the direction. Note that the serialization is important when an object of the class has to be sent to other processes in a parallel distributed execution.

The 0-1 Knapsack Problem [69] is the example chosen. In this problem, as we defined in chapter I, a subset of N given items has to be introduced in a knapsack of capacity C. Each item has a profit p_i and a weight w_i. The problem is to select a subset of items whose total weight does not exceed C and whose total profit is maximum. This is a maximization problem. We need to represent the capacity, the number of elements and the profits and weights. The profits and weights are represented using the class `vector` of the STL (Standard Template Library) [40]. We use `push_back` to insert each value of the profits and weights at the end of the data structures `p` and `w`. The `problem` can be represented as follows:

```
requires class Problem {
  public:
    Number C,         // capacity
           N;         // number of elements
    vector<Number> p, // profits
                   w; // weights
    Problem ();       // constructor
    ~Problem ();      // destructor
    inline Direction direction() const { return Maximize;}
    ...
    friend opacket& operator<< (opacket& os, const Problem& pbm);
    friend ipacket& operator>> (ipacket& is, Problem& pbm);
};
```

The required class `Solution` represents a feasible solution to the problem. It is represented by a vector containing true or false values depending on whether the object belongs to the solution or not:

```
requires class Solution {
  public:
    vector<bool> s;   // solution vector
    Solution ();      // constructor
    ~Solution ();     // destructor
    ...
    friend opacket& operator<< (opacket& os, const Solution& sol);
    friend ipacket& operator>> (ipacket& is, Solution& sol);
};
```

The required class `SubProblem` represents a partial problem. A subproblem is determined by the following data: the current capacity, the next object to be considered, the current profit and the current solution:

```
requires class SubProblem {
  public:
    Number CRest,      // current capacity
           obj,        // next object
           profit;     // current profit
    Solution sol;      // current solution
    SubProblem ();     // constructor
    ~SubProblem ();    // destructor
    ...
    friend opacket& operator<< (opacket& os, const SubProblem& sp);
    friend ipacket& operator>> (ipacket& is, SubProblem& sp);
    void initSubProblem (const Problem& pbm);
    Bound upper_bound (const Problem& pbm, Solution& ls);
    Bound lower_bound (const Problem& pbm, Solution& us);
    void branch (const Problem& pbm, branchQueue<SubProblem>& subpbms);
};
```

This class must also provide the following functionalities:

- The method `initSubProblem()`, that is, the first subproblem. In the knapsack problem, the value of `CRest` is initialized to the total capacity, and the current `profit` to zero because no object has been selected.

  ```
  void SubProblem::initSubProblem (const Problem& pbm) {
    CRest = pbm.C;
    obj = 0;
    profit = 0;
  }
  ```

- The `branch()` method. Given a subproblem, two new ones are generated based on the decision to include the next object in the knapsack or not:

```
void SubProblem::branch (const Problem& pbm,
                         container<SubProblem>& subpbms) {
  SubProblem spNO = SubProblem(CRest,obj+1,profit);
  subpbms.insert(spNO);
  Number newC = CRest - pbm.w[obj];
  if (newC >= 0) {
    SubProblem spYES = SubProblem(newC,obj+1,(profit+pbm.p[obj]));
    subpbms.insert(spYES);
} }
```

- The `lower_bound(pbm,sol)` and `upper_bound(pbm,sol)` methods calculate a lower and upper bound, respectively, of the objective function. The *lower_bound* is calculated by including objects in the knapsack until the current capacity is reached:

```
Bound SubProblem::lower_bound (const Problem& pbm, Solution& us) {
  Bound weight, pft;
  Number i, tmp; us = sol;
  for(i = obj, weight = 0, pft = profit; weight <= CRest; i++){
    weight += pbm.w[i];
    pft += pbm.p[i];
    us.s.push_back(true);
  }
  i--;
  weight -= pbm.w[i];
  pft -= pbm.p[i];
  us.s.pop_back();
  tmp = pbm.N - us.s.size();
  for (Number j = 0; j < tmp; j++)
    us.s.push_back(false);
  return(pft);
}
```

The *upper_bound* is calculated in a similar way, but in this case a portion of the last object considered is included to meet the capacity:

```
Bound SubProblem::upper_bound (const Problem& pbm) {
  Bound upper, weight, pft;
  Number i;
  for(i=obj,weight=0,pft=profit;
  weight<=CRest; i++) {
    weight += pbm.w[i];
    pft += pbm.p[i];
  }
  i--;
  weight -= pbm.w[i];
  pft -= pbm.p[i];
  upper = pft + (Number)((pbm.p[i] * (CRest - weight))/pbm.w[i]);
  return(upper);
}
```

3.5 PROVIDED CLASSES

The provided classes are those which the user has to call when using a skeleton. Theses classes are: Setup and Solver.

- The Setup class groups the configuration parameters of the skeleton. For example, this class specifies whether the search of the solution area will be depth-first, best-first or breadth-first. It also specifies whether the scheme must be centralized, distributed or shared memory.
- The Solver class implements which strategy to follow, and maintains updated information concerning the state of the exploration during the execution:

```
provides class Solver {
  protected:
    Direction dir;                  // maximum or minimum
    branchQueue<SubProblem> queue;  // search space
    Setup st;                       // search method
    const Problem&  pbm;            // problem
    Solution sol;                   // solution
    SubProblem sp;                  // partial problems
    Bound bestSol, high, low;       // partial values
  public: ...
};
```

The execution is carried out through a call to the run() method. This class is specified by Solver_Seq, Solver_Lan and Solver_SM in the class hierarchy (see figure 3.3). In order to choose a given resolution pattern, the user must instantiate the corresponding class in the main() method. The following code (see figure 3.5) shows an example of the application of a sequential solver for the Branch-and-Bound skeleton. Line 2 indicates that a Branch-and-Bound skeleton will be used. An instance of the sequential skeleton (Solver_Seq) is created in line 7. Calling the run() method starts the execution of the skeleton. The bestSolution() and solution() methods provide the best objective value and the solution, respectively.

An innovation of our approach is that it allows the combined use of the skeletons. In this case, as some members of the parallel group of the La Laguna University had implemented the sequential and parallel MaLLBa skeleton for Divide-and-Conquer technique [77], we have collaborated with them to integrate MaLLBa::BnB and MaLLBa::DnC skeletons. In chapter I we commented on the Branch-and-Bound algorithm described by Martello and Toth [69] to

```
 int main (int argc, char** argv) {
   using skeleton BranchAndBound;
   Problem pbm;
   Solution sol;
   Bound bs;

   Solver_Seq sv(pbm);     //Instance of sequential skeleton
   sv.run();               //execution
   bs = sv.bestSolution(); //obtaining the best solution
   sol = sv.solution();    //taking the solution
}
```

Fig. 3.5 MaLLBa::BnB main Method

solve the knapsack problem which requires the elements to be organized in descending order according to the profit/weight relation given by:

$$\frac{p_i}{w_i} \geq \frac{p_{i+1}}{w_{i+1}} \quad (i = 1, ..., N-1)$$

The idea consists of applying the Divide-and-Conquer skeleton to order the objects according to their previous relationship, and then apply the Branch--and-Bound skeleton to solve the problem. This integration of skeletons provided us some publications presented in International Conferences [28, 26, 27].

The figure 3.6 presents the `main()` method that integrates the sequential skeletons for Divide-and-Conquer and Branch-and-Bound techniques. Line 9 indicates an instance of the sequential skeleton for the QuickSort method will be created. Calling the `run()` method starts the execution of the Divide-and-Conquer skeleton in line 10. The solution to the order problem is stored in variable `ssol` in line 11. Once the objects have been ordered, an instance of the Branch-and-Bound sequential skeleton is created in line 19. Methods `bestSolution()` and `solution()` provide the best objective value and the solution to the knapsack Problem, respectively in lines 21-22. The complete code of the required classes of Divide-and-Conquer Skeleton for the QuickSort method is presented in Appendix B.

```
int main (int argc, char** argv) {
  Knapsack::Problem pbm;
  Knapsack::Solution ksol;
  Knapsack::Bound bs;
  ...
  Knapsack::Problem opbm;  //ordered problem
  QuickSort::Solution ssol;
  ...
  QuickSort::Solver_Seq svs(pbm);
  svs.run();
  ssol = svs.solution();
  ...
  opbm.setN(pbm.N);
  opbm.setCapacity(pbm.C);
  for (Knapsack::Number i = 0; i < n ; i++) {
    opbm.setWeight(pbm.w[ssol.l[i]]);
    opbm.setProfit(pbm.p[ssol.l[i]]);
  }
  Knapsack::Solver_Seq svk(opbm, st);
  svk.run();
  bs = svk.bestSolution();
  ksol = svk.solution();
  ...
}
```

Fig. 3.6 main Method Used During the Integration of MaLLBa::DnC and MaLLBa::BnB

3.6 THE SEQUENTIAL ALGORITHM FOR THE MaLLBa::BnB SKELETON

3.6.1 The Data Structure

The data structure implemented is a dynamic queue to control the subproblems flow produced by the `branch` method. The subproblems are inserted or extracted depending on the search method specified in class setup: `lifo` (last-in-first-out), `fifo` (first-in-first-out) or priority. Each subproblem is linked through two pointers to the previous element of the queue and to the next element of the queue respectively. Figure 3.7 shows the doubly linked queue. The complete code of the data structure used is presented in Appendix A.

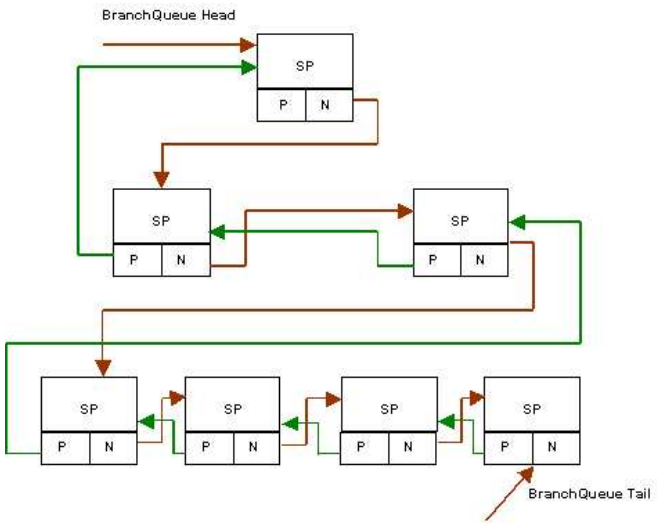


Fig. 3.7 Subproblems Queue of MaLLBa::BnB

3.6.2 The Algorithm

As we said in Chapter I, during the MaLLBa::BnB computation, *subproblems* are successively generated and tested. Given a *subproblem* Π_i, it can be decomposed into $\Pi_{i1}, \Pi_{i2}, ..., \Pi_{ik}$ by a branching operation where $S_i = \bigcup_{j=1}^{k} S_{ij}$. Thus any feasible solution $\sigma \in S_i$ belongs to some S_{ij} and conversely any $\sigma \in S_{ij}$ belongs to S_i. Let $\mathcal{Q}$ denote the set of *subproblems* currently generated. A *subproblem* $\Pi_i \in \mathcal{Q}$ that is neither decomposed nor yet tested is called *live*. The set of *live subproblems* are denoted by $\mathcal{L}$. For each tested subproblem in $\mathcal{Q}$ its *lower bound* and *upper bound* are computed. The greatest lower bound obtained so far is called the *best solution value* and denoted by *bs*. The solution realizing *bs* is called the *best solution* and stored in $\mathcal{T}$.

Figure 3.8 shows the algorithm of the MaLLBa::BnB technique for a maximization problem. It proceeds by repeating the test of live subproblems. The selection of a live subproblem for the next test is done by a *search function s*, such that, $s(\mathcal{L}) = \mathcal{L}$. Departing from a problem and a subproblem the use of the Branch-and-Bound technique is considered to find the best solution (bestSol). The first subproblem is extracted from the queue (line 4), its upper and lower bounds are calculated and supposing it improves the bestSol it is updated (lines 5-10). If the problem is solved, it continues with the process of extract-

```
1   $\mathcal{L} := \{\Pi_0\}$; $\mathcal{Q} := \{\Pi_0\}$;
2   $bs := -\infty$; $\mathcal{T} := \emptyset$;
3   while ($\mathcal{L} \neq \emptyset$) {
4       $\Pi_i := s(\mathcal{L})$;
5         if ($upper\_bound(\Pi_i) > bs$) {
6           if ($lower\_bound(\Pi_i) > bs$) {
7               $bs := lower\_bound(\Pi_i)$;
8               $\mathcal{T} := \{\sigma\}$;
9               // $\sigma$ satisfies $f(x, \sigma) = lower\_bound(\Pi_i)$;
10          }
11          else {
12              *(brach)* decompose $\Pi_i$ into $\Pi_{i1}, \Pi_{i2}, ..., \Pi_{ik}$
13              $\mathcal{L} := \mathcal{L} \cup \{\Pi_{i1}, \Pi_{i2}, ..., \Pi_{ik}\}$ - $\{\Pi_i\}$ ;
14          }
15          $\mathcal{Q} := \mathcal{Q} \cup \{\Pi_{i1}, \Pi_{i2}, ..., \Pi_{ik}\}$ ;
16      }
17      $\mathcal{L} := \mathcal{L}$ - $\Pi_i$;
18  }
19  return $bs$ // *bs is the best solution value*
20  return $\mathcal{S}$ // *$\mathcal{S}$ is the best solution*
```

Fig. 3.8 Algorithm of the MaLLBa::BnB Technique (Maximization Case)

ing problems from the queue and proving the bestSol. If the problem is still unresolved it is branched and the branch method inserts the new subproblems into the queue (lines 12-13).

Figure 3.9 shows the pseudo-code of the MaLLBa::BnB sequential skeleton for a maximization problem. We have followed an iterative approach for the design of the sequential skeleton. In order to implement a skeleton it is necessary that the code be generic and not dependent on the implemented problem, as occurs in the case of recursive algorithms. For this reason we have used iterative algorithms. Line 3 shows the data structure `bqueue` explained in the previous subsection.

As an example of the recursive code to solve the 0/1 Knapsack Problem, the algorithm shown in figure 3.10 is presented. The number of objects to insert into the knapsack is stored in the variable N, the variables w and p store the weights and the benefits of each of them, while the variable M represents the capacity. The variable that stores the best solution found until that moment is `bestSol` (line 3). In line 10 the bound function (`lowerUpper`) is called.

```
1  Bound BB (const Problem& pbm, const SubProblem& sp,
2           Solution& sol){
3    branchQueue<SubProblem> bqueue;
4    Solution sl;
5    Bound high, low;
6    SubProblem sp;
7    bqueue.insert(sp);
8    while (!bqueue.empty()) {
9      sp = bqueue.remove();
10     high = sp.upper_bound (pbm, sl);
11     if (high > bestSol){
12       low = sp.lower_bound(pbm, sl);
13       if (low > bestSol) {
14          bestSol = low;
15          sol = sl;
16       }
17       if (high != low) sp.branch(pbm, bqueue);
18     }
19   }
20   return(bestSol);
21 }
```

Fig. 3.9 Pseudo-code of the MaLLBa::BnB Sequential Skeleton

This function calculates the lower and upper bounds. The `knap` function (lines 5-20) implements a recursive Branch-and-Bound algorithm. The function call that studies the insertion of the object k, is accomplished in line 16, while line 17 considers not inserting it. The condition to update the value of the best solution (`bestSol`) found until that moment is implemented between lines 11 to 13.

```
number N, M;
number w[MAX], p[MAX];
number bestSol = -INFINITY;

number knap(number k, number C, number P) {
  number L,  // lower_bound
         U,  // upper_bound
         next;
  if (k < N) {
    lowerUpper(k,C,P,&L,&U);
    if (bestSol < L) {
      bestSol = L;
    }
    if (bestSol < U) { /* L <= bestSol <= U */
      next = k+1;
      knap(next, C - w[k], P + p[k]);
      knap(next, C, P);
  } }
  return bestSol;
}
```

Fig. 3.10 Pseudo-code of a Sequential Recursive Branch-and-Bound Algorithm

3.7 PARALLEL ALGORITHMS FOR THE MaLLBa::BnB SKELETON

To parallelize our skeleton for the Branch-and-Bound technique we have implemented a master-slave strategy. This strategy can be defined as follows: A master processor holds all the information about the states space, which contains a number of non branching subproblems on the enumeration tree and also stores the incumbent solution. The master assigns a subproblem to an idle slave and receives new subproblems generated from that slave. There are several slaves that evaluate subproblems and generate new ones if necessary. An idle slave receives a subproblem only when there are subproblems that are likely to lead to an optimal solution. This strategy offers significant merits, such as the ability to easily control and parallelize Branch-and-Bound algorithms in a more natural and straightforward way. A MaLLBa::BnB skeleton is composed of the following basic processes:

- Master process: There is only one master process in the skeleton. It contains all the information about the states space and the status of each slave process, e.d., busy or idle.

- Slave process: the number of slave processes is specified in an initiation parameter. Each `slave process` performs all the computations to solve/evaluate a subproblem.

The parallelization has be done using the architecture shown in Figure 3.11. The Master has a queue of unsolved problems. It removes a subproblem from its queue and assigns it to the first idle slave in phase *a*. In phase *b*, the slave branches the subproblem into two subproblems and sends the new generated subproblems to the Master. In phase *c*, the Master receives these subproblems and inserts them into its queue. Finally, the Master removes as many subproblems as is possible and sends them to idle slaves in phase *d*. The process finishes when the Master queue is empty and the best solution has been found.

In the following subsections we explain, in detail, how the MaLLBa::BnB skeleton has been parallelized. Three different implementations are provided, two of them based on the Master-Slave paradigm: one centralized and another distributed, and the third one based on the shared memory paradigm.

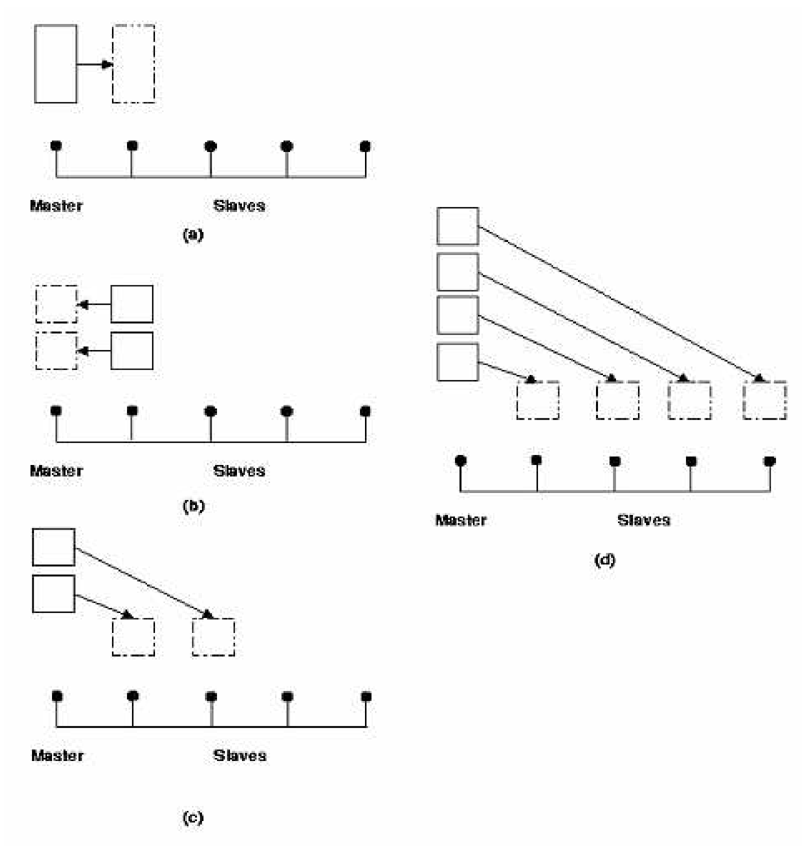


Fig. 3.11 Phases of the Master-Slave Paradigm

3.7.1 Centralized Parallel Scheme

Centralized Data Structure

The data structure for a centralized parallel scheme is represented in figure 3.12. Each one of data structures used in this scheme, follows the same structure used by the sequential case, e.g., they are doubly linked queues. In this case, the master processor has its own global queue of unsolved problems, from where it removes subproblems to send to the idle slaves. Each slave that receives a subproblem, branches it and stores the new generated subproblems in its own local queue. Each slave sends its local queue to the Master, who links the global queue with the queue received from the slave. To link both queues *concatqueue* method presented in Appendix A is used.

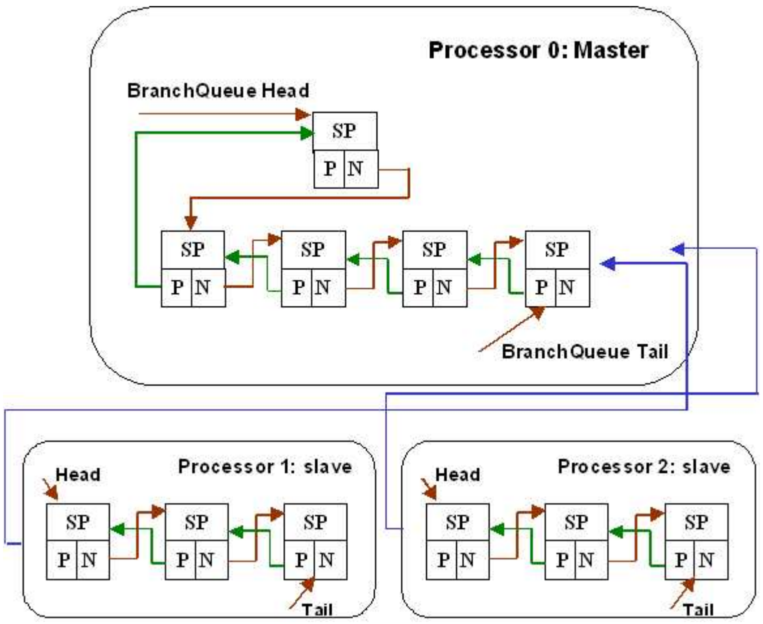


Fig. 3.12 Centralized Data Structure

Centralized Algorithm

This section describes a general parallel pseudo-code [37] for solving a maximization problem using the Branch-and-Bound technique. We present the design and implementation of the MaLLBa::BnB centralized message passing resolution pattern [26]. The scheme presented is a centralized scheme in the sense that it is the Master who distributes the jobs.

The algorithm in figure 3.13 shows the tasks to be accomplish by the Master. The Master is responsible of the distribution of the subproblems between the slaves. The Master has a local queue $\mathcal{L}$ to store the generated subproblems and a data structure $\mathcal{I}$ which registers the occupation state of each slave (line 1);

```
$\mathcal{P} := \{1, ..., p\}$; $\mathcal{I} := \mathcal{P}$;
$bs := -\infty$; $\mathcal{T} := \emptyset$;
while $((\mathcal{L} \neq \emptyset)$ and $(\mathcal{I} \neq \mathcal{P}))$ {
    while $((\mathcal{L} \neq \emptyset)$ or $(\mathcal{I} > 0))$ {
        $\Pi_i := s(\mathcal{L})$;
        $i := s(\mathcal{I})$; $\mathcal{I} := \mathcal{I}$ - $\{i\}$;
        $send(\Pi_i, bs, \mathcal{T}, i)$;
    }
    $receive(q, flag) / q \in \mathcal{P}$;
    while $(flag)$ {
        if ($flag$ =SOLVED) {
            $receive(q, bs, \mathcal{T})$;
        }
        if ($flag$ =BNB) {
            $receive(q, h, bstemp)$;
            if $(h > bs$ ) {
                $bs := bstemp$
                $send(\mathcal{L}\_request, q)$
                $receive(q, \Pi_{i1}, \Pi_{i2}, ..., \Pi_{ik})$
            }
            else
                $send(\mathcal{L}\_delete, q)$;
            $\mathcal{I} := \mathcal{I} \cup \{q\}$
        }
        $receive(q, flag) / q \in \mathcal{P}$;
    }
    $\mathcal{L} := \mathcal{L} \cup \{\Pi_{i1}, \Pi_{i2}, ..., \Pi_{ik}\}$
}
$send($END$, i)$; $\forall i \in \mathcal{P}$
```

Fig. 3.13 Centralized Master Algorithm

```
while ($true$) {
      $receive(q, flag)/q \in \mathcal{P}$;
      while ($flag$) {
        if ($flag$ =END) {
          $receive(Master$,END);
          return;
        }
        if ($flag$ =PBM) {
          $receive(q, \Pi_i, bs_i, \mathcal{T}_i)$;
          $\mathcal{L} := \{\Pi_i\}$; $\mathcal{Q} := \{\Pi_i\}$; $bs := bs_i$; $\mathcal{T} := \mathcal{T}_i$;
          if ($upper\_bound(\Pi_i) > bs$) {
            if ($lower\_bound(\Pi_i) > bs$) {
              $bs := lower\_bound(\Pi_i)$;
              $\mathcal{T} := \{\sigma\}$; / $f(x, \sigma) = lower\_bound(\Pi_i)$;
            }
          }
          if (SOLVED) {
            $send(bs, \mathcal{T}$,SOLVED$, Master)$;
          }
          else {
            $send(bs, upper\_bound(\Pi_i)$, BNB, $Master)$;
          }
          $receive(Master, \mathcal{L}\_request)$;
          if ($\mathcal{L}\_request$ ) {
            *(brach)* decompose $\Pi_j$ into $\Pi_{j_1}, \Pi_{j_2}, ..., \Pi_{j_k}$
            $\mathcal{L} := \mathcal{L} \cup \{\Pi_{j_1}, \Pi_{j_2}, ..., \Pi_{j_k}\}$ - $\{\Pi_j\}$;
            $\mathcal{Q} := \mathcal{Q} \cup \{\Pi_{j_1}, \Pi_{j_2}, ..., \Pi_{j_k}\}$;
            while ($\mathcal{L} \neq \emptyset$) {
              $\Pi_n := s(\mathcal{L})$; $\mathcal{L} := \mathcal{L} - \Pi_n$;
              $send(\Pi_n, Master)$;
            }
          }
        }
        $receive(q, flag)/q \in \mathcal{P}$;
      }
}
```

Fig. 3.14 Centralized Slave Algorithm

at the beginning all the slaves are idle. The Master extracts as subproblems as be possible from its queue and sends them to the idle slaves, while there are idle slaves (lines 4-8). While there are idle slaves and the local queue is not empty, the Master receives information from them and decides the next action to apply depending on whether the problem is solved (line 11) or if there are new subproblems to generate (line 14). If the problem is solved, the solution is received and stored (line 12). When the master receives the upper bound value (line 15), if the upper bound value is better than the actual value of the best solution, the reply to the slave includes the request for newly generated subproblems after branch (line 18). Then, it receives from the slave these subproblems that will be inserted into the local queue (line 27). Alternatively, the answer indicates that it is not necessary to work in this subtree (line 22). When the number of idle slaves is equal to the initial value and the local queue is empty, the search process finishes, and the Master notifies the slaves to end work (line 29).

A slave (see figure 3.14) receives problems, best solution values or an ending signal. When a slave receives a problem (line 9), it calculates the lower and the upper bounds, and if the problem is not solved, sends the Master the upper bound and best solution calculated (line 21). If the slave receives from the Master the order to branch (line 23), new subproblems are generated calling on the `branch` method (line 25). In this case, it sends its local queue directly to the Master (lines 28-30).

The implementation of the MaLLBa::BnB skeleton shown in figures 3.15 and 3.16 uses MPI_Send and MPI_Recv, to send and receive messages respectively. The main loop in the Master and Slaves codes are implemented using MPI_IProbe. When a message is received its status is used to classify what kind of work should be done: finish, receive a problem for bounding and branching, receive a queue request, etc.

```
busy[0] = 1; for i = 1, nProcs { busy[i] = 0; }
idle = nProcs - 1;
auxbqueue; // queue of subproblems sent by the slave
// Insert the subproblem into the initial queue
bqueue.insertAtBack(sp);
// Stop condition: empty queue and all the slaves are idle
while ((!bqueue.empty()) || (idle < groupSize)) {
  // Send all possible subproblems, the best solution and
  // the solution to the slaves
  while ((!bqueue.empty()) && (idle > 0)) {
    auxSp = bqueue.removeFromFront();
    op.send(firstidle,auxSp,bestSol,sol);
    idle--;
    IDLE2WORKING(busy,lastIdle); mark slave as working
  }
  MLB_Probe(MPI_ANY_SOURCE,MPI_ANY_TAG,flag,status)
  while (flag) {
    if (MPI_SOLVE_TAG) {
      ip.recv(source,bestSol,sol);
      WORKING2IDLE(busy,source); mark the salve as idle
    }
    if (MPI_BnB_TAG) {
      // Receive the best solution of the slave
      ip.recv(source,bstemp,high);
      if ( high > bestSol) {
        bestSol = bstemp;
        // request the queue of the subproblems to the slave
        op.send(source, MPI_QUEUEREQUEST_TAG);
        // receive the queue of subproblems generated by the slave
        ip.recv(source, auxqueue);
      }
      else {
        // slave's subproblems queue must be deleted
        op.send(source,MPI_QUEUEDELETE_TAG);
      }
      WORKING2IDLE(busy,source);
    }  // end-if
  }   // final of while flag
  // Link Master's queue with the received queue from the slave
  (bqueue.getTail())->setNext(auxbqueue.getHead());
  bqueue.setTail(auxbqueue.getTail());
} //while
// Send the ending message
for i = 0; groupSize { op.send(i+1, MPI_END_TAG); }
```

Fig. 3.15 Centralized MPI Master Code

```
while (1) {
  MLB_Probe(MASTER,MPI_ANY_TAG,flag,status)
  if (flag) {
    switch(status.MPI_TAG) {
      // Ending message
      if (END_TAG){
        ip.recv(MASTER, MPI_END_TAG);
        return;
      }
      // Receive the problem to branch
      if (PBM_TAG){
        ip.recv(MASTER,MPI_PBM_TAG,auxSp,bestSol,auxSol);
        high = auxSp.upper_bound(pbm,auxSol);
        if ( high > bestSol ) {
          low = auxSp.lower_bound(pbm,auxSol);
          if ( low > bestSol ) {
            bestSol = low;
            sol = auxSol;
          }
        }
        if ( high = low ) {
          // Send the best solution to Master with label SOLVE_TAG
          op.send(MASTER,MPI_SOLVE_TAG,bestSol);
        }
        else {
          op.send(MASTER,MPI_BnB_TAG,bestSol,high);
        }
        // Receive from Master the order to branch and to send
        // the generated subproblems or not to branch
        ip.recv(MASTER, MPI_QUEUEREQUEST_TAG,count);
        if ( count = 1) {   // it has to branch the subproblem
          auxSp.branch(pbm,bqueue);
          op.send(MASTER,bqueue);
        }
        bqueue.clean();
      }
    }   // switch
  }   // flag
} // while
```

Fig. 3.16 Centralized MPI Slave Code

3.7.2 Distributed Parallel Scheme

Distributed Data Structure

The data structure for the distrituted parallel scheme is represented in figure 3.17. Each one of the data structures used in this scheme by the slaves are doubly linked queues. In this case, the master processor does not possess a global queue of unsolved problems. The master sends the first subproblem to the first idle slave. Each slave has a local queue to store the new subproblems generated after previous subproblem branching. The unsolved subproblems are directly distributed between idle slaves by overworked slaves. The subproblems sent by the slaves are stored in the local queues belonging to the other slaves.

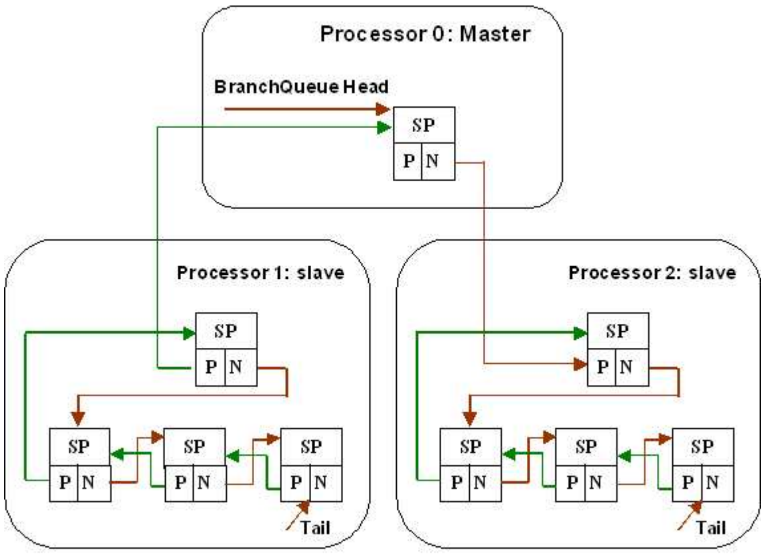


Fig. 3.17 Distributed Data Structure

Distributed Algorithm

This section describes the design and implementation of the MaLLBa::BnB distributed message passing resolution pattern.

```
$\mathcal{P} := \{1, ..., p\}$; $\mathcal{I} := \mathcal{P}$;
$bs := -\infty$; $\mathcal{T} := \emptyset$;
$i := s(\mathcal{I})$; $\mathcal{I} := \mathcal{I}$ - $\{i\}$;
$send(\Pi, i)$;
while ($\mathcal{I} \neq \mathcal{P}$) {
      $receive(q, flag)/q \in \mathcal{P}$;
      while ($flag$) {
         if ($flag$ =SOLVED) {
           $receive(q, bs, \mathcal{T})$;
           }
           if ($flag$ =BnB) {
             $receive(q, h, r)$;
             if ($h > bs$ ) {
                  $\mathcal{I} := \mathcal{I} - \{p_1, \ldots, p_r\}/p_k = s(\mathcal{I})$;
                  $send(bs, \{p_1, \ldots, p_r\}, q)$;
             }
             else
             $send(\text{DONE}, q)$;
           }
           if ($flag$ =IDLE) {
             $receive(q, \text{ IDLE})$;
             $\mathcal{I} := \mathcal{I} \cup \{q\}$
           }
           $receive(q, flag)/q \in \mathcal{P}$;
      }
 }
 $send(\text{END}, i)$; $\forall i \in \mathcal{P}$
```

Fig. 3.18 Distributed Master Processor Algorithm

```
while ($true$) {
    $receive(q, flag)/q \in \mathcal{P}$;
    while ($flag$) {
      if ($flag =$ END) {
        $receive(Master,$END$)$;
        return;
      }
      if ($flag =$ PBM) {
        $receive(q, \Pi_i, bs_i, \mathcal{T}_i)$;
        $\mathcal{L} := \{\Pi_i\}$; $\mathcal{Q} := \{\Pi_i\}$; $bs := bs_i$; $\mathcal{T} := \mathcal{T}_i$;
        while ($\mathcal{L} \neq \emptyset$) {
            $\Pi_j := s(\mathcal{L})$;
            if ($upper\_bound(\Pi_j) > bs$) {
              if ($lower\_bound(\Pi_j) > bs$) {
                $bs := lower\_bound(\Pi_j)$;
                $\mathcal{T} := \{\sigma\}$; / $f(x, \sigma) = lower\_bound(\Pi_j)$;
                $send(bs, \mathcal{T},$SOLVED$, Master)$;
              }
              (brach) decompose $\Pi_j$ into $\Pi_{j_1}, \Pi_{j_2}, ..., \Pi_{j_k}$
              $\mathcal{L} := \mathcal{L} \cup \{\Pi_{j_1}, \Pi_{j_2}, ..., \Pi_{j_k}\}$ - $\{\Pi_j\}$;
              $\mathcal{Q} := \mathcal{Q} \cup \{\Pi_{j_1}, \Pi_{j_2}, ..., \Pi_{j_k}\}$;
              $send(bs, k,$BNB$, Master)$;
              $receive(Master, bs, \mathcal{P})$;
              while ($\mathcal{P} \neq \emptyset$) {
                $n := s(\mathcal{P})$; $\mathcal{P} := \mathcal{P} - \{n\}$; $\Pi_n := s(\mathcal{L})$; $\mathcal{L} := \mathcal{L} - \Pi_n$;
                $send(\Pi_n, n)$;
              }
            }
            $\mathcal{L} := \mathcal{L}$ - $\Pi_j$;
        }
        $send($IDLE$, Master)$;
      }
      $receive(q, flag)/q \in \mathcal{P}$;
    }
}
```

Fig. 3.19 Distributed Slave Processor Algorithm

The algorithms for Master and Slave are presented in figures 3.18 and 3.19 respectively [22]. Figure 3.18 shows the Master tasks. The generation of new subproblems and the evaluation of the results of each of them are completely separate from the individual processing of each subtask. The Master is responsible for the coordination between subtasks. The Master has a data structure $\mathcal{I}$ which registers the occupational state of each slave (line 1); at the beginning all the slaves are idle. The initial subproblem, the best solution and the best value of the objective function are sent to an idle slave (lines 2-4). While there are idle slaves the Master receives information from them and decides the next action to apply depending on whether the problem is solved (line 8), or there is a slaves request (line 11) or whether the slave is idle (line 20). If the problem is solved, the solution is received and stored (line 9). When the master receives a slave request from a slave, it is followed with the upper bound value (line 12). If the upper bound value is better than the actual value of the best solution, the answer to the slave includes the number of slaves that can help to solve the problem (line 15). Otherwise, the answer indicates that it is not necessary to work in this subtree (line 18). When the number of idle slaves is equal to the initial value, the search process finishes, then the Master notifies the slaves to finish work (line 27).

A slave (see figure 3.19) works bounding the received problem (line 9). New subproblems are generated calling on the `branch` method (line 19). The slave asks the Master for help (lines 21-22). If no free slaves are available, the slave continues working locally. Otherwise, it removes subproblems from its local queue and sends them directly to other slaves (lines 24-25).

The scheme presented is a distributed scheme in the sense that each slave must distribute the subproblems between other slaves. On the other hand, the Master checks to see whether there are free slaves or not, for this reason the proposed scheme is not an entirely distributed.

The implementation of the MaLLBa::BnB skeleton shown in figures 3.20, 3.21 and 3.22 uses MPI_Send and MPI_Recv, to send and receive messages respectively [21]. The main loop in the Master and Slave codes are implemented using MPI_IProbe. When a message is received its status is used to classify what kind of work should be done: finish, receive a problem for bounding and branching, receive a request from slaves, etc.

```
busy[0] = 1; for i = 1, nProcs { busy[i] = 0;}
idle = nProcs - 1;
//Send initial subproblem to first idle slave
auxSp = sp.initSubProblem();
outputPackect.send(firstIdle,
                   auxSp,   // initial subproblem
                   bestSol, // best Solution
                   sol);    // current solution
idle--;
IDLE2WORKING(busy,firstIdle);   // mark this slave as working
// while there are working slaves
while (idle < (groupSize-1)) {
  recv(source, flag);
  while(flag) {
    if (SOLVE_TAG) {         // receive the final solution
      inputPacket.recv(source,
                       bestSol,
                       sol);   // current solution
    }
    if (BnB_TAG) {        // receive a slave request
      inputPacket.recv(source,
                       high,     // upper bound of the problem
                       nSlaves); // number of required slaves
      if ( high > bestSol){      // the problem must be branched
        total= ((nSlaves <= idle)?nSlaves:idle);
        for i = 1, total { idle--; IDLE2WORKING(busy,i); }
        outputPacket.send(source,
                          total,   // number of assigned slaves
                          bestSol,     // best Solution
                          1,...,total); // slaves identifiers
      }
      else { // the problem must be bounded
        outputPackted.send(source, DONE);
    } }
    if (IDLE_TAG) {     // receive the signal of an idle slave
      inputPacket.recv(source, IDLE);
      idle++;
      WORKING2IDLE(busy,source); // mark this slave like idle
    }
    recv(source, flag);
} } // while (idle < (groupSize-1))
// Send the ending message
for i = 1, groupSize { outputPacket.send(i, END); }
```

Fig. 3.20 Distributed MPI Master Code

```
while (1) {
  recv(source, flag);
  while (flag) {
    if (END_TAG){         // receive the finishing message
      inputPacket.recv(MASTER, END); return;
    }
    if (PBM_TAG){    // receive the problem to be branched
      inputPacket.recv(source,  // from a slave or Master:
                       auxSp,   // the initial subproblem
                       bestSol, // the best solution value
                       sol);    // the current solution
      auxSol = sol;
      bqueue.insert(auxSp);   // insert it in the local queue
      while(!bqueue.empty()) {
        // pop a problem from the local queue
        auxSp = bqueue.remove();
        high = auxSp.upper_bound(pbm,auxSol);   // upper bound
        if ( high > bestSol ) {
          low = auxSp.lower_bound(pbm,auxSol);  // lower bound
          if ( low > bestSol ) {
            bestSol = low;
            sol = auxSol;
            // send to Master: the problem is solved,
            // best solution value and solution vector.
            outputPacket.send(MASTER,SOLVE_TAG,bestSol,sol);
          }
          if ( high != low ) {
            // calculate the number of required slaves
            rSlaves = bqueue.getNumberOfNodes();
            // send to the Master: request of help,
            // upper bound and number of required slaves
            op.send(MASTER,BnB_TAG,high,rSlaves);
            inputPacket.recv(MASTER,    // receive from Master:
                             nfSlaves,  // number of assigned slaves
                             bestSol,   // best solution value
                             rank{1,...,nfSlaves});
```

Fig. 3.21 Distributed MPI Slave Code (Part I)

```
              if ( nfSlaves >= 0) {
                // branch and save in the local queue
                auxSp.branch(pbm,bqueue);
                // send subproblems to the assigned slaves
                for i=0, nfSlaves{
                  auxSp = bqueue.remove();
                  // send to the slave: a subproblem to solve,
                  // the best solution value and the solution vector
                  outputPacket.send(rank,PBM_TAG,auxSp,bestSol,sol);
              } } // if nfSlaves == DONE the problem is bounded
        } } }
        // send to Master the signal to say it is idle
        outputPacket.send(MASTER, IDLE_TAG);
       }
       recv(source, flag);
   }  } // while(1)
```

Fig. 3.22 Distributed MPI Slave Code (Part II)

3.7.3 Shared Memory Parallel Scheme

Shared Data Structure

The data structure used by the shared memory scheme is represented in figure 3.23. It is exactly the same as the sequential scheme queue, but using shared memory, e.d., a doubly linked queue. In this case, all processors access the unsolved subproblems inserted in this queue and the new subproblems generated by each slave after branching, which are also stored in the same queue.

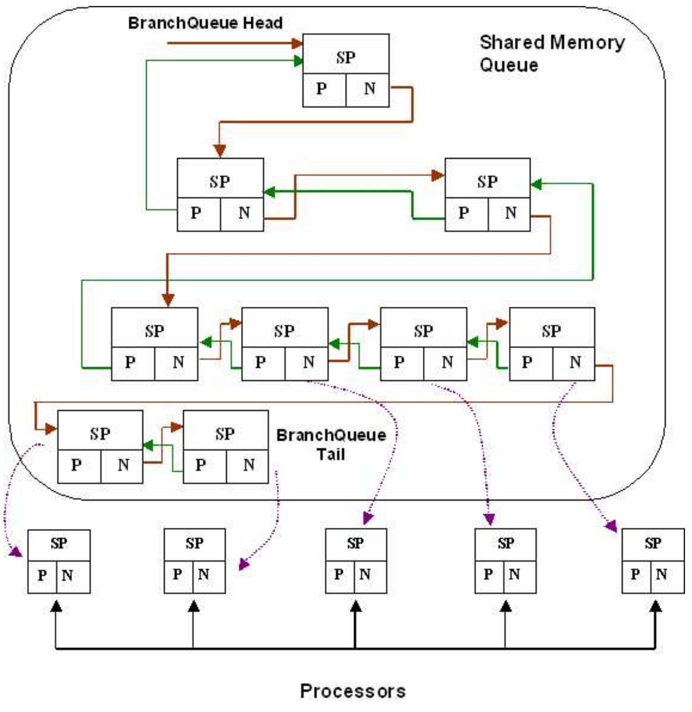


Fig. 3.23 Shared Data Structure

Shared Memory Algorithm

The algorithm proposed in this section works with a global shared queue of $\mathcal{L}$ tasks implemented using a linked data structure. For the implementation of

```
1   $\mathcal{L} := \{\Pi_0\}$; $\mathcal{Q} := \{\Pi_0\}$;
2   $bs := -\infty$; $\mathcal{T} := \emptyset$;
3   $\mathcal{P} := \{1, ..., p\}$;
4   while ($\mathcal{L} \neq \emptyset$) {
5       while ( $\mathcal{P} \neq \emptyset$) {
6           $n := s(\mathcal{P})$; $\mathcal{P} := \mathcal{P} - \{n\}$; $\Pi_n := s(\mathcal{L})$; $\mathcal{L} := \mathcal{L} - \Pi_n$;
7       }
8       parallel for ($n \in \mathcal{P}$) {
9         if ($upper\_bound(\Pi_n) > bs$) {
10          if ($lower\_bound(\Pi_n) > bs$) {
11              $bs := lower\_bound(\Pi_n)$;
12              $\mathcal{T} := \{\sigma\} / f(x, \sigma) = lower\_bound(\Pi_n)$;
13          }
14          if ($upper\_bound(\Pi_n) \neq lower\_bound(\Pi_n)$ ) {
15            (brach) decompose $\Pi_n$ into $\Pi_{n1}, \Pi_{n2}, ..., \Pi_{nk}$
16            $\mathcal{L} := \mathcal{L} \cup \{\Pi_{n1}, \Pi_{n2}, ..., \Pi_{nk}\}$ - $\{\Pi_n\}$ ;
17            $\mathcal{Q} := \mathcal{Q} \cup \{\Pi_{n1}, \Pi_{n2}, ..., \Pi_{nk}\}$ ;
18          }
19        }
20      }
21  }
```

Fig. 3.24 MaLLBa::BnB Shared Memory Skeleton

the MaLLBa::BnB shared memory resolution pattern we have followed a very simple scheme [25] (see figure 3.24). First, the number of threads $\mathcal{P}$ is calculated and established (lines 5-6 and 10). Then $\mathcal{P}$ subproblems are removed from the queue and assigned to each thread (lines 5-7). In the parallel region from line 8 to 18 each assigned thread works on its own subproblem. The lower and upper bounds are calculated. The best solution value and the solution vector must be modified carefully, because only one thread can change the variable at any time. Next the sentences in lines 11 and 12 must be in a critical region. The same special care must take into account when a thread tries to insert a new subproblem in the global shared queue (line 16).

The generic shared memory code [23] outlined in figure 3.25 is instantiated for the OpenMP API as follows: The `omp_set_num_threads` is used to establish the number of threads. OpenMP supports two kinds of work-sharing constructs to speedup parallel programs: the *for* pragma and the *sections* pragma. Neither of these constructs can be utilized to parallelize access to list or tree structured

```
// shared variables  {bqueue, bstemp, soltemp, data}
// private variables {auxSol, high, low}
// the initial subproblem is already inserted
// in the global shared queue
while(!bqueue.empty()) {
  // calculate the number of necessary threads
  nn = bqueue.getNumberOfNodes();
  nt = (nn > maxthread)?maxthread:nn;
   // compute the subproblem for each thread
  data = new SubProblem[nt];
  for (int j = 0; j < nt; j++)
    data[j] = bqueue.remove();
  // establish the right number of threads
  set.num.threads(nt);
  parallel forall (i = 0; i < nt; i++) {
    high = data[i].upper_bound(pbm,auxSol);
    if ( high > bstemp ) {
      if ( low > bstemp ) {   // critical region
        // only one thread can change the value at any time
        bstemp = low;
        soltemp = auxSol;
      }
      if ( high != low ) {    // critical region
        // just one thread can insert subproblems
        // in the queue at any time
        data[i].branch(pbm,bqueue);
} } } }
bestSol = bstemp;
sol = soltemp;
```

Fig. 3.25 MaLLBa::BnB Skeleton OpenMP Code

data [81]. The `parallel for` pragma substitutes the generic parallel forall. Finally, the OpenMP pragma `critical` has been used for the implementation of the two critical regions.

4 Computational Results

This chapter is organized as follows: we present the programming language and the parallel tool chosen to develop MaLLBa in sections 4.1 and 4.2 respectively. All types of machines used to obtain the computational results are mentioned in section 4.3. The computational results of MaLLBa::BnB skeletons for distributed and shared memory are presented in section 4.4. Finally, in section 4.5 we present the performance analysis of MaLLBa::BnB skeletons using the tools CALL-LLAC and PARAVER, and some conclusions of this study.

4.1 CHOOSING THE PROGRAMMING LANGUAGE

With the objective of taking a decision about the programming philosophy to develop the MaLLBa tool, we accomplished a study of the time required by the sequential Branch-and-Bound algorithm implemented, using an imperative language and an Oriented Object language. The experiments were executed on a Pentium III/600 Mhz with 256 Mb of memory.

The following graphics show the times obtained executing the MaLLBa::BnB implementation versus the Ansi C implementation. Figures 4.1 and 4.2 present the comparison between both languages for different sizes of the knapsack prob-

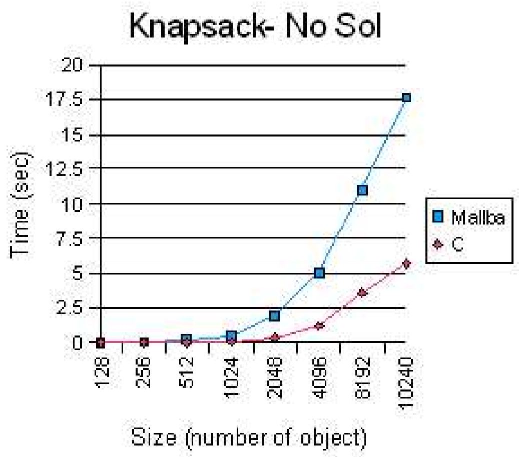


Fig. 4.1 Comparison between Sequential MaLLBa and C Implementations

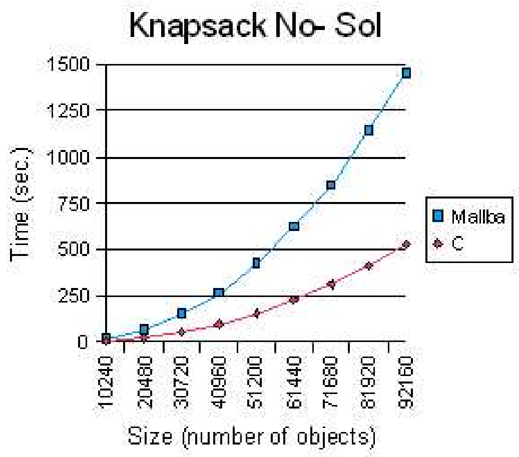


Fig. 4.2 Comparison between Sequential MaLLBa and C Implementations

lem; these problems were generated for sizes between [128..92160]. Figures 4.3 and 4.4 represent the MaLLBa time divided by Ansi C time. The algorithm computes only the best solution in all cases. As can be appreciated, the compu-

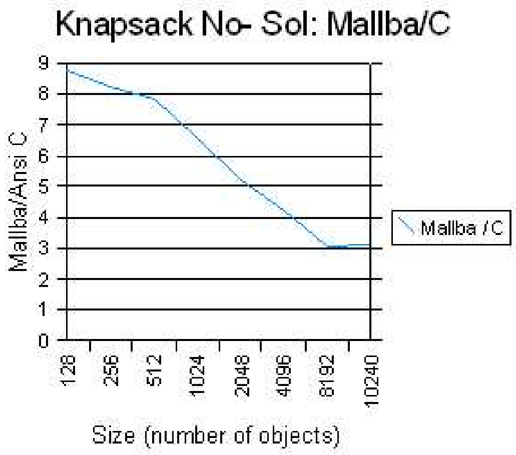


Fig. 4.3 MaLLBa/Ansi C Representation

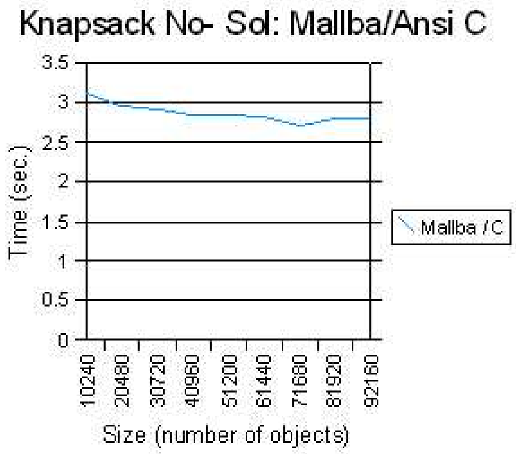


Fig. 4.4 MaLLBa/Ansi C Representation

tational results obtained with MaLLBa::BnB were good, but lightly less efficient than those obtained using C language.

After some coordination meetings between the project members of the three participating universities, we opted for an oriented object (OO) methodology. To the advantages this philosophy provides: modularity, re-usable, modifiable, and so on, is added the interpretation facility of the skeleton, mainly because there is quite an intuitive relationship between the entities participating in the resolution pattern and the classes implemented by the skeleton.

4.2 CHOOSING THE PARALLEL PROGRAMMING TOOLS

It is a well established fact that Parallel Computing offers efficient solutions to Combinatorial Optimization Problems. However, the parallel supercomputers are not obtainable by many organizations due to the high cost of the hardware, maintenance and programming. A smaller cost alternative widely used and consolidated is the use of personal computer networks. The possibility of connecting personal computer networks dispersed geographically through Internet allows for the improvement in performance of certain applications.

In taking a decision about the parallel tool to use during the development of the MaLLBa project, some experiments to measure the communications performance were accomplished. The experiments consisted of the implementation of the Ping-Pong communications pattern between two machines in three different areas using PVM and MPI: two local area machines belonging to La Laguna cluster, two machines from the Barcelona cluster and two machines a great distance away: one from the La Laguna cluster and the other from the Barcelona cluster. The time measurements were taken during the night and assuming exclusivity (some tests accomplished during the day provided worse results). The graphics show the results obtained for MPI and PVM. The tests were accomplished for sizes between 4B and 4MB. The times represent the average time in seconds for the repetition of each test, 10 times for each size, and for BCN, LL and LL-BCN.

Representing graphically the PVM as compared to the MPI, it can be seen that the times obtained using MPI are inferior to those obtained by PVM for the Barcelona cluster (see graphic 4.5). With the test results from La Laguna the same occured (see graphic 4.6). Finally, the test results between the clusters of La Laguna and Barcelona are presented in graphic 4.7. In spite of the PVM obtaining better times for small sizes, for large sizes of messages MPI provides smaller times. Therefore, we decided to implement MaLLBa using MPI.

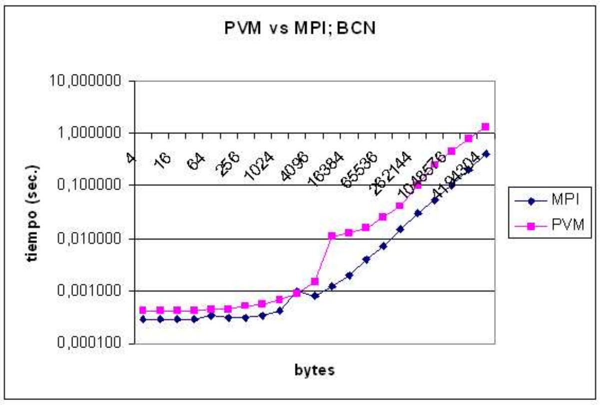


Fig. 4.5 Comparison of Performance Analysis in Communications between MPI and PVM Using the Barcelona Cluster

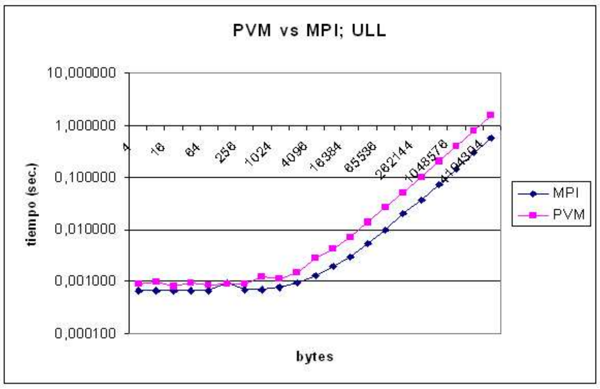


Fig. 4.6 Comparison of Performance Analysis in Communications between MPI and PVM Using the La Laguna Cluster

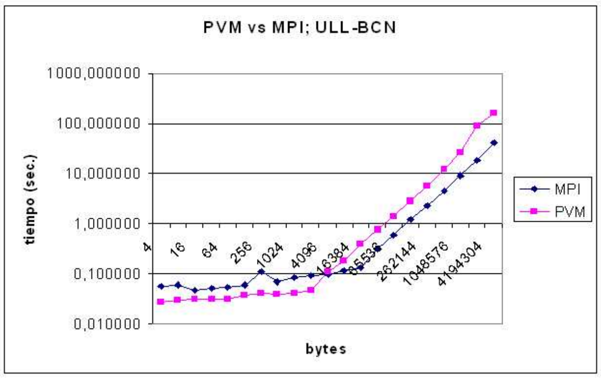


Fig. 4.7 Comparison of Performance Analysis in Communications between MPI and PVM Using the Clusters of La Laguna and Barcelona

4.3 DESCRIPTION OF THE MACHINES

The experiments were carried out on four different machines:

- Sunfire 6800 SMP, with the following configuration: 24 750 MHz UltraSPARC-III processors, 48 Gbyte of shared memory each and 120 Gbyte of disk space per system. This machine belongs to the EPCC (Edinburgh Parallel Computing Center) at the University of Edinburgh [33].

- Origin 3000, whose configuration is 160 600 MHz MIPS R14000 processors, 1 Gbyte of memory each and 900 Gbyte of disk. The CIEMAT (Centro de Investigaciones Energéticas, Medioambientales y Tecnológicas) has allowed us access to these computers [13].

- A 344-processor T3E-900/LC configured with: 344 450 MHz Alpha processors, each having a peak performance of 309 GFlops with a mix of 64 and 128 Mbytes of memory. It also belongs to the EPCC.

  We ran some experiments of the MaLLBa::BnB skeleton on this machine during our visit to the EPCC. Since the preliminary results of the skeleton were obtained for small sizes of the knapsack problem (between 1000 and 6000 objects), they have not been included in this chapter. At the same time the algorithm was improved, the machine was turned off and has been out of service since September 2002.

- An heterogeneous Cluster of PCs, which was configured with 2 750 MHz AMD Duron Processors, 4 800 MHz AMD Duron Processors, 7 500 MHz AMD-K6 3D processors, 256 Mbyte of memory each and 32 Gbyte of hard disk each. This was the cluster of PCs installed by the Parallel Computing Group of La Laguna for use in the MALLBA project [65].

The software used in the Sunfire 6800 SMP was the mpCC compiler of C++ and the UNICOS/mk tool. In the Origin 3000 the MPISpro CC compiler of C++ (version 7.3.1.2m) and IRIX MPI was used. Finally, in the PCs cluster the operating system was Debian Linux version 2.2.19 (herbert@gondolin), the C++ compiler was GNU gcc version 2.7.2.3 and the *mpich* version was 1.2.0.

4.4 COMPUTATIONAL RESULTS

In the previous chapters the algorithm used for the tests is the classic Knapsack 0/1 problem described by Martelo and Toth [69] has been explained. In this chapter we analyze the experimental behavior for this problem on sets of randomly generated test problems. Since the difficulty of such problems is affected by the correlation between profits and weights, we only considered those with a strong correlation.

Tables 4.1, 4.2 and 4.3 were obtained using MPI. The first and the second tables show the speedup results of five executions of a 50,000 size problem, while the third was obtained through ten executions. Only the optimal value of the objective function was calculated. The first column contains the number of processors. The second column shows the average time in seconds. The column labeled "Speedup-Av" presents the speedup for the average times. The third and fifth columns give the minimum times (seconds) and the associated speedup whereas the fourth and sixth columns show the maximum times (seconds). Table 4.4 shows the results for the problem using the OpenMP skeleton. The solution vector was not calculated in these experiments. If we compare these results with the ones obtained using MPI a similar behavior can be appreciated between both. However, when the number of processors increase the speedup of the OpenMP version decreases while the MPI stays stable.

Tables 4.5 and 4.6 show the obtained results when the solution vector is calculated. The number of items is 50,000. Only the results for the Origin 3000 and the Sunfire 6800 are presented. As can be appreciated, when the solution is obtained the speedups are better because the problem to solve is

Table 4.1 Sunfire 6800. MPI Implementation. N = 50,000. No Solution Vector. The Sequential Average Time is 196.00 Seconds

Procs	Average	Min	Max	Speedup-Av	Speedup-Max	Speedup-Min
2	204.59	203.36	206.03	0.96	0.96	0.95
3	139.41	135.74	141.87	1.41	1.44	1.38
4	115.59	110.05	122.65	1.70	1.78	1.60
8	66.81	65.99	67.81	2.93	2.97	2.89
16	48.32	47.85	48.87	4.06	4.10	4.01
24	48.14	47.50	49.26	4.07	4.13	3.98

Table 4.2 Origin 3000. MPI Implementation. N = 50,000. No Solution Vector. The Sequential Average Time is 123.71 Seconds

Procs	Average	Min	Max	Speedup-Av	Speedup-Max	Speedup-Min
2	124.37	124.28	124.61	0.99	1.00	0.99
3	84.38	84.05	84.57	1.47	1.47	1.46
4	67.81	64.68	71.03	1.82	1.91	1.74
8	38.83	36.55	41.90	3.19	3.39	2.95
16	28.65	28.17	29.65	4.32	4.39	4.17
24	29.63	28.96	29.99	4.18	4.27	4.12
32	29.70	28.36	31.13	4.17	4.36	3.97

Table 4.3 PC Cluster. MPI Implementation. N = 50,000. No Solution Vector. The Sequential Average Time is 758.78 Seconds

Procs	Average	Min	Max	Speedup-Av	Speedup-Max	Speedup-Min
2	1664.78	1649.66	1678.38	0.46	0.46	0.45
3	1247.19	1138.87	1515.34	0.61	0.67	0.50
4	912.63	860.86	975.20	0.83	0.88	0.78
8	766.59	736.54	813.34	0.99	1.03	0.93
10	745.56	692.28	844.66	1.02	1.10	0.90

Table 4.4 Origin 3000. OpenMP Implementation. N = 50,000. No Solution Vector. The Sequential Average Time in Seconds is 869,76

Procs	Average	Min	Max	Speedup-Av	Speedup-Max	Speedup-Min
2	747.85	743.98	751.02	1.16	1.17	1.16
3	530.74	517.88	540.76	1.64	1.68	1.61
4	417.62	411.02	422.59	2.08	2.12	2.06
8	235.10	231.26	237.00	3.70	3.76	3.67
16	177.72	162.00	207.15	4.89	5.37	4.20
24	157.30	151.18	175.67	5.53	5.75	4.95
32	176.43	174.23	179.01	4.93	4.99	4.86

Table 4.5 Sunfire 6800. MPI Implementation. N = 50,000. The Sequential Average Time is 2,649.95 Seconds

Procs	Average	Min	Max	Speedup-Av	Speedup-Max	Speedup-Min
2	2,663.29	2,658.90	2,669.59	0.99	1.00	0.99
3	1,909.75	1,807.75	2,011.74	1.39	1.47	1.32
4	1,473.65	1,427.73	1,525.89	1.80	1.86	1.74
8	844.00	781.28	896.20	3.14	3.39	2.96
16	459.18	445.37	471.71	5.77	5.95	5.62
24	419.07	412.17	430.98	6.32	6.43	6.15

harder, that is, it is a coarse-grain problem. For problems of size 100,000 memory allocations problems appear.

Figure 4.8 shows the obtained results for the knapsack problem when the number of object is 50,000 and the solution vector is also calculated. The results of the executions of the Origin 3000 and the Sunfire 6800 are represented. As can be appreciated, when the solution is obtained the speedups are better because the problem to solve is harder, that is, it is a coarse-grain problem.

Tables 4.7, 4.8 and 4.9 were obtained using MPI. The first and the second tables show the speedup results for five executions of the 0/1 Knapsack Problem randomly generated for size 100,000, while the third was obtained through ten executions. Only the optimal value of the objective function was calculated. The meaning of the labels are the same as those previously used. Figure 4.9 shows these results graphically.

Table 4.6 Origin 3000. MPI Implementation. N = 50,000. The Sequential Average Time is 563.93 Seconds

Procs	Average	Min	Max	Speedup-Av	Speedup-Max	Speedup-Min
2	561.94	555.50	577.85	1.00	1.02	0.98
3	429.59	368.07	528.08	1.31	1.53	1.07
4	415.50	390.16	484.82	1.36	1.45	1.16
8	194.01	186.73	205.83	2.91	3.02	2.74
16	121.87	117.23	130.77	4.63	4.81	4.31
24	108.80	107.06	110.88	5.18	5.27	5.09
32	103.84	102.11	106.31	5.43	5.52	5.30

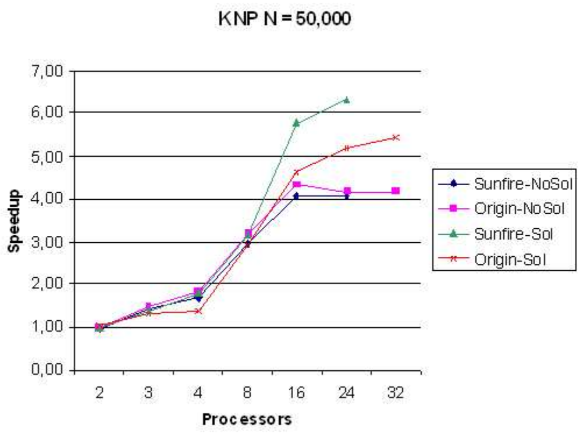


Fig. 4.8 Speedups. With and Without Solution Vector

Due to the fine grain of the 0/1 knapsack problem, there is no lineal increase in the speed up when the number of processors increase. For large numbers of processors the speed up is poor. For these cases there is still the advantage of being able to manage large size problems that cannot be solved sequentially. The limited performance of two processor systems is due to the Master/Slave scheme followed in the implementation. In this case, one of the processors is the Master and the others are the workers and therefore, communications and work are not overlapped.

Table 4.7 Sunfire 6800. MPI Implementation. The Sequential Average Time is 1529.91 Seconds

Procs	Average	Min	Max	Speedup-Av	Speedup-Max	Speedup-Min
2	1560.15	1554.38	1569.08	0.98	0.98	0.98
3	1185.65	1177.28	1204.49	1.29	1.30	1.27
4	767.02	625.81	896.83	1.99	2.44	1.71
8	412.41	378.02	462.95	3.71	4.05	3.30
16	303.66	284.78	315.77	5.04	5.37	4.85
24	250.10	239.37	258.37	6.12	6.39	5.92

Table 4.8 Origin 3000. MPI Implementation. The Sequential Average Time in Seconds is 867.50

Procs	Average	Min	Max	Speedup-Av	Speedup-Max	Speedup-Min
2	933.07	932.67	933.80	0.93	0.93	0.93
3	743.65	740.73	745.29	1.17	1.17	1.16
4	454.77	399.86	492.43	1.91	2.17	1.76
8	251.82	236.94	266.96	3.44	3.66	3.25
16	186.18	174.24	192.80	4.66	4.98	4.50
24	152.49	144.29	167.92	5.69	6.01	5.17
32	151.09	144.69	166.46	5.74	6.00	5.21

Table 4.9 PCs Cluster. MPI Implementation. The Sequential Average Time is 5,357.02 Seconds

Procs	Average	Min	Max	Speedup-Av	Speedup-Max	Speedup-Min
2	11,390.69	11,343.45	11,434.42	0.47	0.47	0.47
3	9,344.12	6,175.15	10,044.11	0.57	0.87	0.53
4	5,162.14	4,538.29	6,195.14	1.04	1.18	0.86
8	3,722.09	3,478.06	4,642.56	1.44	1.54	1.15
10	3,518.24	3,299.57	3,699.64	1.52	1.62	1.45

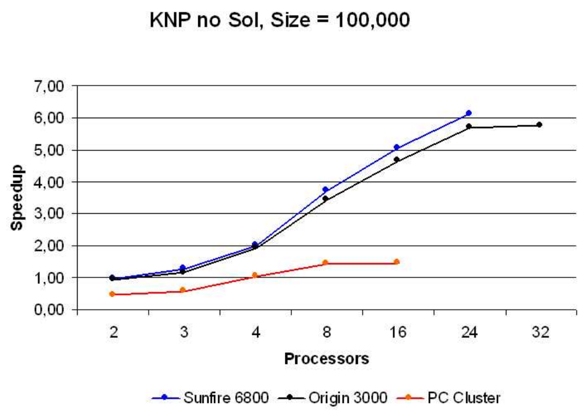


Fig. 4.9 Speedups. No Solution Vector

Table 4.10 Origin 3000. OpenMP Implementation. The Sequential Average Time in Seconds is 869,76

Procs	Average	Min	Max	Speedup-Av	Speedup-Max	Speedup-Min
2	747.85	743.98	751.02	1.16	1.17	1.16
3	530.74	517.88	540.76	1.64	1.68	1.61
4	417.62	411.02	422.59	2.08	2.12	2.06
8	235.10	231.26	237.00	3.70	3.76	3.67
16	177.72	162.00	207.15	4.89	5.37	4.20
24	157.30	151.18	175.67	5.53	5.75	4.95
32	176.43	174.23	179.01	4.93	4.99	4.86

The table 4.10 shows the results of the problem using the OpenMP skeleton. The solution vector is not calculated in these experiments. Figure 4.10 compares the results obtained using the MPI and the OpenMP implementations without the solution vector. A similar behavior can be appreciated between both libraries. However, when the number of processors increase the speedup of the OpenMP version decreases while the MPI stays stable.

Figure 4.11 shows a comparison between the results obtained for three different sizes: 10.000, 50.000 and 100.000 objects of the knapsack Problem

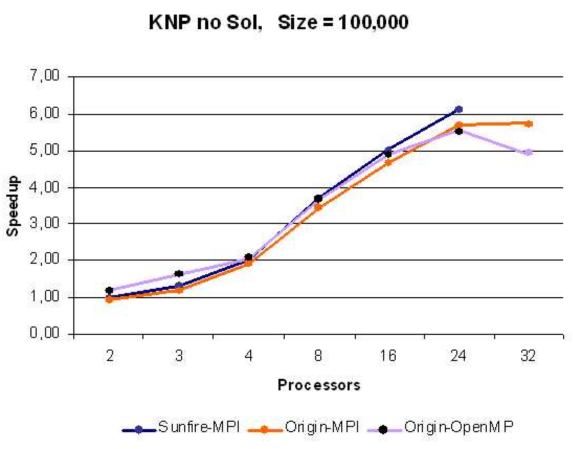


Fig. 4.10 Comparison between MPI and OpenMP

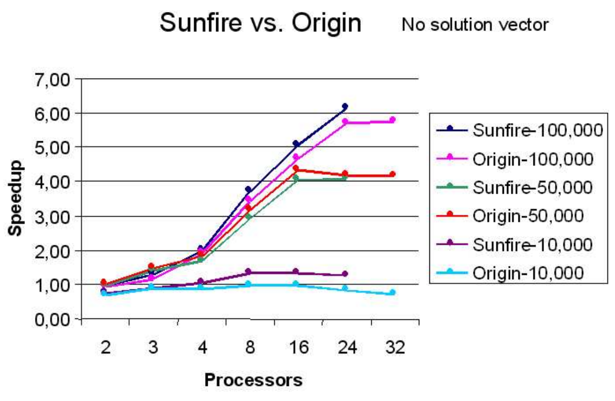


Fig. 4.11 Comparison between Sunfire and Origin

run on Sunfire and Origin Machines. It can be appreciated that the Sunfire obtains better results than the Origin for large size problems.

4.5 PERFORMANCE ANALYSIS

Once the computational results were obtained, we decided to accomplish the performance analysis of our presented skeletons. We specifically approached the study of the load balancing and the communication pattern between the processors in this section, using the tools CALL-LLAC [76] and PARAVER [11] available in the ULL-Cluster. It is important to emphasize that these tools work on code implemented in C, therefore we have developed a version in C of MaLLBa::BnB tool.

4.5.1 Study of Load Balancing

To accomplish the study of the load balancing of an algorithm, the code can be annotated by hand at the special points where the number of visited nodes is increased. Since some members of our parallel group have developed a tool called CALL to tackle studies like this, we have collaborated with them to apply the CALL tool to our skeleton [24].

The complexity analysis of an algorithm produces as a result a “complexity function” that provides an approximation of the number of operations to accomplish. The CALL tool allows the logging of the C code that implements the algorithm with that complexity expression.

The algorithm presented in figure 4.12 shows the recursive code used to solve the Knapsack Problem. The number of objects for insertion into the knapsack is stored in the variable N, the variables w and p store the weights and the benefits of each of them, while the variable M represents the capacity. Since it is a maximization problem, the initial value $-\infty$ is assigned to the variable that stores the best solution found until that moment: `bestSol` (lines 1 to 4). Between lines 8 and 18 the bound function (`lowerUpper`) is defined. We use the same function to calculate the lower and upper bounds. The lower bound is defined as the maximum benefit that can be obtained from a given subproblem, while the upper bound includes the proportional part of the benefit of the last object that could not be inserted into the knapsack. The function, `knap` (lines 20-35), implements a recursive Branch-and-Bound algorithm. The call to the function that studies the insertion of the object k is accomplished in line 29,

```
/* ficheros de cabecera */
number N, M;
number w[MAX], p[MAX];
number bestSol = -INFINITY;

#pragma cll code double numvis;

void lowerUpper(number k, number C,number P,number *L, number *U) {
  number i, weig, prof;
  if (C <0) {*L = -INFINITY; *U = -INFINITY;  }
  else {
    for(i=k, weig = 0, prof = P; weig <= C; i++)
      {weig += w[i]; prof += p[i];}
      i--;
      weig -= w[i]; prof -= p[i];
      *L = prof;  *U = prof+(p[i]*(C-weig))/w[i];
  }
}

number knap(number k, number C, number P) {
  number L, U, next;
    if (k < N) {
      lowerUpper(k,C,P,&L,&U);
      if (bestSol < L) {
        bestSol = L;
      }
      if (bestSol < U) { /* L <= bestSol <= U */
        next = k+1;
        knap(next, C - w[k], P + p[k]);
        knap(next, C, P);
#pragma cll code numvis += 2 ;
      }
    }
    return bestSol;
}

int main(int argc, char ** argv) {
number sol;
readKnap(data);
#pragma cll code double numvis = 0.0;
#pragma cll kps kps[0]*unknown(numvis) posteriori numvis
/* obj. sig., capacidad rest., beneficio */
  sol = knap( 0, M, 0);
#pragma cll end kps
  printf("\nsol = ", sol);
#pragma cll report all
```

Fig. 4.12 Using CALL in the Sequential Implementation

while line 30 considers not inserting it. The condition to update the value of the best solution (`bestSol`) found until that moment is implemented between lines 22 to 26.

We are interested in knowing the behavior of the algorithm. To be precise, we would like to know the number of search space nodes visited until the best solution is found. To accomplish this study, we note down the code with CALL directives. Line 6 is used to specify the CALL definition of a double type variable to store the number of visited nodes, `numvis`, whose initial value is zero (line 40). The directive CALL `code` provides the possibility of inserting source code to the logging program. This code does not modify the original program, it is only used in the CALL sentences. The string `cll` written after the reserved word `#pragma`, gives the indication to the C compiler that it is a directive for the CALL compiler.

A typical CALL experiment named `kps` is created in line 41. This experiment measures the execution time of the sentences between the beginning and ending directives (line 44). The sentence `pragma cll end` followed by the experiment name stops the CALL chronometer and the times are stored. The expression specified after the experiment `kps` (line 41) synthesizes the complexity function, which describes the behavior of the time. The constants kps[0],...,kps[i] are associated to the complexity function. The CALL syntax requires the use of the experiments name to index the constants associated with the complexity function. These constants will be evaluated with the LLAC tool.

Intuitively, we can say the time spent by the Branch-and-Bound algorithm is related to the number of visited nodes in the search space. Thus, our expression has to be written in terms of the variable `numvis` - number of visited nodes. However, the final value of numvis will only be known once the problem is solved, i.e., when the call to knap(0,M,0) finishes. Therefore, this specified value is not know (unknown) in the complexity expression and it will be evaluated at the end (posteriori).

The number of visited nodes (numvis) increases when two new subproblems obtained from each subproblem are studied, i.e., when the problems that include (line 29) and do not include the following object (line 30) are solved. We have to indicate this CALL increase with the corresponding directive, as it is shown in the line 31.

Finally, the directive `report` has to be added after the specification of all the experiments. This directive indicates to CALL compiler to generate a file with all the results obtained during the execution. In our case, we add it before the return sentence of the main function (line 46). The word `all` has been used to keep the results of all the experiments defined in the program. Alternatively, a list of identifiers can be used to specify the experiments whose results will be stored.

Once the code is annotated, the source code with the CALL directives can be compiled by any C compiler and can be executed exactly as if it had not been modified. The reason for this being that the compiler ignores all the CALL directives considering them as commentaries.

Supposing Algorithm 1 is stored in the file kpr.c, we compile it with CALL using the command:

```
> call kpr.c
```

As a result, two exit files: kpr.cll.c and kpr.cll.h are generated. Below, we compile these files with a C compiler, specifying where the necessary CALL files are (/usr/local/CALL/include):

```
> cc -o kpr kpr.cll.c
```

Upon executing the resulting program (kpr), a results file of the defined experiment is obtained: kpr.c.dat. Figure 4.13 shows the content of this file. The program was executed on a capacity knapsack M = 1.254.202 and the number of objects was N = 5000. We paid special attention to the last two lines of the file, where the results of our experiment appear. Since it is considered a sequential execution, the number of CPUS used is 1, with name 0. The number of visited nodes (numvis) was 261.134 and the execution time of the function knap() was 0,16 seconds. A description of all the information that appears in the file can be found in the user manual.

The parallel version of Branch-and-Bound algorithm in C follows a Master/Slave scheme. It was implemented using Message Passing. As in the sequential case, we wanted to study the number of visited nodes (numvis). This value was distributed between the slaves, because the master did not perform task bounding. Therefore, we will annotate the parallel code exactly the same as the sequential code, where the only necessary change is the line where the number of visited nodes is increased. The CALL directives have been added in the points where the problems are extracted from the local queue of each slave to study them (line 16) or to send them to another slave to be solved (line 42). Figure 4.14 shows the slave code with the CALL directives noted down. The Master code has not changed.

As the LLAC tool is an extension of R [49], it allows us to analyze the obtained results of the execution of the experiments with CALL. With the same

```
EXPERIMENT: "kps"
BEGIN_LINE: 115
END_LINE: 119
FORMULA: p 0 p 1 v 0 * +
INFORMULA: kps[0]+kps[1]*numvis
MAXTESTS: 131072
DIMENSION: 2
PARAMETERS:
NUMIDENTS: 1
IDENTS: numvis
OBSERVABLES: CLOCK
COMPONENTS: 1 numvis
POSTFIX_COMPONENT_0: 1
POSTFIX_COMPONENT_1: v 0
NUMTESTS: 1
SAMPLE:
  CPU  NCPUS  numvis    CLOCK
    0      1  261134.0  0.16491100
```

Fig. 4.13 Information of the File knr.c.dat

samples different types of representations to analyze the similarity between the expression with which we annotated the program and the obtained results can be accomplished.

The sequential executions of the knapsack problem were run on a AMD-DURON processor at 800 MHz with 256 Mb of memory. The experiment consisted of ten randomly generated knapsack problems with capacities in the range of [500, 5000]. Figure 4.15, generated with LLAC, shows the obtained results. Each one of the ten round points associated with each size between (500-5000) represents the number of visited nodes to solve that problem. The "x" linked with a dotted line represents the average of visited nodes. The continuous line represents a second degree polynomial. It indicates the average number of visited nodes is approximated to a parable. We conclude that the number of visited nodes was more scattered, when the number of objects were larger. This behavior could be due to the generator of random problems that we were using.

A very interesting parameter for studying the parallel Branch-and-Bound implementation is the "balanced work load" among the processors that take part in the execution. Figure 4.16 shows the distribution of visited nodes among eight processors for five executions of a randomly generated knapsack problem of size 50,000. The first two processors were AMD-DURON at 800 MHz and the rest at 500 MHz, all of them with 256 Mb of memory. It can

```
 while (1) {
   recv(source, flag);
   while (flag) {
     if (END_TAG){           // receive the finishing message
       inputPacket.recv(MASTER, END); return;
     }
     if (PBM_TAG){                 // the problem to be branched
        inputPacket.recv(source,    // source = slave or master:
                         auxSp,     // the initial subproblem
                         bestSol,   // the best solution value
                         sol);      //  the current solution
        auxSol = sol;
        bqueue.insert(auxSp);       // insert in the local queue
        while(!bqueue.empty()) {
          auxSp = bqueue.remove(); // pop from the local queue
 #pragma cll code numvis++;
          high = auxSp.upper_bound(pbm,auxSol);    // upper bound
          if ( high > bestSol ) {
            low = auxSp.lower_bound(pbm,auxSol);   // lower bound
            if ( low > bestSol ) {
              bestSol = low;
              sol = auxSol;
              outputPacket.send(MASTER,    // send to the Master:
                                SOLVE_TAG,    // problem solved
                                bestSol,      // best solution value
                                sol);         // solution vector
            }
            if ( high != low ) {
              rSlaves = bqueue.getNumberOfNodes();
              op.send(MASTER,
                      BnB_TAG,
                      high,     // upper bound
                      rSlaves);        // num. of slaves req.
              inputPacket.recv(MASTER,
                               nfSlaves,   // num. of slaves assig.
                               bestSol,    // updated best solution
                               rank {1,..., nfSlaves});
              if ( nfSlaves >= 0) {
                auxSp.branch(pbm,bqueue);   // branch
                for i=0, nfSlaves{          // send problems to slaves
                  auxSp = bqueue.remove();
 #pragma cll code numvis++;
                  outputPacket.send(rank,  // send to the slave:
                                    PBM_TAG, //   tag
                                    auxSp, //   subproblem
                                    bestSol, // best solution value
                                    sol); //   the solution vector
              } } // if nfSlaves == DONE the problem is bounded (cut)
        } } }
        outputPacket.send(MASTER, IDLE_TAG);  /idle slave
       }
       recv(source, flag);
}  } // while(1)
```

Fig. 4.14 Logged Slave Implementation With CALL

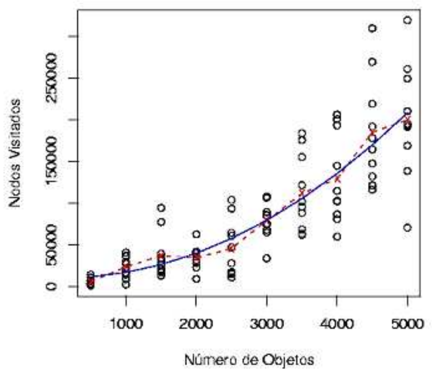


Fig. 4.15 Results of CALL Sequential Execution

be appreciated that processor zero does not visit any node, because it is the Master. We believe the first slave explores the width part of the search space because it is the fastest, and due to the size of the problem, not much work remains for the rest of the processors.

4.5.2 Study of the Communication Pattern

To analyze the communication pattern the tool PARAVER [11] is used. Paraver is a performance visualization and analysis tool that can be used to analyze MPI, OpenMP, MPI+OpenMP, Java, Hardware counters profile, Operating system activity and some other things. Paraver was developed in response to the need for having a qualitative global perception of the application behavior by visual inspection in order to focus on the detailed quantitative analysis of the problems. It provides useful information for improving the decisions on whether and where to invert the programming effort to optimize an application. Some Paraver features are the support for: detailed quantitative analysis of program performance, concurrent comparative analysis of several traces, fast analysis of very large traces, support for mixed message passing and shared memory, customizable semantics of the visualized information, sharing views of the tracefile and building of derived metrics.

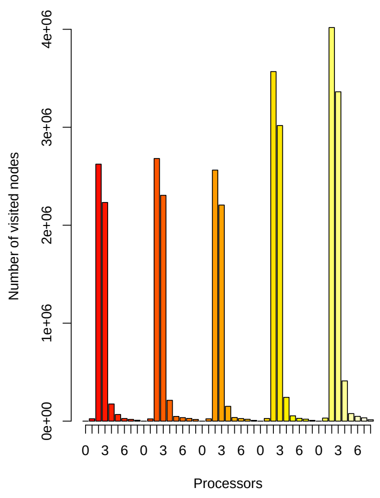


Fig. 4.16 Load Balance for 5 Executions of Size 50000

Paraver offers a graphical view to represent the behavior of the application against time in a way that easily conveys to the user a general understanding of the application behavior. The Paraver view consists of a time diagram with one line for each represented object. The types of objects displayed by Paraver are closely related to the parallel programming model concepts and to the execution resources. In the first group, the objects considered are: workload, application, task and thread. In the resources group, the objects considered are: system, node and CPU. Paraver displayed information consists of three elements: a time dependent value for each represented object, flags that correspond to punctual events within a represented object, and communication lines that relate to the displayed objects. The visualization module determines how each of these elements is displayed. Together with the possibility to change the types of representation, colors and scales.

Paraver specifies a trace format and some mechanisms on how the records and values encoded will be processed in the visualization. Every record specifies the object to which it refers (indicating application task and thread) and the absolute time in which it occurs. For each type of record, some additional fields can be encoded as desired by the user. These fields are:

- State records: include an integer value that is usually referred to as the state
- Event records: include a user event type and a user event value
- Relation/Communication records: include a communication tag and a communication size. Even if they have this name because it is usual to encode the tag and size of a MPI communication, Paraver does not rely on these semantics.

Next we define the tracing tool, called MPItrace, needed to accomplish the analysis of the communication pattern.

MPItrace

The MPItrace tool parallel code instruments that use the message passing (MPI) programming model. This tool generates a Paraver trace file where the basic activity of an MPI program is recorded. The MPItrace tool assumes each MPI process is single threaded. A tracefile represents a single MPI program run, thus it includes only one application with several tasks (as stated in the mpirun command) and one thread per task. The major encoding choices are:

- States: will record whether the thread is running, Waiting for Messages or doing I/O.
- Communication: The tag and size are set according to those in the calls. Physical communication is assumed to be identical to logical communication as it is not possible through the MPI instrumentation to find out when the actual data transfer takes place.
- Events: are used to tag the beginning and ending of MPI operations, such as Barriers, Broadcast, AlltoAll, and all kind of send-receive calls.

This instrumentation module provides the typical message passing visualization functionalities.

Using Paraver: an example

The following example of the Knapsack Problem has been called knap21.dat [69]: a knapsack with capacity $C = 102$ and the number of objects is $N = 8$

with the following weights and benefits

$$\begin{aligned} p_k &= \{15, 100, 90, 60, 40, 15, 10, 1\} \\ w_k &= \{2, 20, 20, 30, 40, 30, 60, 10\} \end{aligned}$$

It was run on a homogeneous cluster of four AMD Duron Processors PCs at 800 MHz with 256 Mbyte of memory each.

The file Makefile was modified to include the trace using PARAVER as follows:

```
PAPI_HOME    = /soft/parallel/papi-2.3.4.3 MPITRACE_HOME=
/soft/parallel/MPItrace MPITRACE_LIB = -L$(MPITRACE_HOME)/lib
PAPI_LIB     = -L$(PAPI_HOME)/lib

LFLAGS = -lmpitrace -lmpich -lpapi -lperfctr

CC = g++ CCFLAGS = -c -O3 -DBEOWULF=1 MPIFLAGS =
-I/soft/parallel/mpich/include/ -L/soft/parallel/mpich/lib/
-L$(MPITRACE_HOME)/lib -L$(PAPI_HOME)/lib

all: MainPar.par KNP.req.o KNP.pro.o main.o

MainPar.par: main.o MainPar.o KNP.req.o KNP.pro.o iopacket.o
    $(CC) $(MPIFLAGS) $^ -o $@ $(LFLAGS)

KNP.req.o: KNP.req.cc KNP.hh bbqueue.hh mlb.hh
    $(CC) $(MPIFLAGS) $(CCFLAGS) KNP.req.cc -o $@

KNP.pro.o: KNP.pro.cc KNP.hh bbqueue.hh mlb.hh
    $(CC) $(MPIFLAGS) $(CCFLAGS) KNP.pro.cc -o $@

iopacket.o: iopacket.cc iopacket.hh
    $(CC) $(MPIFLAGS) $(CCFLAGS) iopacket.cc -o $@

MainPar.o: MainPar.cc
    $(CC) $(MPIFLAGS) $(CCFLAGS) MainPar.cc -o $@

main.o: main.cc
    $(CC) $(MPIFLAGS) $(CCFLAGS) main.cc -o $@

clean:
    rm -f *.o MainPar.par
```

To run the application under MPItrace, the following ought to be specified:

```
mpitrace mpirun -np <n_procs> <filename>
```

The result of a simple analysis of the communication pattern is shown in the following snapshots.

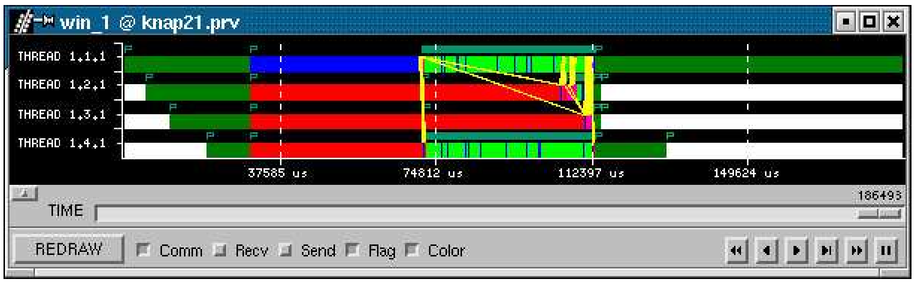
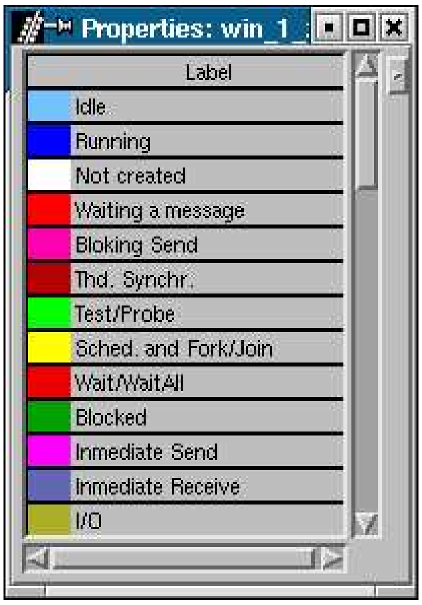


Fig. 4.17 Execution Trace


Properties: win_1

	Label
🟦	Idle
🟦	Running
⬜	Not created
🟥	Waiting a message
🟪	Bloking Send
🟥	Thd. Synchr.
🟩	Test/Probe
🟨	Sched. and Fork/Join
🟥	Wait/WaitAll
🟩	Blocked
🟪	Inmediate Send
🟪	Inmediate Receive
🟨	I/O

Fig. 4.18 Properties of Paraver

Figure 4.17 shows the execution trace, the meaning of each color is explained in figure 4.18. Figure 4.19 shows the trace of the same execution without colors, only the communication between processors are represented. As can

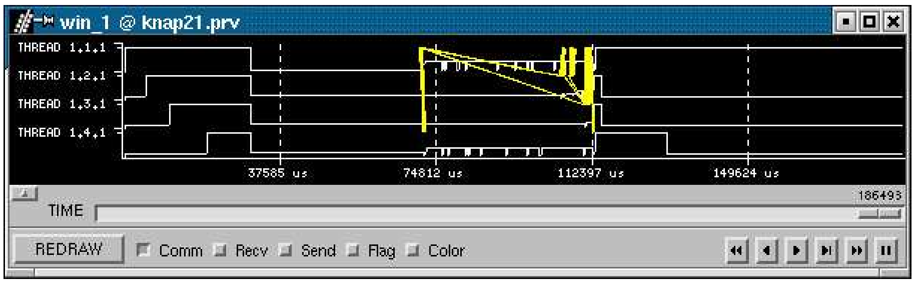


Fig. 4.19 Trace of the Communications between Processors

be appreciated, first the master processor communicates with slave processor number three, and processors one and two continue waiting for jobs. After that the master processor communicates with processor one it is this slave which sends the job to processor two. Before the execution finishes slave two communicates with the master.

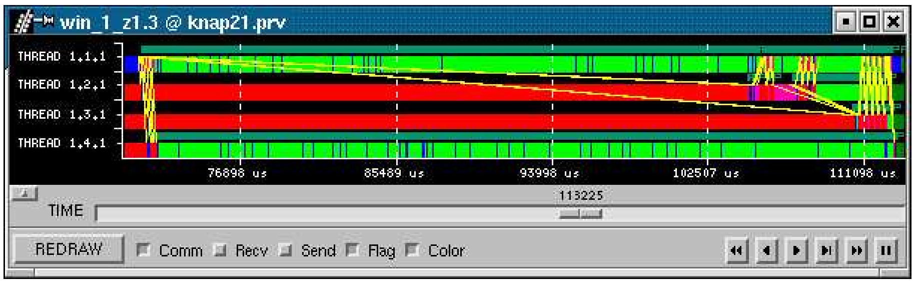


Fig. 4.20 Zoom of the Trace

Figure 4.20 shows the zoom of the trace previously presented. Figures 4.21 and 4.22 represent the zoom of the first and the second part of the trace respectively. Analyzing the figures, it can be concluded that the processors spend the most part of the time waiting for jobs or are inside a Probe loop and really spend very little time running tasks. This is due to the fact that the size of the problem is too small. But, it is not possible to run large size problems because the trace file generated is so large that it is not sustained by the cluster of PC.

Currently a version of PARAVER to run C++ code on OpenMP using a cluster of PC is not available. An account for the use of a machine in Barcelona,

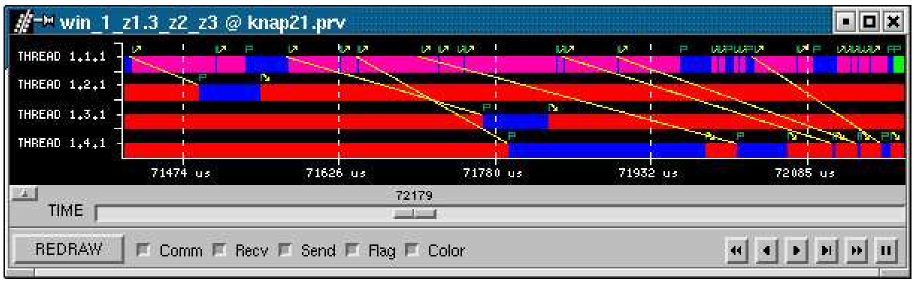


Fig. 4.21 Zoom of the First Part of the Communication

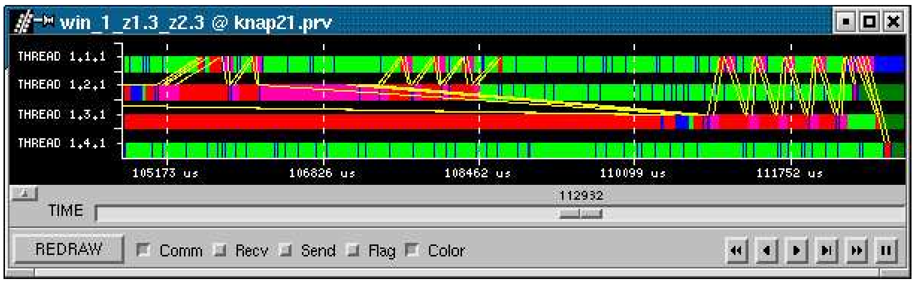


Fig. 4.22 Zoom of the Second Part of the Communication

to check if it is possible to work with the compiler of C++ and OpenMP, has been requested.

4.5.3 Some Conclusions of the Analysis Accomplished

The program performance analysis can be viewed as an integral part of program development. Thus, there are varying degrees of precision implicit in the expression `performance analysis`. One of the factors that influence the running time of a program is the size of the problem. We will represent the time a program takes to solve a problem of size n from the beginning of execution to the completion of execution by $T(n)$. Other factors to take into account are the I/O times. It does not make sense to count I/O statements with statements that only involve the CPU and RAM. Thus, as we are analyzing programs that include I/O statements, the initial expression include two terms: one for the calculation and one for I/O:

$$T(n) = T_{calc}(n) + T_{I/O}(n)$$

One clear difference between serial and parallel program performance analysis is that the runtime of a parallel program should depend on two variables: input size and number of processors. Thus, the previous expression adapted to a parallel program can be written as:

$$T(n,p) = T_{calc}(n,p) + T_{I/O}(n,p)$$

From the study the Master/Slave communication pattern accomplished, we have noted that the cost of the communication is significantly more expensive than the cost of the local calculation. Thus, in order to make reasonable estimations of parallel program performance, we count the cost of communication separately from the cost of calculation and I/O. That is,

$$T(n,p) = T_{calc}(n,p) + T_{I/O}(n,p) + T_{comm}(n,p)$$

Analyzing the behavior of the 0/1 Knapsack Problem for the algorithm we have implemented, when the number of objects is n, the number of subproblems to solve is 2^n. In the case of the resolution without solution vector, each subproblem has a current capacity, a value to store the next object and a profit. A constant value c of memory used is represented. Thus, the maximum amount of consumed memory can be calculated as $c \times 2^n$. To solve the problem obtaining the solution vector, each subproblem contains a vector to store the solution. In this case, the maximum memory needed is calculated by $n \times 2^n$.

To be precise, it has been discovered that the number of visited nodes in the search space for the randomly generated data of figure 4.15 is of the order n^2. Therefore, the amount of memory needed to solve this problem obtaining the solution vector is $n \times n^2$.

5 Conclusions and Future Research

In many combinatorial problems, the time taken to obtain a solution using a sequential computer is too high. This is unacceptable in applications of real time. One way to solve this inconvenience consists of using parallel computation. On a *parallel computer*, several processors cooperate to solve a problem simultaneously in a fraction of the time required by one processor. This idea is not new; it has been occurring naturally every time a large job needs to be done. However, in Computer Science, it is an idea whose time has come, thanks to the declining cost of processors and advances in microelectronics.

Among the key ingredients necessary for parallel computation are the architecture (or hardware), the operating system and programming language compilers (or software). However, most important of all is the *parallel algorithm*, without which no problems can be solved in parallel. In the same way that algorithms occupy a central place in Computer Science, parallel algorithms are at the heart of parallel computation.

The objective of this work was the development of a work methodology to approach the resolution of combinatorial optimization problems through the Branch and Bound technique using parallelism, parting from concrete cases in order to generalize a form of working to solve different problems. For this reason the *skeleton* concept was used. The skeleton concept implemented is not such low level programming as that presented by Cole [15]. However, the programming level of our skeleton proposal is not as complete as *Design Patterns*.

A bibliography review of the related works has been included in this memory. The tools which have a greater similarity to the work developed are: PPBB [92], PUBB [84], BOB [60] and PICO [32].

Tool	C	C++	MPI	PVM	OpenMP	Object oriented	Automatic load balancing	Easy of interface
MaLLBa::BnB	X	X	X		X	X		X
PPBB	X			X			X	X
PUBB	X	X		X		X	X	
Bob++	X	X		X				
PICO		X	X			X		

Table 5.1 Comparison between the Studied Tools

The main characteristics of the related tools including our MaLLBa::BnB proposal are summarized in the table 5.1. First column represents the related tools studied. Columns two and three summarize the programming languages used to implement these tools. Columns four and five specify the use of distributed memory with MPI or PVM, and column six represents the use of shared memory with OpenMP for the parallel implementations. Column seven shows the appropriate use of oriented object methodology. In column eight we specify the tools that provide load balancing strategies. Finally, the ease of use of some of the tools interface is represented in column nine.

Some skeletons for the Branch-and-Bound strategy are presented in this dissertation: a sequential skeleton, two parallel skeletons using message passing and a shared memory skeleton. The high level programming offered has to be emphasized. The schemes used can be easily transformed to solve other combinatorial optimization problems. When MaLLBa is used, as with any other object oriented system, the user has to put each piece of code in the right position. Once the `Problem`, `Solution`, and `Subproblem` classes have been modified by filling in the basic data and defining the functionalities, and inserted into the code, the skeleton provides for a sequential solver and three parallel solvers without additional user effort. An advantage of our proposal is that the user does not need to be an expert in parallelism.

As a work methodology the following steps are proposed:

1.- Represent the data structure that will compose the Problem, Subproblem and Solution.

2.- Obtain the sequential solver, the parallel solver and check if the results obtained are satisfactory.

3.- In the case that the results are not satisfactory, analyze the parallel proposals and implements an AD-HOC resolution.

The main objective of MaLLBa::BnB is to simplify the task of researchers and users that have to implement algorithms using the Branch-and-Bound technique. It provides the user an added value not only in terms of the amount of code written, but also modularity and conceptual clarity. MaLLBa::BnB makes a balanced use of object oriented programming. In fact it provides the minimum necessary patterns to obtain a good computational efficiency and, at the same time, to perform a secure compilation process.

Furthermore the system allows the user the possibility of generating and experimenting with new algorithms that permit the integration of techniques. In this work, it has been shown with the Knapsack Problem that the integration of different techniques is possible.

The obtained computational results suggest that for a large number of processors the proposed algorithms for message passing are not scalable. This is not a general trend in Branch-and-Bound algorithms. The lack of scalability in our results could be a consequence of the communications overload. One line of work to improve this deficiency is to parameterize the number of subproblems to solve by each processor before performing any request.

Other future research lines to improve the scalability of message passing scheme consists of improving the memory management. In the message passing algorithms that we present, all the information related to a subproblem is grouped in the subproblems that travel between processors. In particular,

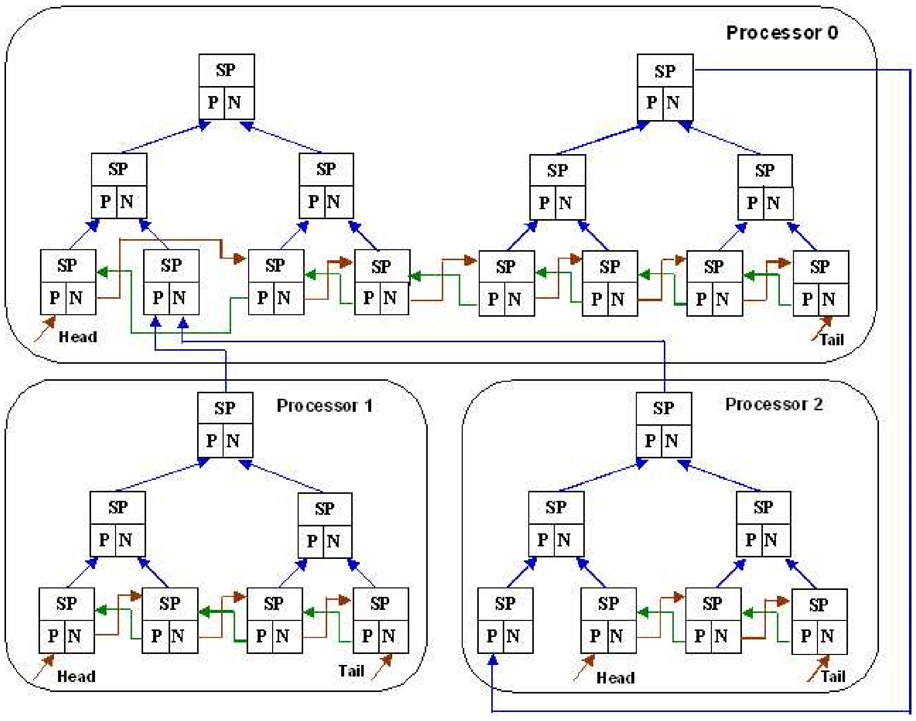


Fig. 5.1 Total Distributed Queue

the path of decisions taken by the original problem down to that subproblem in particular. Optimization is possible here since the subproblems share some common past. The natural solution necessarily implies the distribution of the decision vector among the different processors and the introduction of mechanisms to recover that distributed information. Along these lines, we are developing a total distributed algorithm where there is no difference between Master or slave processors. For this, two additional pointers in each subproblem to link the parent node with its two sibling nodes. Applying Branch-and-Bound technique, to free a tree node, both children of this node ought to specify that the best solution will not be found in these branchs. For this scheme we will implement the data structure shown in figure 5.1, where each processor manages its own data structure.

In the case of shared memory, the algorithm presented is very simple and easy to understand. In this sense, we can conclude that the OpenMP paradigm is more manageable than that of the MPI. It is also possible to implement a new skeleton to fuse OpenMP and MPI. This approach will allow a better management of the computational resources, especially in hybrid share-distributed memory architecture. The scheme to develop consists of the following idea: each multiprocessor is composed by two different kind of threads: ones to accomplish communication tasks with its neighboring multiprocessors and the other to manage the shared memory subproblems queue using OpenMP. The communication between neighboring multiprocessors is accomplished through the communication threads using the MPI tool. In this case, each multiprocessor only communicates with the two nearest multiprocessors (in a circular way), and it is not allowed to communicate with any other. The figure 5.2 shows this future research line.

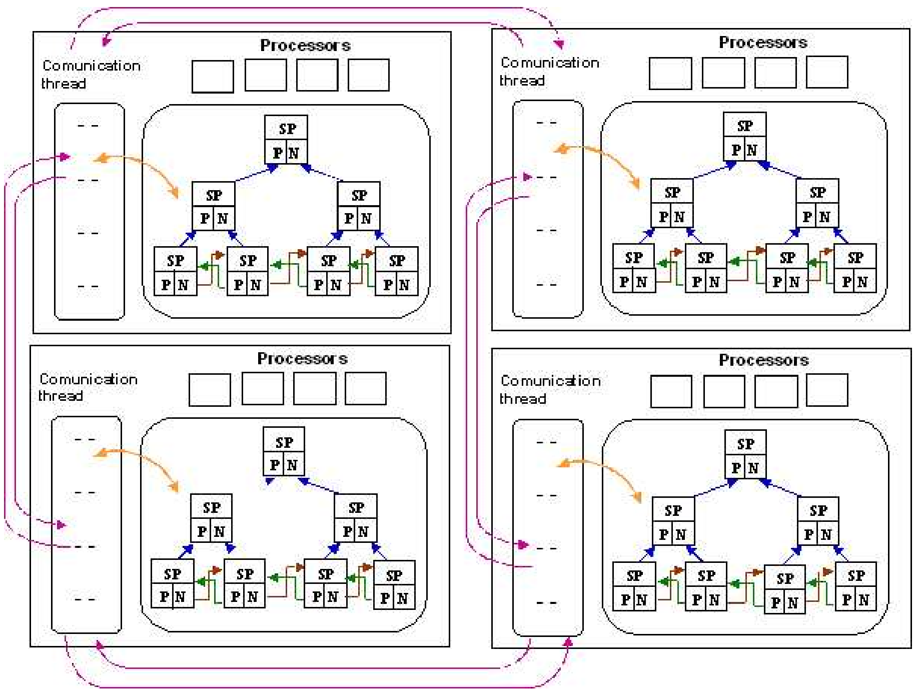


Fig. 5.2 OpenMP-MPI Scheme

From the analysis accomplished, we have discovered that the number of visited nodes in the search space for the randomly generated data is of the order n^2. Therefore one challenge is to try to generate a problem with enough grain to achieve some increase in parallel performance. We are also looking at more suitable problems to be solved using the Branch-and-Bound technique.

We conclude that the choice of programming language used to develop MaLLBa was not the most appropriate in terms of the available tools. It is difficult to find OpenMP for C++, PARAVER for C++ and even CALL-LLAC work on C code. Therefore we have developed a version in C of the MaLLBa::BnB tool. It has also been required an account to Cepba to try to run MaLLBa::BnB for shared memory skeleton with PARAVER.

Appendix A
Data Structure Code of BnB Queue

```
/**********************************************************************
* FILE:  bbqueue.hh                                                   *
* OBJECTIVE: Implementation of a dynamic queue to control the problems *
*            flow produced by the branch method.                      *
* CONTENTMENT: -- Class Node: It permit us to instance an object that  *
*                 represents a node of the tree.                      *
*              -- Class bandbQueue: Abstract Class that represents     *
*                 a queue.                                            *
*              -- Class branchQueue: Queue used during branch phase.   *
* NOTE: Library implemented with templates to permit the use of both   *
* classes Subproblem and Solution whose type will depend on the       *
* implemented example.                                                *
* For the utilization of this library should be declared the classes   *
* SubProblem and Solution that represent to a subproblem and a solution*
* in the Branch-and-Bound technique.                                  *
**********************************************************************/
```

```
#ifdef SUNFIRE6800
    #include <iostream>
    #include <cassert>
    using namespace std;
#else
  #include <iostream.h>
#endif

enum searchMethod {LIFO=0, FIFO=1, PRIORITY=2};

#define BBNODE BBNode<SP>


/*********************************************************************
* CLASS BBNODE: Implementation of the class that represents a node    *
*********************************************************************/
template <class SP> class BBNode {
  protected:
    SP sp;              // Subproblem to solve.
    BBNODE* next;       // Pointer to the next element of the queue
    BBNODE* previous;   // Pointer to the previous element of the queue
  public:
    BBNode ();       // Constructor.
    // SetThings
    void setSubProblem (const SP& sp);
    void setNext (BBNODE* const next);
    void setPrevious (BBNODE* const previous);
    // GetThings
    const SP& getSubProblem () const;
    BBNODE* getNext () const;
    BBNODE* getPrevious () const;
    ~BBNode () {};   // Destructor.
};


/*********************************************************************
* CLASS BANDBQUEUE: Abstract class to represent a generic queue.      *
*********************************************************************/
template <class SP> class bandbQueue {
  protected:
    BBNODE* head;     // Pointer to the first element of the queue.
    BBNODE* tail;     // Pointer to the last element of the queue.
    int numberOfNodes;  // Used by Master to know the number of
                        // nodes into the queue.
    searchMethod sm;  // establish the search method
  public:
    bandbQueue () {head = tail = NULL; numberOfNodes = 0; sm = LIFO;};
    void setNumberOfNodes (int initialValue);
    void setSearchMethod (searchMethod im);
    int getNumberOfNodes () const;
    searchMethod getSearchMethod () const;
    bool empty ();
    void decreaseCount ();
    void increaseCount ();
    void setTail (BBNODE* t);
    BBNODE* getHead () const;
    BBNODE* getTail () const;
    virtual ~bandbQueue () {};
};


/**********************************************************************
* CLASS BRANCHQUEUE: Class to generate the subproblems queue used     *
*                    during branch phase.                             *
**********************************************************************/
template <class SP> class branchQueue: public bandbQueue<SP> {
  public:
    branchQueue ();
    int compare ( const SP& sp1, const SP& sp2);
    void insert (const SP& sp);
    const SP& remove ();
    void insertAtBack (const SP& sp);
    void insertIntermediate (const SP& sp);
    const SP& removeFromFront ();
    const SP& removeFromBack ();
    void print ();
    void clear ();
    ~branchQueue () {};
};


//***** IMPLEMENTATION OF MEMBER METHODS  *****//

//***** CLASS BBNODE *****//
/**************************************************************
* Node: Constructor class Node. Initialization of member      *
*       variables to null.                                    *
**************************************************************/
template <class SP> BBNode<SP>::BBNode () {
  next = previous = NULL;
}


/**************************************************************
* setSubProblem: Member function that assign value to the     *
*                subproblem stored in the node.               *
* Input: sp -- Value to assign to the member variable sp.     *
**************************************************************/
template <class SP>
void BBNode<SP>::setSubProblem (const SP& sp1) {
  this->sp = sp1;
}
```

```
/************************************************************
* getSubProblem: Member function that permit obtain the value *
*                of the member variable sp.                 *
* Output:  -- Return the value of the member variable sp.     *
************************************************************/
template <class SP>
const SP& BBNode<SP>::getSubProblem() const {
  return sp;
}

/************************************************************
* setNext: Establish the link of the node in the queue during *
* the branch phase.                                           *
* Input: next1 -- Next element into the queue.                *
************************************************************/
template <class SP>
void BBNode<SP>::setNext(BBNODE* const next1) {
  this->next = next1;
}

/************************************************************
* setPrevious: Establish the link of the node in              *
*              the branch queue.                              *
* Input: previous1 -- Previous element in the queue.          *
************************************************************/
template <class SP>
void BBNode<SP>::setPrevious (BBNODE* const previous1) {
  this->previous = previous1;
}

/************************************************************
* getNext: Obtain the direction of next node in the queue     *
* Output:  -- Return a pointer to the next node in the queue  *
************************************************************/
template <class SP>
BBNODE* BBNode<SP>::getNext () const {
  return next;
}

/************************************************************
* getPrevious: Obtain the direction of the previous node of   *
*              the queue.                                     *
* Output:  -- Return a pointer of the previous node of
*             the queue la cola                               *
************************************************************/
template <class SP>
BBNODE* BBNode<SP>::getPrevious () const {
  return previous;
}
```

```
//***** CLASS BANDBQUEUE *************************************

template <class SP>
void bandbQueue<SP>::setTail (BBNODE* t) {
  tail = t;
}


/***********************************************************
* setCount: Establish the number of nodes not solved into     *
*           the queue                                        *
* Input: initialValue -- Number of nodes of the queue.        *
***********************************************************/
template <class SP>
void bandbQueue<SP>::setNumberOfNodes (int initialValue) {
  numberOfNodes = initialValue;
}


/***********************************************************
* setSearchMethod: Establish the insert method into           *
* the problems queue.                                         *
* Input: method -- enum type searchMethod.                    *
***********************************************************/
template <class SP>
void bandbQueue<SP>::setSearchMethod (searchMethod im) {
  sm = im;
}


/***********************************************************
* getNumberOfNodes: Obtain the value of the counter that      *
* indicates the number of nodes into the queue.               *
* Output:  -- Integer to indica the number of subproblems into*
*             the queue.                                      *
***********************************************************/
template <class SP> int bandbQueue<SP>::getNumberOfNodes () const
{
  return numberOfNodes;
}


/***********************************************************
* getSearchMethod: Obtain the search method specified.        *
************************************************************/
template <class SP> searchMethod
bandbQueue<SP>::getSearchMethod()const{
  return sm;
}
```

```
/************************************************************
* empty: Indicate if the queue is empty or not.             *
* Output:  -- Value boolean = true => Empty queue.          *
************************************************************/
template <class SP> bool bandbQueue<SP>::empty () {
  if (head == NULL)
    return true;
  return false;
}


/************************************************************
* decreaseCount: Decrease the counting of the number of nodes *
* in the queue.                                             *
************************************************************/
template <class SP > void bandbQueue<SP>::decreaseCount () {
  numberOfNodes--;
}


/************************************************************
* increaseCount: Increase the counting of the number of nodes *
*                in the queue.                              *
************************************************************/
template <class SP>
void bandbQueue<SP>::increaseCount () {
  numberOfNodes++;
}


/************************************************************
* getHead: Function to obtain the direction of the first      *
*          element inserted into the queue.                 *
* Output:  -- Pointer to the head of the queue.             *
************************************************************/
template <class SP>
BBNODE* bandbQueue<SP>::getHead() const {
  return head;
}


/************************************************************
* getTail: Function to obtain obtener the direction of the    *
*          last element into the queue.                     *
* Output:  -- Pointer to the tail.                          *
************************************************************/
template <class SP>
BBNODE* bandbQueue<SP>::getTail() const {
  return tail;
}
```

```
//***** CLASS BRANCHQUEUE ************************************

/************************************************************
* Constructor by defect.                                    *
************************************************************/
#ifdef SUNFIRE6800
  template <class SP> branchQueue<SP>::branchQueue(){
  }
#else
  template <class SP>
  branchQueue<SP>::branchQueue():bandbQueue<SP>::bandbQueue(){
  }
#endif


/************************************************************
* compare: Compare two elements of the queue.               *
* Input:   sp1 -- Subproblem to compare.                    *
*          sp2 -- Subproblem to compare.                    *
************************************************************/
template< class SP >
int branchQueue<SP>::compare( const SP &sp1, const SP &sp2 )
{
  return (sp1 > sp2) ? -1 : ((sp1 < sp2)? 1:0);
}


/************************************************************
* insertAtBack: Insert an element at the end of the list.   *
* Input:        sp -- Subproblem to insert into the list.   *
************************************************************/
template< class SP >
void branchQueue<SP>::insertAtBack( const SP &sp )
{
  BBNODE *n = new BBNode< SP >;
  assert (n != NULL);
  n->setSubProblem(sp);
  if ( empty() )
    head = tail = n;
  else {
    n->setPrevious(tail);
    tail->setNext(n);
    tail = n;
  }
  increaseCount();
}
```

```
/************************************************************
* insertIntermediate: Insert an element into the list in     *
*                     a concrete position.                   *
* Input: sp -- Subproblem to insert into the queue.          *
************************************************************/
template< class SP >
void branchQueue<SP>::insertIntermediate( const SP &sp )
{
  BBNODE *n = new BBNode <SP>;
  assert (n != NULL);
  n->setSubProblem(sp);
  if ( empty() )
    head = tail = n;
  else {
    BBNODE *current = head;
    while ( (current->getNext() != 0)
            && (compare( current->getSubProblem(), sp ) == -1) )
      current = current->getNext();
    if ((current == head)
        && (compare(current->getSubProblem(),sp) != -1))
    {
      n->setNext(head);
      head->setPrevious(n);
      head = n;
    }
    else if ((current == head)
             && (compare (current->getSubProblem(),sp) == -1)) {
      n->setPrevious(head);
      head->setNext(n);
      tail = n;
    }
    else if ((current->getNext() == 0)
             && (compare(current->getSubProblem(),sp) == -1)) {
      n->setPrevious(tail);
      tail->setNext(n);
      tail = n;
    }
    else {
      n->setPrevious(current->getPrevious());
      n->setNext(current);
      (current->getPrevious())->setNext(n);
      current->setPrevious(n);
  } }
  increaseCount();
}
```

```
/************************************************************
* insert: Insert an element into the list depending on       *
* the search method used.                                    *
* Input: sp -- Subproblem to insert into the list.           *
************************************************************/
template< class SP >
void branchQueue<SP>::insert( const SP &sp )
{
  if (sm == PRIORITY)
    insertIntermediate(sp);
  else
    insertAtBack(sp);
}

/**************************************************************
* removeFromFront: Extract a node from the beginning of the list*
* Output:  -- Return the subproblem stored into the node.      *
**************************************************************/
template< class SP > const SP&branchQueue<SP>::removeFromFront() {
  if ( !empty() ) {
    BBNODE *aux = head;
    if ( head == tail )
      head = tail = 0;
    else {
      (head->getNext())->setPrevious(0);
      head = head->getNext();
    }
    decreaseCount();
    return aux->getSubProblem();
} }

/************************************************************
* removeFromBack: Extract the node from the end of the list.  *
* Output:  -- Return the subproblem stored in the node.       *
************************************************************/
template< class SP >
const SP& branchQueue<SP>::removeFromBack()
{
  if ( !empty() ) {
    BBNODE *aux = tail;
    if ( head == tail )
      head = tail = 0;
    else {
      BBNODE *current = head;
      while ( current->getNext() != tail )
        current = current->getNext();
      tail = current;
      current->setNext(0);
    }
    decreaseCount();
    return aux->getSubProblem();
} }
```

```
/************************************************************
* remove: Extract a node from the list whose position depending*
*         on the search method.                             *
* Output:  -- Return the value of the subproblem stored in     *
*             the node.                                     *
************************************************************/
template< class SP >
const SP& branchQueue<SP>::remove()
{
  if (sm == LIFO)
    return removeFromBack();
  return removeFromFront();
}

/************************************************************
* print: Show the subproblems into the list.                *
************************************************************/
template< class SP > void branchQueue<SP>::print() {
  if ( !empty() ) {
    BBNODE *current = head;
    cout << "La lista es: ";
    while ( current != 0 ) {
      cout << current->getSubProblem() << ' ';
      current = current->getNext();
    }
    cout << "\n\n";
} }

/************************************************************
* concatqueue: Link the current queue with a provided queue.   *
* Input:  -- bqueue1: Queue generated by a processor to be     *
*            linked to the master queue.                    *
************************************************************/
template< class SP > void
branchQueue<SP>::concatqueue(branchQueue<SP> &bqueue1) {
  NODE* aux;
  aux = bqueue1.getHead();
  if ( head == tail ) {
    setHead(bqueue1.getHead());
    setTail(bqueue1.getTail());
    setNumberOfNodes(bqueue1.getNumberOfNodes());
  }
  else {
    aux->setPrevious(tail);
    tail->setNext(aux);
    setTail(bqueue1.getTail());
    setNumberOfNodes( getNumberOfNodes() + bqueue1.getNumberOfNodes() );
  }
  bqueue1.setHead(NULL);
  bqueue1.setTail(NULL);
  bqueue1.setNumberOfNodes(0);
}
```

```
/************************************************************
* clear: Empty the subproblems list.                        *
************************************************************/
template< class SP > void branchQueue<SP>::clear() {
  while ( !empty() ) {
    BBNODE *aux = head;
    if ( head == tail )
      head = tail = 0;
    else
    {
      delete aux;
      (head->getNext())->setPrevious(0);
      head = head->getNext();
    }
    decreaseCount();
} }
```

Appendix B
Applying MaLLBa::DnC Skeleton to QuickSort Method

Quicksort [41, 51] is a divide and conquer method for sorting. It works by partitioning the array to sort into two parts, then sorting the parts independently. The key point of the method is a partition procedure, which must rearrange the array to make the following three conditions hold:

1. The element pivot = a[i] is its final place in the array for some index i.
2. All the elements in a[first], ..., a[i-1] are less than or equal to a[i].
3. All the elements in a[i+1], ..., a[last] are greater than a[i].

At this point the same method is applied recursively to the subproblems a[first], ..., a[i-1] and a[i+1], ..., a[last]. Let us show how to instantiate this sorting algorithm using MaLLBa::DnC Skeleton. The following classes are required to the user:

Problem. An array A of items to be sorted. It is defined in Knapsack.hh.

Solution. An array A of items sorted. It must provide the following method:

```
requires class Solution {
  public:
    vector<Knapsack::Number> l;

    Solution ();
    void combine (const Problem& pbm, const Auxiliar& aux,
                  const vector<Solution>& subsols);

    friend bool operator==(const Solution& sol1, const Solution& sol2);
    friend bool operator!=(const Solution& sol1, const Solution& sol2);
    Solution& operator=(const Solution& sol);
    friend ostream& operator<< (ostream& os, const Solution& sol);
    friend istream& operator>> (istream& is, Solution& sol);
    friend opacket& operator<< (opacket& op, const Solution& sol);
    friend ipacket& operator>> (ipacket& ip, Solution& sol);
};
```

Fig. B.1 QS::Solution

- `combine(pbm, sps, sol)`: implements an algorithm to put together partial solutions in order to obtain the solution of the original problem.

```
void Solution::combine (const Problem& pbm, const Auxiliar& aux,
                        const Vector<Solution>& subsols) {
  for (Knapsack::Number i = 0; i < subsols[0].l.size(); i++)
    l.push_back(subsols[0].l[i]);
  for (Knapsack::Number i = 0; i < aux.l.size(); i++)
    l.push_back(aux.l[i]);
  for (Knapsack::Number i = 0; i < subsols[1].l.size(); i++)
    l.push_back(subsols[1].l[i]);
}
```

Fig. B.2 QS::Solution::combine

```
opacket& operator<< (opacket& op, const Solution& sol) {
  op << (int)sol.l.size();   // Pack the size of the list.
  for (unsigned i = 0; i < sol.l.size(); i++)
    op << sol.l[i];
  return op;
}
```

```
ipacket& operator>> (ipacket& ip, Solution& sol) {
  int size, value;
  ip >> size;   // Unpack the size of the list.
  assert(size >= 0);
  sol.l.clear();
  for (int i = 0; i < size; i++) {
    ip >> value;
    sol.l.push_back(value);
  }
  return ip;
}
```

Fig. B.3 Solution::Operators

```
Solution& Solution::operator=(const Solution& sol) {
  l = sol.l;
  return *this;
}
```

```
bool operator==(const Solution& sol1, const Solution& sol2) {
  return(sol1.l == sol2.l);
}
```

```
bool operator!=(const Solution& sol1, const Solution& sol2) {
  return(sol1.l != sol2.l);
}
```

```
ostream& operator<< (ostream& os, const Solution& sol) {
  os << "0-" << sol.l.size()-1 << ": ";
  for (Knapsack::Number i = 0; i < sol.l.size(); i++)
    os << sol.l[i] << ' ';
  os << endl;
  return os;
}
```

```
istream& operator>> (istream& is, Solution& sol) {
  Knapsack::Number n, value;
  is >> n;
  for (Knapsack::Number i = 0; i < n; i++) {
    is >> value;
    sol.l.push_back(value);
  }
  return is;
}
```

Fig. B.4 Solution::Operators

Subproblem. It represents the structure of problems to be solve. This class must provide the following functionalities:

```
requires class SubProblem {
   public:
     vector<Knapsack::Number> l;

     SubProblem ();
     void initSubProblem (const Problem& pbm);
     bool easy ();
     void solve (Solution& sol);
     void divide (const Problem& pbm, vector<SubProblem>& subpbms,
                  Auxiliar& aux);

     friend bool operator==(const SubProblem& sp1, const SubProblem& sp2);
     friend bool operator!=(const SubProblem& sp1, const SubProblem& sp2);
     SubProblem& operator=(const SubProblem& sp);

     friend ostream& operator<< (ostream& os, const SubProblem& sp);
     friend istream& operator>> (istream& is, SubProblem& sp);
     friend opacket& operator<< (opacket& op, const SubProblem& sp);
     friend ipacket& operator>> (ipacket& ip, SubProblem& sp);
 };
```

Fig. B.5 QS::SubProblem

- `initSubProblem(pbm)`: generates the first subproblem from the original problem.

```
void SubProblem::initSubProblem (const Problem& pbm) {
  for (Knapsack::Number i = 0; i < pbm.N; i++)
    l.push_back(i);
}
```

Fig. B.6 QS::SubProblem::initSubProblem

- `easy(pbm)`: this boolean method returns true when the subproblem is considered simple, i. e., a subproblem is considered easy if it has only one item to sort.
- `solve(pbm)`: this method provides the resolution technique for simple subproblems.

```
bool SubProblem::easy () {
  return (l.size() <=  1);
}
```

Fig. B.7 QS::SubProblem::easy

```
void SubProblem::solve (Solution& sol) {
  sol.l = l;
}
```

Fig. B.8 QS::SubProblem::solve

- `divide(pbm, sps, sol)`: provides an algorithm to divide the original problem into smaller problem with the same structure.

```
void SubProblem::divide (const Problem& pbm, Vector<SubProblem>& subpbms,
                         Auxiliar& aux){
  SubProblem sp1;
  SubProblem sp2;

  double pivot = (double)((double)(pbm.p[l[0]]) / (double)(pbm.w[l[0]]));

  for (Knapsack::Number i = 0; i < l.size(); i++) {
    volatile double tmp = (double)((double)(pbm.p[l[i]])/(double)(pbm.w[l[i]]));
    if (tmp > pivot)
      sp1.l.push_back(l[i]);
    if (tmp == pivot)
      aux.l.push_back(l[i]);
    if (tmp < pivot)
      sp2.l.push_back(l[i]);
  }
  subpbms.push_back(sp1);
  subpbms.push_back(sp2);
}
```

Fig. B.9 QS::SubProblem::divide

```
SubProblem& SubProblem::operator=(const SubProblem& sp) {
  l = sp.l;
  return *this;
}
```

```
bool operator==(const SubProblem& sp1, const SubProblem& sp2) {
  return(sp1.l == sp2.l);
}
```

```
bool operator!=(const SubProblem& sp1, const SubProblem& sp2) {
  return(sp1.l != sp2.l);
}
```

```
ostream& operator<< (ostream& os, const SubProblem& sp) {
  os << "0-" << sp.l.size()-1 << ": ";
  for (Knapsack::Number i = 0; i < sp.l.size(); i++)
    os << sp.l[i] << ' ';
  os << endl;
  return os;
}
```

```
istream& operator>> (istream& is, SubProblem& sp) {
  Knapsack::Number n, value;
  is >> n;
  for (Knapsack::Number i = 0; i < n ; i++) {
    is >> value;
    sp.l.push_back(value);
  }
  return is;
}
```

Fig. B.10 SubProblem::Operators

Auxiliar. It is needed when the results of the division process produces a problem which does not have the same characteristic of the original one.

```
opacket& operator<< (opacket& op, const SubProblem& sp) {
  op << (int)sp.l.size();   // Pack size of the list.
  for (unsigned i = 0; i < sp.l.size(); i++)
    op << sp.l[i];
  return op;
}
```

```
ipacket& operator>> (ipacket& ip, SubProblem& sp) {
  int size, value;
  ip >> size;    // Unpack size of the list.
  assert(size >= 0);
  sp.l.clear();
  for (unsigned i = 0; i < size; i++) {
    ip >> value;
    sp.l.push_back(value);
  }
  return ip;
}
```

Fig. B.11 SubProblem::Operators

Appendix C
Using MaLLBa::BnB Skeleton to Solve Other Problems

C.1 THE TRAVELLING SALESMAN PROBLEM (TSP)

Let G = (V, E) be a directed completed graph with edge non negative costs $c[i, j]$ and $c[i, i] = \infty$. Let $|V| = N$ (with $N > 1$). A Hamiltonian directed cycle (HDC) of G is a directed cycle that includes every vertex in V. The cost of the HDC is the sum of the cost of the edges on the HDC. The Travelling Salesman Problem is to find a HDC of minimum cost.

In order to use LC-Branch-and-Bound to search the travelling salesperson state space tree, it is necessary to define a cost function $c(.)$ and two other functions $\hat{c}(.)$ and $u(.)$ such that $\hat{c}(R) \leq c(R) \leq u(R)$ for all nodes R. $c(.)$ is such that the solution node with least $c(.)$ corresponds to a shortest tour in G. One choice for $c(.)$ is:

$$\text{c(A)=} \begin{cases} \text{length of tour defined by the path from the root to } A \text{ if } A \text{ is a leaf} \\ \\ \text{cost of a minimum cost leaf in a subtree a if } A \text{ is not a leaf} \end{cases}$$

The use of good bounding functions will enable us to solve some problem instances in much less time than required by others. A good $\hat{c}(.)$ may be obtained by using the reduced cost matrix corresponding to G [44].

A row (column) is said to reduced if it contains at least one zero and all remaining entries are non-negative. A matrix is reduced if every row and column is reduced. Since every tour on a graph includes exactly one edge $< i, j >$ with $i = k$, $1 \leq k \leq N$ and exactly one edge $< i, j >$ with $j = k$, $1 \leq k \leq N$, subtracting a constant t from every entry in one column or one row of the cost matrix reduces the length of every tour by exactly t. A minimum cost tour remains a minimum cost tour following this subtraction operation. If t is chosen to be the minimum entry in *row i (column j)*, then subtracting it from all entries in *row i (column j)* will introduce a zero into *row i (column j)*. Repeating this as often as needed, the cost x may be reduced. The total amount subtracted from all the columns and rows is a lower bound on the length of a minimum cost tour and may be used as the value for the root of the state space tree. With every node in the travelling salesman state space tree we may associate a reduced cost matrix. Let A be the reduced cost matrix for node R. Let S be a child of R such that the tree edge *(R, S)* corresponds to including edge $< i, j >$ in the tour. If S is not a leaf then the reduced cost matrix for S may be obtained as follows:

(i) change all entries in *row i* and *column j* of A to ∞. This prevents the use of any more edges leaving vertex i or entering vertex j.

(ii) set *A(j, 1)* to ∞. This prevents the use of edge $< j, 1 >$.

(iii) reduce all rows and columns in the resulting matrix except for rows and columns containing only ∞.

Let the resulting matrix be B. Steps (i) and (ii) are valid as no tour in the subtree S can contain edges of the type $< i, k >$ or $< k, j >$ or $< j, 1 >$ (except for edge $< i, j >$). If r is the total amount subtracted in step (iii) then $c(S) = c(R) + A(i, j) + r$. For the upper bound function, we may use $u(R) = \infty$ for all nodes R.

Let us outline how to approximate the TSP using MaLLBa::BnB skeleton. The following classes are required to the user:

Problem. A completed directed weighted graph where each node represents a city to visit is represented in matrix of cost c and the number of cities to visit is N. The matrix obtained of the reduction of the initial cost matrix is represented by r, and ic represents the value of this reduction.

```
requires class Problem {
   private:
     Number N;  // Number of elements
     Matrix c;  // cost
     Matrix r;  // reduced matrix
     Number ic; // initial cost

   public:
     Problem ();
     ~Problem ();

     inline Direction direction() const { return Minimize;}

     //Setter
     inline void setN (Number n) { N = n;}
     inline void setc(Matrix& m) {c = m;}

     //Getter
     inline Number GetN() const {return N;}
     inline Matrix Getc() const {return c;}
     inline Number GetIC() const {return ic;}
     inline Number Getc(Number i, Number j) const {return c[i][j];}
     inline Matrix Getr() const {return r;}
     inline Number Getr(Number i, Number j) const {return r[i][j];}
     inline Number operator() (Number i, Number j) const {return c[i][j];}

     void reduce();

     Problem& operator=(const Problem& pbm);
     friend bool operator==(const Problem& pbm1, const Problem& pbm2);
     friend bool operator!=(const Problem& pbm1, const Problem& pbm2);
     friend ostream& operator<< (ostream& os, const Problem& pbm);
     friend istream& operator>> (istream& is, Problem& pbm);
     friend opacket& operator<< (opacket& op, const Problem& pbm);
     friend ipacket& operator>> (ipacket& ip, Problem& pbm);
 };
```

Fig. C.1 TSP::Problem

```
void Problem::reduce()
{
  Number sumrow = 0, sumcol = 0, min, i, j;

  for (i=0; i < N; i++)
  {
    min = (int)plus_infinity;
    for (j=0; j < N; j++)
      if (min != 0)
        if (c[i][j] < min)
          min = c[i][j];
    if ((min != 0) && (min != plus_infinity))
      for (j = 0; j < N; j++)
      // se compara con la c original la primera vez
        if (c[i][j] != plus_infinity)
          r[i][j] = c[i][j] - min
        else
          r[i][j] = plus_infinity;
    sumrow += min;
  }

  for (j = 0; j < N; j++)
  {
    min = (int)plus_infinity;
    for ( i = 0; i < N; i++)
      if (min != 0)
        if (r[i][j] < min)
          min = r[i][j];
    if ( (min != 0) && (min != (int)plus_infinity))
      for (i = 0; i < N; i++)
        if (r[i][j] != plus_infinity)
          r[i][j] = r[i][j] - min;
    sumcol += min;
  }
  ic = sumrow + sumcol;
}
```

Fig. C.2 TSP::Problem:reduce

```
Problem& Problem::operator=(const Problem& pbm)
{
  N = pbm.N;
  c = pbm.c;
  return *this;
}
```

```
bool operator==(const Problem& pbm1, const Problem& pbm2)
{
  return (((pbm1.N == pbm2.N) &&
           (pbm1.c == pbm2.c)
          )?true:false
         );
}
```

```
bool operator!=(const Problem& pbm1, const Problem& pbm2)
{
  return (((pbm1.N != pbm2.N) &&
           (pbm1.c != pbm2.c)
          )?true:false
         );
}
```

Fig. C.3 TSP::Problem::operators

```
ostream& operator<< (ostream& os, const Problem& pbm)
{
  os << " N = " << pbm.N << endl;
  for(Number i = 0; i < pbm.N; i++) {
    for(Number j = 0; j < pbm.N; j++)
      os <<  " c[" << i << ", " << j << "] = " << pbm.c[i][j];
    os << endl;
  }
  os << endl;
  for(Number i = 0; i < pbm.N; i++) {
    for(Number j = 0; j < pbm.N; j++)
      os <<  " r[" << i << ", " << j << "] = " << pbm.r[i][j];
    os << endl;
  }
  os << endl;
  os << "ic : " << pbm.ic << endl;
  return os;
}
```

```
istream& operator>> (istream& is, Problem& pbm)
{
  Number c_ij;
  is >> pbm.N;
  for (Number i = 0; i < pbm.N; i++) {
    row r_i;
    row rr_i;
    for(Number j = 0; j < pbm.N; j++){
      is >> c_ij;
      r_i.push_back(c_ij);
      rr_i.push_back(0);
    }
    pbm.c.push_back(r_i);
    pbm.r.push_back(rr_i);
  }
  return is;
}
```

Fig. C.4 TSP::Problem:i/ostream

```
opacket& operator<< (opacket& op, const Problem& pbm)
{
  Number aux;
  op << pbm.N;
  for(Number i = 0; i < pbm.N; i++) {
    for(Number j = 0; j < pbm.N; j++){
      aux = pbm.c[i][j];
      op << aux;
    }
  }
  for(Number i = 0; i < pbm.N; i++) {
    for(Number j = 0; j < pbm.N; j++){
      aux = pbm.r[i][j];
      op << aux;
    }
  }
op << pbm.ic;
return op;
}
```

```
ipacket& operator>> (ipacket& ip, Problem& pbm)
{
  Number c_ij, r_ij;
  ip >> pbm.N;
  for (Number i = 0; i < pbm.N; i++) {
    row c_i;
    for(Number j = 0; j < pbm.N; j++){
      ip >> c_ij;
      c_i.push_back(c_ij);
    }
    pbm.c.push_back(c_i);
  }
 for (Number i = 0; i < pbm.N; i++) {
    row r_i;
    for(Number j = 0; j < pbm.N; j++){
      ip >> r_ij;
      r_i.push_back(r_ij);
    }
    pbm.r.push_back(r_i);
  }
  ip >> pbm.ic;
  return ip;
}
```

Fig. C.5 TSP::Problem:i/opacket

Solution. The value and a data structure for the minimum cost hamiltonian directed cycle. It is stored in the set of visited cities of class subproblem.

```
requires class Solution {
  public:
    Solution ();
    ~Solution ();

    friend ostream& operator<< (ostream& os, const Solution& sol);
    friend istream& operator>> (istream& is, Solution& sol);

    friend opacket& operator<< (opacket& op, const Solution& sol);
    friend ipacket& operator>> (ipacket& ip, Solution& sol);
};
```

Fig. C.6 TSP::Solution

Subproblem. This class will contain at least the data structures to store the following information: the set of visited nodes, the initial and final vertices of the current path, the cost associated to this path and the reduced of this subproblem.

- initSubProblem: Generates the first SubProblem from a given Problem.
- branch(): Generates all the subproblems resulting of including the link

  $< initial\ vertex,\ visited\ node >$

  on the path and inserts these subproblems in CONTAINER.
- upper_bound(): Computes a new upper bound.
- lower_bound(): Computes the sum of the costs of an arbitrary path from the initial vertex to the final vertex on the non-visited nodes plus the cost of the current path.

```
requires class SubProblem {
    private:
      Number iV,       // initial vertex
             eV;       // final vertex
      Number cost;     // current cost
      Matrix r;        // reduced matrix
      Tset visitedSet; // visited cities

    public:
      SubProblem ();
      ~SubProblem ();

      SubProblem (Number ini, Number end, Number cst);

      inline void setIV(Number iv) { iV = iv;}
      inline void setEV( Number ev) { eV = ev;}
      inline void setCost( Number c) { cost = c;}
      inline void setR( Matrix& rhs) { r = rhs;}
      inline void setR( int i, int j, int v) { r[i][j] = v;}
      inline void setVisitedSet( Tset st) { visitedSet = st;}
      inline void setVisitedSet( int i) { visitedSet.insert(i);}
      inline Number getIV() const { return iV;}
      inline Number getEV() const  { return eV;}
      inline Number getCost() const { return cost;}
      inline Number getr(Number i, Number j) const {return r[i][j];}
      inline Tset getVisitedSet() const { return visitedSet;}

      void initSubProblem (const Problem& pbm);
      Bound upper_bound (const Problem& pbm, Solution& ls);
      Bound lower_bound (const Problem& pbm, Solution& us);
      void branch (const Problem& pbm, branchQueue <SubProblem>& subpbms);

      Number reduce (const Problem& pbm, Number fil, Number col);
      friend Number reduce (const Problem& pbm, SubProblem& sp, Number r, Number c);

      // for the priority method is needed to overwrite the relational operators
      SubProblem& operator=(const SubProblem& sp);
      friend bool operator==(const SubProblem& sp1, const SubProblem& sp2);
      friend bool operator!=(const SubProblem& sp1, const SubProblem& sp2);
      friend bool operator<=(const SubProblem& sp1, const SubProblem& sp2);
      friend bool operator>=(const SubProblem& sp1, const SubProblem& sp2);
      friend bool operator<(const SubProblem& sp1, const SubProblem& sp2);
      friend bool operator>(const SubProblem& sp1, const SubProblem& sp2);

      friend ostream& operator<< (ostream& os, const SubProblem& sp);
      friend istream& operator>> (istream& is, SubProblem& sp);
      friend opacket& operator<< (opacket& os, const SubProblem& sp);
      friend ipacket& operator>> (ipacket& is, SubProblem& sp);
  };
```

Fig. C.7 TSP::SubProblem

```
void SubProblem::initSubProblem(const Problem& pbm)
{
  iV = 0;
  eV = 0;
  cost = pbm.GetIC();
  r = pbm.Getr();
  setVisitedSet(0);
}
```

Fig. C.8 TSP::SubProblem::initSubProblem

```
void SubProblem::branch (const Problem& pbm,
branchQueue<SubProblem>& subpbms)
{
  SubProblem sp_i;
  Number tmp = 0;
  Number reVi = 0;

  for (Number i = 0; i < pbm.GetN(); i++){

    Tset::iterator it = visitedSet.find(i);

    if (it == visitedSet.end()) { // i not belong to the set
      sp_i.setIV(i);
      sp_i.setEV(iV);  //sp_i.setEV(eV);
      sp_i.setVisitedSet(visitedSet);
      sp_i.setVisitedSet(i);
      reVi = getr(iV,i);

      sp_i.setR(r);
      tmp = 0;
      tmp = sp_i.reduce(pbm, iV, i); // value obtained of the
      reduction of matrix

      sp_i.setCost(cost + tmp + reVi); //actualized cost of the
      subproblem

      subpbms.insert(sp_i);  // sp_i is the generated subproblem
    }
  }
}
```

Fig. C.9 TSP::SubProblem::branch

```
Bound SubProblem::upper_bound (const Problem& pbm, Solution& sl)
{
  Bound c = plus_infinity;

  if (visitedSet.size() == pbm.GetN())
    c = cost;
  return c;
}
```

Fig. C.10 TSP::SubProblem::upper_bound

```
Bound SubProblem::lower_bound (const Problem& pbm, Solution& us)
{
  Bound u = cost;
  return u;
}
```

Fig. C.11 TSP::SubProblem::lower_bound

```
Number SubProblem::reduce (const Problem& pbm, Number fil, Number col)
{
  Number sumrow = 0, sumcol = 0, min, i, j, c = 0;

  for (j = 0; j < pbm.GetN(); j++)
    setR( fil, j, (int)plus_infinity);
  for (i = 0; i < pbm.GetN(); i++)
    setR( i, col, (int)plus_infinity);

  setR( col, 0, (int)plus_infinity);

  for (i=0; i < pbm.GetN(); i++) {
    if ( i != fil) {
      min = (int)plus_infinity;
      for (j=0; j < pbm.GetN(); j++)
        if (min != 0)
          if (getr(i,j) < min)
            min = getr(i,j);
      if ((min != 0) && (min != plus_infinity))
        for (j = 0; j < pbm.GetN(); j++)
          if (getr(i,j) != plus_infinity)
            setR( i, j, getr(i,j) - min);
      if (min != plus_infinity)
        sumrow += min;
  } }

  for (j = 0; j < pbm.GetN(); j++) {
    if (j != col) {
      min = (int)plus_infinity;
      for ( i = 0; i < pbm.GetN(); i++)
        if (min != 0)
          if (getr(i,j) < min)
            min = getr(i,j);
      if ( (min != 0) && (min != (int)plus_infinity))
        for (i = 0; i < pbm.GetN(); i++)
          if (getr(i,j) != plus_infinity)
            setR( i, j, getr(i,j) - min);
      if (min != plus_infinity)
         sumcol += min;
  } }

  c = sumrow + sumcol;
 return (c);
}
```

Fig. C.12 TSP::SubProblem::reduce

```
SubProblem& SubProblem::operator=(const SubProblem& sp)
{
  iV = sp.iV;
  eV = sp.eV;
  cost = sp.cost;
  r = sp.r;
  visitedSet = sp.visitedSet;
  return *this;
}

bool operator==(const SubProblem& sp1, const SubProblem& sp2)
{
  return (((sp1.iV == sp2.iV) &&
           (sp1.eV == sp2.eV) &&
           (sp1.cost == sp2.cost) &&
           (sp1.r == sp2.r) &&
           (sp1.visitedSet == sp2.visitedSet)
            )?true:false);
}

bool operator!=(const SubProblem& sp1, const SubProblem& sp2)
{
  return (((sp1.iV != sp2.iV) &&
           (sp1.eV != sp2.eV) &&
           (sp1.cost != sp2.cost) &&
           (sp1.r != sp2.r) &&
           (sp1.visitedSet != sp2.visitedSet)
            )?true:false);
}

bool operator>=(const SubProblem& sp1, const SubProblem& sp2)
{
  return (sp1.cost >= sp2.cost)?true:false;
}

bool operator<=(const SubProblem& sp1, const SubProblem& sp2)
{
  return (sp1.cost <= sp2.cost)?true:false;
}

bool operator>(const SubProblem& sp1, const SubProblem& sp2)
{
  return (sp1.cost > sp2.cost)?true:false;
}

bool operator<(const SubProblem& sp1, const SubProblem& sp2)
{
  return (sp1.cost < sp2.cost)?true:false;
}
```

Fig. C.13 SubProblem::Operators

```
ostream& operator<< (ostream& os, const Tset& s)
{
  copy(s.begin(), s.end(), ostream_iterator <Tset::value_type> (os, " "));
  return os;
}
```

```
ostream& operator<< (ostream& os, const Matrix& m)
{
  for (Number i = 0; i < m.size(); i++) {
    for(Number j = 0; j < m.size(); j++){
      os << m[i][j] << "  ";
    }
    os << endl;
  }
  return os;
}
```

```
ostream& operator<< (ostream& os, const SubProblem& sp)
{
  os << " iV = " << sp.iV << " eV = " << sp.eV;
  os << " Nvist = " << sp.visitedSet.size() << " cost = " << sp.cost << endl;
  os << " r = " << sp.r << endl;
  os << " visited Set = " << sp.visitedSet << endl;
  return os;
}
```

```
istream& operator>> (istream& is, SubProblem& sp)
{
  return is;
}
```

Fig. C.14 TSP::SubProblem::i/ostreams

```
opacket& operator<< (opacket& op, const Matrix& m) {
  Number sz;
  sz = m.size();
  op << sz;
  for (Number i = 0; i < m.size(); i++) {
    for(Number j = 0; j < m.size(); j++){
      op << m[i][j];
  } }
  return op;
}

opacket& operator<< (opacket& op, const SubProblem& sp) {
  Number j, ne ;
  op << sp.iV;
  op << sp.eV;
  ne = sp.visitedSet.size();
  op << ne;
  op << sp.cost;
  op << sp.r;
  for (Number i =0; i < sp.r.size(); i++){
    if (sp.visitedSet.count(i) == 1){
      op << i;
  } }
  return op;
}

ipacket& operator>> (ipacket& ip, SubProblem& sp) {
  Number c_ij, v, sz1, sz2;
  int count;
  ip >> sp.iV;
  ip >> sp.eV;
  ip >> sz1;
  ip >> sp.cost;
  ip >> sz2;
  for (Number i = 0; i < sz2; i++) {
    row r_i;
    for(Number j = 0; j < sz2; j++){
      ip >> c_ij;
      r_i.push_back(c_ij);
    }
    sp.r.push_back(r_i);
  }
  for (Number i = 0; i < sz1; i++) {
    ip >> v;
    sp.visitedSet.insert(v);
  }
  return ip;
}
```

Fig. C.15 TSP::SubProblem::i/opackets

C.2 THE N-QUEENS PROBLEM

The N-Queens Problem can be stated as follows: We will consider an NxN chessboard and try to place N queens so that no two attack, that is so that no two of them are on the same row, column or diagonal. Let us number the rows and columns of the chessboard 1 through N. The queens may also be numbered 1 through N. Since each queen must be on a different row, we can without loss of generality assume queen i is to be placed on row i. All solutions to the N-queens problem can therefore be represented as N-tuples (x_1, ...,x_N) where x_i is the column on which queen i is placed.

Let us outline how to implement the N-Queens Problem using the Branch-and-Bound skeleton:

- Problem: This class will contain the number of queens.

```
requires class Problem {
  private:
    Number N;   // Number of queens

  public:
    Problem ();
    ~Problem ();

    inline Direction direction() const { return Minimize;}

    //Setter
    inline void setN (Number n) {N = n;}

    //Getter
    inline Number getN () { return N;}

    //Operators
    Problem& operator= (const Problem& pbm);
    friend bool operator== (const Problem& pbm1, const Problem& pbm2);
    friend bool operator!= (const Problem& pbm1, const Problem& pbm2);

    //iostream and iopacket operators
    friend ostream& operator<< (ostream& os, const Problem& pbm);
    friend istream& operator>> (istream& is, Problem& pbm);
    friend opacket& operator<< (opacket& op, const Problem& pbm);
    friend ipacket& operator>> (ipacket& ip, Problem& pbm);
};
```

Fig. C.16 NQueen::Problem

```
Problem& Problem::operator=(const Problem& pbm)
{
    N = pbm.N;
    return *this;
}
```

```
bool operator==(const Problem& pbm1, const Problem& pbm2)
{
    return((pbm1.N == pbm2.N) ?true:false);
}
```

```
bool operator!= (const Problem& pbm1, const Problem& pbm2)
{
    return((pbm1 != pbm2) ?true:false);
}
```

```
ostream& operator<< (ostream& os, const Problem& pbm)
{
    os << " N = " << pbm.N << endl;
    return os;
}
```

```
istream& operator>> (istream& is, Problem& pbm)
{
    is >> pbm.N;
    return is;
}
```

```
opacket& operator<< (opacket& op, const Problem& pbm)
{
    op << pbm.N;
    return op;
}
```

```
ipacket& operator>> (ipacket& ip, Problem& pbm)
{
    ip >> pbm.N;
    return ip;
}
```

Fig. C.17 NQueen::Problem::operators

- Solution: This data structure will contain a column x_i where the queen i (fil i) is placed. It is stored in the vector s.

```
requires class Solution {
  private:
    vector<int> s;
  public:
    Solution ()
    ~Solution ();

    //Setter
    inline void insert(int s_i) { s.push_back(s_i);}

    //Getter
    inline int getS(int s_i) const { return s[s_i];}
    inline int getSize() const { return s.size();}

    //Operators
    Solution& operator= (const Solution& sol);
    friend bool operator== (const Solution& s1, const Solution& s2);
    friend bool operator!= (const Solution& s1, const Solution& s2);

    //iostream and iopacket operators
    friend ostream& operator<< (ostream& os, const Solution& sol);
    friend istream& operator>> (istream& is, Solution& sol);
    friend opacket& operator<< (opacket& op, const Solution& sol);
    friend ipacket& operator>> (ipacket& ip, Solution& sol);
}
```

Fig. C.18 NQueen::Solution

- Subproblem: This class will contain at least the data structures to store the following information: the current row, current column, current queen and the set of columns fixed.

```
requires class SubProblem {
  private:
    unsigned row,      // current row
           col,        // current column
           queen;      // current queen
    Tset visitedSet;   // column fixed
    Solution sol;      // current Solution
  public:
    SubProblem ();
    SubProblem (Number ini, Number end, Number cst);

    inline void setRow(unsigned row) {row = ROW;}
    inline void setCol(unsigned COL) {col = COL; }
    inline void setQueen(unsigned Q) {queen = Q; }
    inline void setVisitedSet(Tset st) {visitedSet = st; }
    inline void setVisitedSet(int i) {visitedSet.insert(i); }
    inline void setSol( Solution s) { sol = s;}
    inline void setSol( int i) { sol.insert(i);}

    inline unsigned getRow() const { return row;}
    inline unsigned getCol() const { return col;}
    inline unsigned getQueen() const { return queen;}
    inline Tset getVisitedSet() const { return visitedSet;}
    inline Solution getSol() const { return sol;}
    inline Number getNVisit() const { return sol.getSize();}

    void initSubProblem (const Problem& pbm);
    Bound upper_bound (const Problem& pbm, Solution& us);
    Bound lower_bound (const Problem& pbm, Solution& us);
    void branch (const Problem& pbm, container<SubProblem>& subpbms);

    //operators
    SubProblem& operator= (const SubProblem& sp);
    friend bool operator== (const SubProblem& sp1, const SubProblem& sp2);
    friend bool operator!= (const SubProblem& sp1, const SubProblem& sp2);

    friend ostream& operator<< (ostream& os, const SubProblem& sp);
    friend istream& operator>> (istream& is, SubProblem& sp);
    friend opacket& operator<< (opacket& op, const SubProblem& sp);
    friend ipacket& operator>> (ipacket& ip, SubProblem& sp);
 }
```

Fig. C.19 NQueen::SubProblem

```
void SubProblem::initSubProblem (const Problem& pbm)
{
   row = -1;
   col = -1;
   queen = -1;
   setVisitedSet(0);
   setSol(0);
}
```

Fig. C.20 NQueen::SubProblem::initSubProblem

```
Bound SubProblem::upper_bound(const Problem& pbm, Solution& us)
{
   if (row == -1) return 99999999;
   return pbm.GetN();
}
```

Fig. C.21 NQueen::SubProblem::upper_bound

```
Bound SubProblem::lower_bound(const Problem& pbm, Solution& s)
{
   Solution sl;
   int i;

   if (row == -1) return 0;
   for (i=0; i < (int)row; i++){
      sl.insert(i);
      if( (sol.getS(i) == (int)col) || ( abs(sol.getS(i) - col) == abs(i-row))) {
         s = sl;
         return getNVisit() - 1;
      }
   }
   s = sl;
   return getNVisit();
}
```

Fig. C.22 NQueen::SubProblem::lower_bound

```
void SubProblem::branch (const Problem& pbm, container<SubProblem>& subpbms)
{
   SubProblem sp_i;

   for (Number i = 0; i < pbm.GetN(); i++){
      Tset::iterator it = visitedSet.find(i);
      if (it == visitedSet.end()) { // i not belong to the set
         sp_i.setRow(row+1);
         sp_i.setCol(i);
         sp_i.setQueen(row+1);
         sp_i.setVisitedSet(visitedSet);
         sp_i.setVisitedSet(i);
         sp_i.setSol(sol);
         sp_i.setSol(i);

         subpbms.insert(sp_i);
      }
   }
}
```

Fig. C.23 NQueen::SubProblem::branch

References

1. S. G. Akl. *The Design and Analysis of Parallel Algorithms.* Prentice-Hall, 1989.

2. S. G. Akl. *Parallel Computation. Models and Methods.* Prentice-Hall, 1997.

3. E. Alba, F. Almeida, M. Blesa, J. Cabeza, C. Cotta, M. Díaz, I. Dorta, J. Gabarró, C. León, J. Luna, L. Moreno, C. Pablos, J. Petit, A. Rojas, and F. Xhafa. MALLBA: A library of skeletons for combinatorial optimisation. *Proceedings of the Euro-Par. Lecture Notes in Computer Science. Springer-Verlag*, 2400:927–932, 2002.

4. E. Alba, F. Almeida, M. Blesa, C. Cotta, M. Díaz, I. Dorta, J. Gabarró, J. González, C. León, L. Moreno, J. Petit, J. Roda, A. Rojas, and F. Xhafa. MALLBA: Towards a Combinatorial Optimization Library for Geographically Distributed Systems. *Actas de las XII Jornadas de Paralelismo*, pages 105–110, 2001.

5. F. Almeida, I. Dorta, F. García, J.A. González, D. González, C. León, J.L. Roda, C. Rodríguez, and F. Sande. Algoritmos Exactos de Optimización Combinatoria: Una Aproximación Geográficamente Distribuida.

Perspectivas del Paralelismo en Computadores. Actas de las XI Jornadas de Paralelismo, pages 271–275, 2000.

6. R. Batoukov and T. Sørevik. A Generic Parallel Branch and Bound Enviroment on a Network of Workstations. *In proceedings of Hiper-99*, pages 474–483, 1999.
7. R. Bellman. *Dynamic Programming.* Princeton U. P., 1957.
8. M. Benaïchouche, V.-D. Cung, S. Dowaji, B. Le cun, Th. Mautor, and C. Roucairol. Bob : Experiments with a branch-and-bound optimization library. In *IFORS'96, The International Federation of Operational Research Societies*, 1996.
9. M. Benchouche, V. D. Cung, S. Dowaji, B. Le Cun, T. Mautor, and C. Roucairol. Building a parallel branch and bound library, in solving combinatorial optimization problems in parallel. *Lecture Notes in Computer Science, Springer, 201*, 1054, 1996.
10. G. Brassard and P. Bratley. *Fundamentos de Algorítmia.* Prentice-Hall, 1997.
11. Cepba. Paraver. *http://www.cepba.upc.es/paraver*, 2000.
12. N. Christofides. *Graph Theory An Algorithmic Approach.* Academic Press Inc., 1975.
13. Ciemat. Centro de Investigaciones Energéticas, Medioambientales y Tecnológicas. *http://www.ciemat.es/informatica/index_supercom.html*, 1986.
14. M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* MIT Press, Cambridge, Mass, 1989.
15. M. Cole. Skeletal Parallelism Homepage. *http://www.dcs.ed.ac.uk/home/mic/skeletons.html*, 2000.
16. W. J. Cook, W. H. Cunningham, W. R. Pulleyblank, and A. Schrijver. *Combinatorial Optimization.* John Wiley & Sons, Inc, 1998.
17. V. D. Cung, S. Dowaji, B. Le Cun, T. Mautor, and C. Roucairol. Concurrent data structures and load balancing strategies for parallel branch-and-bound/A* algorithms. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 30:141–161, 1997.
18. V. D. Cung, M. Hifi, and B. Le Cun. Constrained Two-Dimensional Cutting Stock Problems. *Technical report n.97/020, PRISM, Université de Versailles*, 1997.

19. A. De Bruin, G. A. P. Kindervater, and H. W. J. M. Trienekens. Towards an abstract parallel branch and bound machine. *Lecture Notes in Computer Science*, 1054:145–153, 1996.

20. T. Decker, M.Fischer, R. Lling, and S. Tschke. A distributed load balancing algorithm for heterogeneous parallel computing systems. *Proc. of the 1998 Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'98), H. R. Arabnia (ed.), CSREA Press*, 2:933–940, 1998.

21. I. Dorta, C. León, and C. Rodríguez. Comparing MPI and OpenMP implementations of the 0-1 Knapsack Problem. *Parallel and Distributed Computing Practices (to appear).*

22. I. Dorta, C. León, and C. Rodríguez. Parallel Branch-and-Bound Skeletons: Message Passing and Shared Memory implementations. *Fifth International Conference on Parallel Processing and Applied Mathematics (PPAM2003) Proceedings Published in the Springer Verlag Series LNCS (to appear).*

23. I. Dorta, C. León, and C. Rodríguez. A comparison between MPI and OpenMP Branch-and-Bound Skeletons. *Proceeding of the Eighth International WorkShop on High-Level Parallel Programming Models and Supportive Environments. IEEE Computer Society*, pages 66–73, 2003.

24. I. Dorta, C. León, C. Rodríguez, G Rodríguez., and A. Rojas. Complejidad Algorítmica: de la Teoría a la Práctica. *Actas de las IX Jornadas de Enseñanza Universitaria de la Informática*, 2003.

25. I. Dorta, C. León, C. Rodríguez, and A. Rojas. MPI and OpenMP implementations of Branch-and-Bound Skeletons. *Parallel Computing 2003 (ParCo2003) Conference Proceedings will be published by Elsevier Science (to appear).*

26. I. Dorta, C. León, C. Rodríguez, and A. Rojas. Branch-and-Bound and Divide-and-Conquer Parallel Skeletons: an Integrated Approximation. *Technical Report DEIOC TR-03-06*, 2003.

27. I. Dorta, C. León, C. Rodríguez, and A. Rojas. Parallel Skeletons for Divide-and-Conquer and Branch-and-Bound Techniques. *Proceeding of the Eleventh Euromicro Conference on Parallel, Distributed and Network Based Processing. IEEE Computer Society*, pages 292–298, 2003.

28. I. Dorta, C. León, C. Rodríguez, and A. Rojas. Resolución del Problema de la Mochila 0/1 usando esqueletos Divide-y-Vencerás y Ramificación-y-Acotación. *Investigación Operacional*, 25:4–13, 2004.

29. I. Dorta, A. Rojas, C. León, and P. Dorta. Utilización de software en la docencia de técnicas algorítmicas. *Actas de las VII Jornadas de Enseñanza Universitaria de la Informática*, pages 190–195, 2001.

30. J. Eckstein. Parallel Branch and Bound Algorithms for General Mixed Integer Programming on the CM-5. *SIAM Journal on Optimization 4*, 1994.

31. J. Eckstein, W.E. Hart, and C.A. Phillips. Resource Management in a Parallel Mixed Integer Programming Package. *http://www.cs.sandia.gov/ISUG97/papers/eckstein.ps*, 1997.

32. J. Eckstein, C.A. Phillips, and W.E. Hart. PICO: An Object-Oriented Framework for Parallel Branch and Bound. *Rutcor Research Report*, 2000.

33. EPCC. Edinburgh Parallel Computing Center. *http://www.epcc.ed.ac.uk/computing/services/sun/*, 1990.

34. Message Passing Interface Forum. MPI: A message-passing interface standard. *International Journal of Supercomputing, Applications and High Performance Computing*, 8:3–4, 1994.

35. I. Foster. *Designing and Building Parallel Programs.* Addisson-Wesley Publishing Company, USA, 1th edition, 1995.

36. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: elements of reusable object-oriented software. *Addison-Wesley Longman Publishing Co., Inc., Boston, MA*, 1995.

37. F. García. Programación en Paralelo y Técnicas algorítmicas. *Tesis Doctoral. DEIOC. Universidad de La Laguna. In spanish*, 1995.

38. M. Garey and D. Jhonson. Computers and intractability: A guide to the theory of NP-Completeness. *Freeman*, 1979.

39. B. Gendron and T. G. Crainic. Parallel branch-and-bound algorithms: survey and synthesis. *Operations Research*, 42:1042–1066, 1994.

40. Company Hewlett Packard. Standard template library programmer's guide. *http://www.sgi.com/tech/stl/*, 1994.

41. C. A. R. Hoare. Algorithm 64: Quicksort. *Communications of the ACM*, 4:321, 1961.

42. R. M. Hord. *Understanding Parallel Supercomputing.* IEEE Press, 1995.

43. E. Horowitz and S. Sahni. Computing partitions with applications to the knapsack problem. *Journal of ACM*, 21:277–292, 1974.

44. E. Horowitz and S. Sahni. *Fundamentals of Computer Algorithms.* Computer Science Press, 1978.

45. T. Ibaraki. Finite State Representations of Discrete Optimization Problems. *SIAM J. Computing*, 2:193–212, 1973.

46. T. Ibaraki. Enumerative Approaches to Combinatorial Optimization, Part I. *Annals of Operations Research*, 10-11, 1987.

47. T. Ibaraki. Enumerative Approaches to Combinatorial Optimization, Part II. *Annals of Operations Research*, 11, 1988.

48. IBM. *COIN: Common Optimization INterface for Operations Research.* 2000.
http://oss.software.ibm.com/developerworks/opensource/coin/index.html.

49. R. Ihaka and R. Gentleman. R, A language for Data Analysis and Graphics. *Journal of Computational and Graphical Statistics*, 5: 3:299–314, 1996.

50. L. Joyanes. *Programación en C++. Algoritmos, estructuras de datos y objetos.* Mc Graw Hill, 2000.

51. T. Kielmann, R. Nieuwpoort, and H. Bal. Satin: Efficient Parallel Divide-and-Conquer in Java. *Euro-Par, Springer-Verlag*, pages 690–699, 2000.

52. B. Korte and J. Vygen. *Combinatorial Optimization Theory and Algorithms.* Springer, 2000.

53. L. Kronsjö and D. Shumsheruddin. *Advances in Parallel Algorithms. Chapter:The Divide-and-Conquer Paradigm as a Basis for Parallel Language Design.* John Wiley & Sons, 1992.

54. H. Kuchen. A Skeleton Library. *In Proceeding of Euro-Par 2002*, pages 620–629, 2002.

55. H. Kuchen. The Skeleton Library Web Pages. *http://danae.uni-muenster.de/lehre/kuchen/Skeletons*, 2002.

56. V. Kumar, A. Grama, A. Gupta, and G. Karypi. *Introduction to Parallel Computing. Design and Analysis of Algorithms.* Benjamin-Cummings, 1993.

57. L. Ladányi, T.K. Ralphs, and M.J. Saltzman. Implementing Scalable Parallel Search Algorithms for Data-intensive Applications. *The Proceedings of the International Conference on Computational Science*, 1:592–603, 2002.

58. E. L. Lawler and D. E. Wood. Branch and Bound Methods: A survey. *Operations Research*, 14:699–719, 1966.

59. B. Lecun. Bob++ Library, v0.2. *http://www.prism.uvsq.fr/ blec/Research/BOBO/index.html*, 2003.

60. B. Lecun and C. Roucairol. Bob : Branch and bound optimization library. In *INFORMS, Institute For OR and Management Science*, 1995.

61. B. Lecun and C. Roucairol. Bob : a unified platform for implementing branch-and-bound like algorithms. In *5th INFORMS Computer Science Technical Section Conference*, 1996.

62. B. Lecun and C. Roucairol. Experiments with the bob development environment. In *EUROXV/INFORMSXXXIV Conference*, 1997.

63. E. K. Lee and J. E. Mitchell. Computational experience of an interior point SQP algorithm in a parallel branch-and-bound framework. In H. Frenk *et al.*, editor, *High Performance Optimization*, chapter 13, pages 329–347. Kluwer Academic Publishers, Dordrecht, The Netherlands, 2000.

64. C. León Hernández. Parallel Library for Integration of the Divide and Conquer and Branch and Bound Techniques and its application in the resolution of the Nesting Problem. *http://nereida.deioc.ull.es/˜cleon/magos/index.html*, 2001.

65. LL-Cluster. The La Laguna Cluster for the MALLBA Project. *http://nereida.deioc.ull.es/˜cicyt/llbeowulf.html*, 1999.

66. R. Lüling and B. Monien. Load balancing for distributed branch and bound algorithm. *In Proceedings of 6th International Parallel Processing Symposium*, pages 543–548, 1992.

67. B. Mans and C. Roucairol. Performances des algorithmes branch-and-bound paralleles a strategie meilleur d&apos;abord. Technical Report RR-1716, Inria, Institut National de Recherche en Informatique et en Automatique.

68. S. Martello and P. Toth. An upper bound for the zero-one knapsack problem and a branch and bound algorithm. *European Journal of Operational Research*, 1:169–175, 1977.

69. S. Martello and P. Toth. *Knapsack Problems Algorithms and Computer Implementations.* John Wiley & Sons Ltd., 1990.

70. B. Monien and O. Vornberger. Parallel Processing of Combinatorial Search Trees. *Proceedings of the 1983 International Workshop on Parallel Algorithms and Architectures, Berlin*, pages 60–69, 1987.

71. D. Morales, J. Roda, F. Almeida, C. Rodríguez, and F. García. Integral Knapsack Problem: Parallel Algorithms and their implementations on Distributed Systems. *Proceedings of the 1995 International Conference on Supercomputing*, pages 218–226, 1995.

72. . OpenMP Architectur Review Board. OpenMP C and C++ Application Program Interface. Version 2.0. *http://www.openmp.org*, 2002.

73. P. S. Pacheco. *Parallel Programming with MPI.* Morgan Kaufmann Publishers, Inc, 1997.

74. C. A. Phillips, W. E. Hart, V. Leung, B. Carr, and J. Eckstein. PICO - Parallel Integer and Combinatorial Optimization. *http://www.cs.sandia.gov/ caphill/proj/pico.html*, 2000.

75. T.K. Ralphs and L. Ladányi. SYMPHONY: A Parallel Framework for Branch, Cut and Price. *http://www.branchandcut.org/SYMPHONY*, 2000.

76. G. Rodríguez. CALL: a complexity Analysis Tool. *Proyecto Fin de Carrera, Centro Superior de Informática, Universidad de La Laguna*, Junio 2002.

77. A. Rojas and C. León. Esqueletos Paralelos Divide y Vencerás. *DT-00-02. Departamento de Estadística, I.O. y Computación. Universidad de La Laguna.*, Diciembre, 2000.

78. R. A. Rushmeier and G. L. Nemhauser. Experiments with parallel branch-and-bound algorithms for the set covering problem. *Operations Research Letters*, 13(5):277–285, June 1993.

79. S. Sahm. *Data Structures, Algorithms and Applications in Java.* Mc Graw Hill, 2000.

80. F. Sande. *El Modelo de Computación Colectiva: Una Metodología Eficiente para la Ampliación del Modelo de Librería de Paso de Mensajes con Paralelismo de Datos Anidado.* Universidad de La Laguna, 1998.

81. S. Shah, G. Haab, P. Petersen, and J. Throop. Flexible control structures for parallelism in OpenMP. *Concurrency: Practice and Experience*, 12:1219–1239, 2000.

82. Y. Shinano, T. Fujie, Y. Ikebe, and R. Hirabayashi. Solving the Maximum Clique Problem Using PUBB. *Proc. of 12th International Parallel Processing Symposium and 9th Symposium on Parallel and Distributed Processing(IPPS'98)*, pages 326–332, 1998.

83. Y. Shinano, T. Fujie, and Y. Kounoike. Effectiveness of Parallelizing the ILOG-CPLEX Mixed Integer Optimizer in the PUBB2 Framework . *Euro-Par 2003, to appear*, 2003.

84. Y. Shinano, K. Harada, and R. Hirabayashi. Control Schemes in a Generalized Utility for Parallel Branch-and-Bound Algorithms. *Proc. of 11th International Parallel Processing Symposium (IPPS'97)*, pages 621–627, 1997.

85. Y. Shinano, M. Higaki, and R. Hirabayashi. A Generalized Utility for Parallel Branch and Bound Algorithms. *Proc. of the 7nd IEEE Symposium on Parallel and Distributed Processing (SPDP'95), IEEE Computer Society Press*, pages 392–401, 1995.

86. Y. Shinano, M. Higaki, and R. Hirabayashi. A Selection Rule for Parallel Branch and Bound Algorithms (In Japanese). *Transactions of the Society of Instrument and Control Engineers*, 32(9):1379–1387, 1996.

87. Y. Shinano, M. Higaki, and R. Hirabayashi. An Interface Design for General Parallel Branch-and-Bound Algorithms. *Parallel Algorithms for Irregularly Structured Problems(IRREGULAR'96), Springer LNCS 1117*, pages 277–284, 1996.

88. M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI - the Complete Reference.* MIT Press, 2nd edition, 1998.

89. B. Stroustrup. *The C++ Programming Language.* Adisson-Wesley, 3th edition, 1997.

90. A. S. Tanenbaum and M. van Steen. *Distributed Systems Principles and Paradigms.* Prentice Hall, 2002.

91. S. Tschöke and N. Holthöfer. A new parallel approach to the constrained two-dimensional cutting stock problem. 980:285–300, 1995.

92. S. Tschöke and T. Polzer. *Portable Parallel Branch-and-Bound Library.* User manual. Technical report D-33095, University of Paderborn, http://www.uni-paderborn.de/ ppbb-lib, 1995.

93. S Tschöke, M Räcke, R. Lüling, and B Monien. Solving the Traveling Salesman Problem with a Distributed Branch-and-Bound Algorithm on a

1024 Processor Network. *Proc. of the 9th International Parallel Processing Symposium (IPPS'95)*, pages 182–189, 1995.

94. B. Wilkinson and M. Allen. *Parallel Programming Techniques and Applications Using Networked Workstations and Parallel Computers.* Prentice Hall, 1999.

95. C. Xu, S. Tschöke, and B. Monien. Performance Evaluation of Load Distribution Strategies in Parallel Branch and Bound Computations. *Proc. of the 7th IEEE Symposium on Parallel and Distributed Processing, SPDP'95*, pages 402–405, 1995.