Resolución de Problemas en Paralelo

Coromoto León Hernández

Profesora Titular de Lenguajes y Sistemas Informáticos
Departamento de Estadística, Investigación Operativa y Computación
Universidad de La Laguna

http://nereida.deioc.ull.es/~cleon

Resumen

En este trabajo se abordará la resolución de algunos problemas de optimización utilizando *esqueletos* paralelos basados en las técnicas Divide y Vencerás y Ramificación y Acotación. Las implementaciones están hechas utilizando el lenguaje de programación C++, y el usuario sólo tiene que especificar el tipo de problema, el tipo de solución y las características específicas de la técnica algorítmica que quiere utilizar. Esta información se combina con los esqueletos de resolución que se proporcionan para obtener programas secuenciales y paralelos tanto para arquitecturas de Memoria Compartida, como para las de Memoria Distribuida.

Palabras Clave

Técnicas Algorítmicas, Ramificación y Acotación, Divide y Vencerás, Esqueletos, Paso de Mensajes, Memoria Compartida.

1. Introducción

El ser humano siente una gran curiosidad por comprender el funcionamiento de las cosas que le rodean, lo que ha traído como consecuencia el desarrollo del método científico. El proceso empieza con una observación cuidadosa del fenómeno a estudiar y la formulación del problema. A continuación se aborda la construcción del modelo científico, en general matemático, que intenta abstraer la esencia del problema real. Llegado este punto, se propone la hipótesis de que el modelo es una representación lo suficientemente precisa de las características esenciales de la situación como para que las soluciones obtenidas sean válidas también para el problema real. Esta hipótesis se verifica y modifica mediante las pruebas adecuadas. Sin embargo, el objetivo no es sólo encontrar una solución al problema en cuestión, la meta es encontrar la mejor solución, la *solución óptima*. El estudio de este tipo de problemas es parte del campo de acción de la rama de las Matemáticas Aplicadas denominada *Investigación Operativa*. Concretamente, la *Programación Matemática* trata de resolver problemas de decisión en los que se deben determinar acciones que optimicen un determinado objetivo, pero satisfaciendo ciertas limitaciones en los recursos disponibles. La *Optimización Combinatoria* es una rama de la Programación Matemática que intenta la resolución de problemas de optimización caracterizados por tener un número finito de posibles soluciones … pero dicho número es muy grande. Y a este tipo de problemas es al que hace referencia el título de este trabajo.

Fijado el tipo de problema, se trata de proporcionar el conjunto de operaciones y procedimientos que deben seguirse para resolverlo. Al conjunto de todas las operaciones a realizar y el orden en el que deben efectuarse se le denomina *Algoritmo*. El diseño de un algoritmo que resuelva un problema es, en general, una tarea difícil. Sin embargo, no es necesario partir de cero; se puede facilitar esta labor recurriendo a técnicas conocidas de diseño de algoritmos, es decir, a esquemas muy generales que pueden adaptarse a un problema particular (*Técnicas Algorítmicas*). Muchos problemas pueden resolverse buscando una solución fácil y directa. Este método, denominado de *fuerza bruta*, puede ser muy directo, pero con un análisis superficial se concluye que los algoritmos no son eficientes. El esquema algorítmico más sencillo es el llamado *Divide y Vencerás*, basado en la descomposición de un problema en subproblemas. Hay problemas cuya solución sólo puede hallarse mediante un proceso de búsqueda. A pesar de lo complejas que son las operaciones de búsqueda, su uso adecuado mediante el esquema de *Búsqueda con Retroceso* (o backtracking) permite ganar gran eficiencia respecto a soluciones de fuerza bruta. La técnica de *Ramificación y Acotación* constituye una variante del método de retroceso para problemas donde se trata de encontrar el valor máximo o mínimo de cierta función objetivo.

El papel que juegan los *ordenadores* en el proceso de diseño e implementación de los algoritmos para resolver problemas de optimización es fundamental. Sin embargo, aún existen problemas tan grandes que no son resolubles en un tiempo razonable. Una forma de proceder consiste en utilizar *paralelismo*. Existen diferentes tipos de máquinas paralelas comerciales. Las que pueden ejecutar múltiples instrucciones sobre múltiples datos (MIMD) se dividen en dos subclases, dependiendo de la relación entre la memoria y los procesadores. Las máquinas paralelas de *Memoria Compartida* permiten a cualquier procesador acceder a cualquier módulo de memoria global, mientras que las de *Memoria Distribuida* conectan procesadores individuales con sus propios módulos de memoria e implementan los accesos a memoria remota mediante el *paso de mensajes* entre procesadores. Para cada una de estas arquitecturas se ha desarrollado software paralelo, pero de forma dependiente de la misma.

La solución exacta de la mayoría de los problemas combinatorios 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. El esquema de trabajo de estas técnicas se puede generalizar para desarrollar herramientas software que permitan el análisis, diseño e implementación de resolutores para problemas concretos. 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.

En este trabajo se presentan los esqueletos orientados a objeto DnC y BnB para la resolución de problemas de optimización combinatoria con las técnicas Divide y Vencerás y Ramificación y Acotación respectivamente. La implementación de los esqueletos se ha realizado en C++. Se proporciona el código secuencial y el código paralelo de la parte invariante de los resolutores para ambos paradigmas. Además, se pone a disposición del usuario la plantilla de la parte que ha de rellenar. Las clases que componen dicha plantilla sirven para establecer la relación entre el resolutor principal y el problema instanciado. Una vez que el usuario ha instanciado su problema, obtiene gratis un resolutor paralelo de su algoritmo, sin realizar ningún esfuerzo adicional.

Los esqueletos proporcionan modularidad al diseño de algoritmos exactos, lo que supone una gran ventaja respecto a la implementación directa del algoritmo desde el principio, no sólo en términos de reutilización de código sino también en metodología y claridad de conceptos.

Se han encontrado en la revisión bibliográfica varias herramientas para la implementación paralela de algoritmos generales de Ramificación y Acotación. PPBB (Portable Parallel Branch-and-Bound Library) [14] y PUBB (Parallelization Utility for Branch and Bound algorithms) [13] proponen implementaciones en el lenguaje de programación C. BOB [10] y PICO (An Object-Oriented Framework for Parallel Branch-and-Bound) [4] están desarrollados en C++ y en ambos casos se proporciona una jerarquía de clases que el usuario ha de extender para resolver su problema particular. En cuanto a los sistemas de programación paralelos orientados a la metodología Divide y Vencerás podemos citar los siguientes: Cilk [7] basado en el lenguaje C y Satin [8] basado en Java. Sin embargo, no hemos encontrado ninguna herramienta que permita trabajar de forma integrada con más de una técnica algorítmica, y esta es una de las principales aportaciones de nuestra propuesta. Los esqueletos DnC y BnB forman parte del proyecto MaLLBa [1,2,3,9].

El resto del artículo se organiza como sigue. En la segunda sección se presenta la arquitectura MaLLBa y las interfaces particulares de DnC y BnB. A continuación, en el tercer epígrafe, se describen los esqueletos secuenciales y paralelos. En la cuarta sección se utilizan DnC y BnB de forma integrada en la resolución del Problema de la Mochila 0/1. Finalmente, se presentan las conclusiones y los trabajos futuros.

2. Descripción de la arquitectura MaLLBa

Se denomina *esqueleto algorítmico* a 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.

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 la Orientación 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.

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*, a las que se 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 (porque 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.

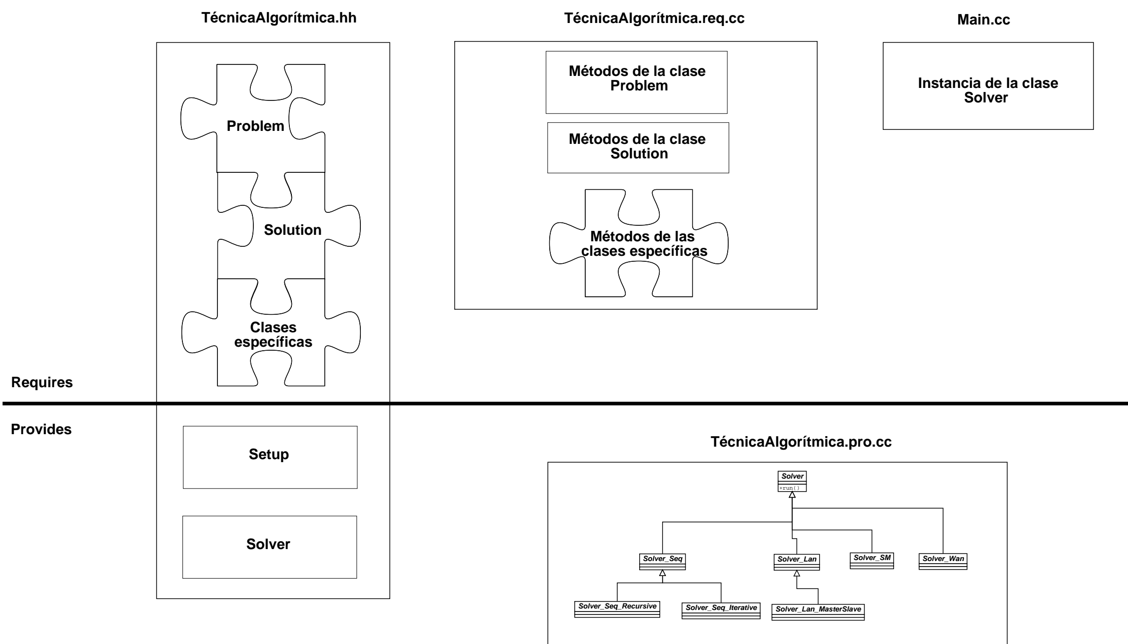


Figura 1. Arquitectura de los Esqueletos MaLLBa.

La Figura 1 presenta la arquitectura del esqueleto. En el fichero con extensión `.hh` se definen las cabeceras de las clases requeridas y proporcionadas. El fichero `.pro.cc` contiene las implementaciones C++ de los patrones de resolución, mientras que el fichero `.req.cc` es el que ha de implementar el usuario.

2.1. 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. En los epígrafes siguientes se describen las clases que el usuario ha de implementar cuando utilice DnC y BnB.

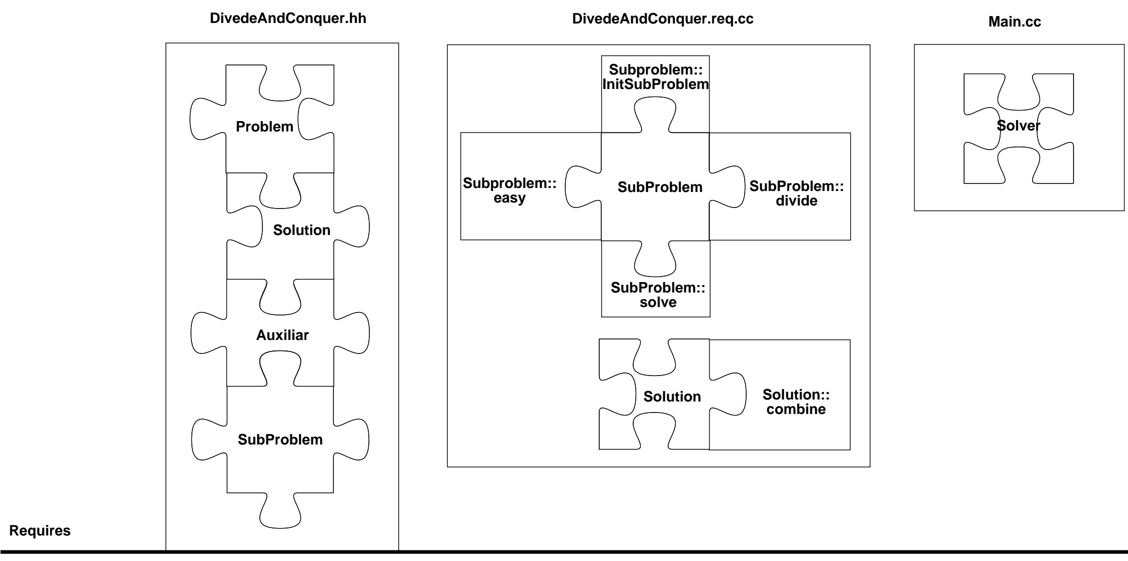


Figura 2. Clases requeridas por el esqueleto de Divide y Vencerás.

2.1.1. Interfaz del esqueleto de Divide y Vencerás. Para instanciar un esqueleto DnC además de las clases `Problem` y `Solution`, es necesario implementar las clases `SubProblem` y `Auxiliar` (véase la Figura 2).

- La clase `SubProblem`: representa una partición de un problema en partes disjuntas. Esta clase se ha introducido en aras de una mayor eficiencia en la implementación. La alternativa hubiera sido representar a los subproblemas para que fueran del mismo tipo que los problemas. Esta clase debe proporcionar las siguientes funcionalidades:
  - `initSubProblem(pbm)`: inicializa el subproblema de partida a partir del problema original.
  - `easy(pbm)`: determina si un subproblema es lo sufientemente pequeño para ser considerado simple.
  - `solve(pbm,sol)`: proporciona el algoritmo básico de resolución para subproblemas fáciles.
  - `divide(pbm,sps,aux)`: genera una lista de subproblemas.
- La clase `Auxiliar`: en algunos casos es necesario el uso de una clase auxiliar. Su necesidad se hace patente cuando como resultado de dividir el problema se obtiene una partición que no constituye un subproblema.

- Además la clase `Solution` debe proporcionar la siguiente funcionalidad:
  - o `combine(pbm,sols,aux)`: dado un conjunto de soluciones parciales de subproblemas, este método ha de combinarlas para crear una nueva solución parcial más general.

2.1.2. Interfaz del esqueleto de Ramificación y Acotación. 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`. Así pues, el esqueleto BnB a diferencia de DnC sólo requiere que el usuario le suministre dicha clase (véase la Figura 3). Además el usuario debe especificar en la definición de la clase `Problem` si el problema a resolver es de máximo o de mínimo.

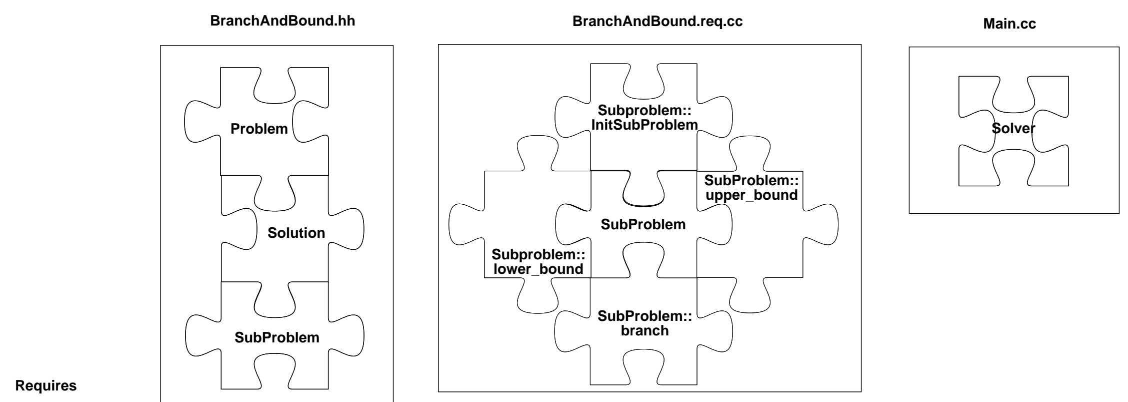


Figura 3. Clases requeridas por el esqueleto de Ramificación y Acotación.

- La clase `Subproblem`: representa el área de soluciones no exploradas. Esta clase debe proporcionar las siguientes funcionalidades:
  - o `initSubProblem(pbm)`: inicializa el subproblema de partida a partir del problema original.
  - o `solve(pbm)`: este método booleano devuelve cierto cuando el área de soluciones factibles a ser explorada es vacía.
  - o `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.
  - o `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.
  - o `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.

2.2. 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, primero el mejor, etc.
- `Solver`: esta clase está comprendida por ella misma y sus hijas `Solver_Seq`, `Solver_Lan` y `Solver_Wan`. Implementa la estrategia a seguir: Divide y Vencerás, Ramificación y Acotación, etc., y mantiene información actualizada del estado de la exploración durante la ejecución. La ejecución se lleva a cabo mediante una llamada al método `run()`.

Para elegir un patrón de resolución determinado, el usuario debe instanciar en el método `main()` la clase correspondiente. El siguiente código muestra un ejemplo de instanciación de un resolutor secuencial, para el esqueleto de Ramificación y Acotación.

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

      //instancia del esqueleto secuencial
       Solver_Seq sv(pbm);
      //se ejecuta el esqueleto secuencial
      sv.run();
      //se obtiene la mejor solución
      bs = sv.bestSolution();
      //se obtiene la solución
      sol = sv.solution();
}
```

En primer lugar, se indica que se va a utilizar un esqueleto de Ramificación y Acotación. A continuación se crea una instancia del esqueleto secuencial (`Solver_Seq`). Para que se ejecute, es necesario realizar una llamada al método `run()`. Se invoca al método `bestSolution()` para obtener la mejor solución. La solución se obtiene utilizando el método `solution()` que proporciona el esqueleto.

3. Implementación de los patrones de resolución

En esta sección se describe la forma en la que se han diseñado e implementado los patrones de resolución generales tanto en secuencial como en paralelo.

Las implementaciones de DnC y BnB se han realizado mediante un esquema *Maestro-Esclavo*. En el patrón maestro-esclavo un procesador *Maestro* (*Master*) controla la actividad de un grupo de procesadores *Esclavos* (*Slaves*), asignando el trabajo que se ha de realizar en paralelo y ocupándose también de las operaciones de entrada/salida.

Dado un problema de grandes dimensiones, se divide y se distribuye sobre los esclavos disponibles. Cada Esclavo calcula los resultados parciales de manera independiente y se los envía al Maestro (véase la Figura 4).

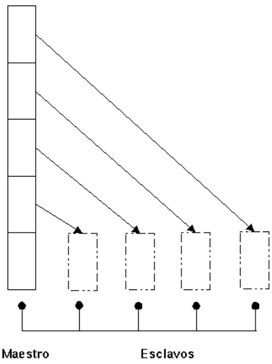


Figura 4. Esquema Maestro-Esclavo.

Las tareas del Maestro consisten en distribuir trabajo a los componentes Esclavos, arrancar la ejecución de los esclavos y generar el resultado final usando los resultados de los diferentes esclavos. Un esclavo ha de implementar la tarea que le indique el maestro.

3.1. Esqueletos Divide y Vencerás. La estrategia Divide y Vencerás consiste en descomponer un problema en subproblemas más simples del mismo tipo, resolverlos de forma independiente y una vez obtenidas las soluciones parciales combinarlas para obtener la solución del problema original. El siguiente código refleja la estructura general de un algoritmo basado en la estrategia Divide y Vencerás:

```
procedure DandC(pbm, sol)
local var aux;
begin
      if easy(pbm) then solve(pbm)
      else begin
            divide(pbm, subpbm, aux);
            for i := 1 to numProblem() do
                  DandC(subpbm[i], subsol[i]);
            combine(subsol, aux, sol);
      end
end;
```

En el caso de DnC se ha utilizado el patrón Maestro/Esclavo dos veces, uno para la *Fase de División* del problema original en subproblemas y una segunda vez en la *Fase de Combinación* de soluciones parciales para obtener la solución del problema original. La división del trabajo entre las subtareas y la evaluación conjunta de los resultados de las subtareas están completamente separados del procesamiento individual de cada subtarea.

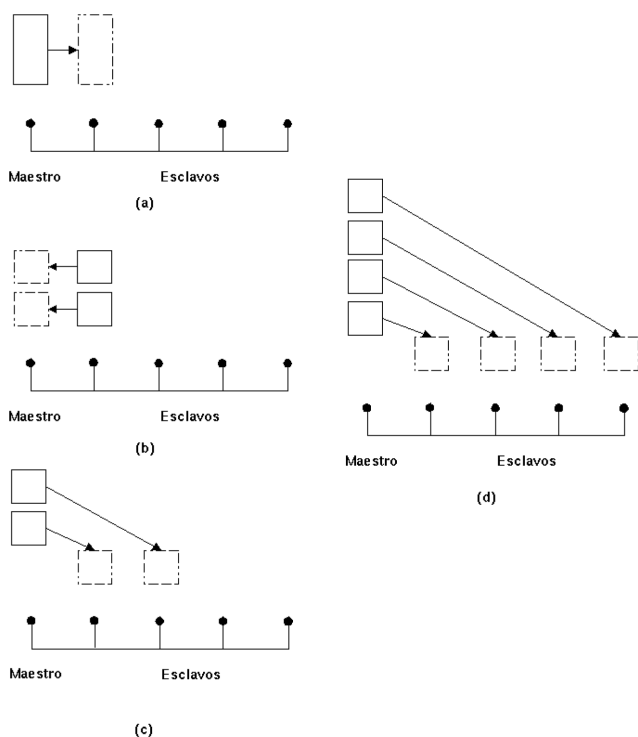


Figura 5. Esquema de la Fase de División.

En la Figura 5 (a) el maestro envía el problema al primer esclavo libre. Éste divide el problema, en este caso en dos, y le devuelve los dos nuevos subproblemas al maestro (b). El maestro vuelve a distribuir los problemas que le quedan sin resolver entre los procesadores que están ociosos (c). Finalmente todos los esclavos estarán trabajando en la división de algún subproblema (d).

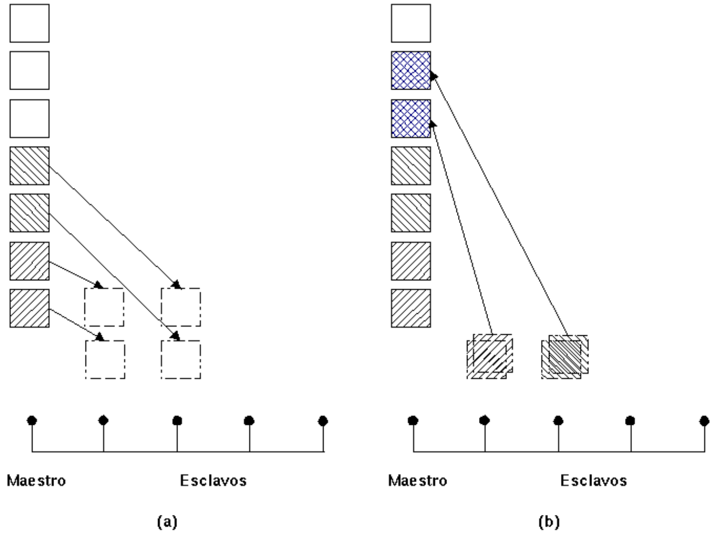


Figura 6. Esquema de la Fase de Combinación.

En la Figura 6 (a) el maestro envía las soluciones hermanas a los esclavos libres. Los esclavos combinan las soluciones, en este caso dos, y le devuelve la nueva solución al maestro (b) quien la coloca en el lugar correspondiente de la estructura de datos para repetir el esquema y enviar a un esclavo un nuevo conjunto de soluciones hermanas.

3.2. Esqueletos Ramificación y Acotación. El siguiente código muestra un esquema de Ramificación y Acotación para un problema de minimización:

```
procedure BandB (pbm, sp, sol)
begin
      SubProblem Queue[];
      Solution sl;
      Bound high, low, bestSol;

      bestSol := INFINITE;
      Queue.insert(sp);
      while not Queue.empty() do begin
            sp := Queue.extract();
            high := sp.upper_bound (pbm,sl);
            if high > bestSol then begin
                  low := sp.lower_bound(pbm, sl);
                  if low > bestSol then begin
                        bestsol := low;
                        sol := sl;
                  end;
                  if not sp.solve(pbm) then
                        sp.branch(pbm, HP);
                  end;
            end;
      return bestSol;
end;
```

La implementación paralela de BnB se realiza siguiendo el mismo esquema que el de la *Fase de División* de la técnica Divide y Vencerás.

4. Un caso de estudio: Problema de la Mochila 0/1

Se ha implementado utilizando los esqueletos Divide y Vencerás y Ramificación y Acotación el algoritmo que describen Martelo y Toth [11] para la resolución del problema de la Mochila 0/1 clásico. En dicho algoritmo, es necesario que los elementos estén ordenados de forma ascendente mediante la relación peso/beneficio. Haciendo uso de este requisito se ha implementado el método de ordenación Quicksort [6] con DnC para ordenar los elementos a insertar en la Mochila. A continuación mediante el uso de BnB se ha implementado el algoritmo de Ramificación y Acotación, lo que nos ha permitido utilizar de forma integrada ambos esqueletos.

4.1. Definición del problema. Se puede plantear el Problema de la Mochila 0/1 de la siguiente forma:

> "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 b_k y peso p_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 un problema de optimización es:

$$\max \sum_{k=1}^{N} p_k x_k \leq C$$

sujeto a:

$$x_k \in \{0,1\}, k \in \{1,...,N\}$$

4.2. Usando DnC. Para que el algoritmo propuesto funcione correctamente los elementos a insertar en la mochila se han de ordenar en función de la relación peso/beneficio: $b_1/p_1 \geq b_2/p_2 \geq ... \geq b_N/p_N$.

El *Quicksort* trabaja particionando el conjunto a ordenar en dos partes, para después ordenar dichas partes independientemente. El punto clave del algoritmo está en el procedimiento que divide el conjunto. El proceso de división del conjunto debe cumplir las siguientes tres condiciones:

(a) El elemento pivote=a[i] está en su posición final en el array para algún índice i.
(b) Todos los elementos en a[first], ..., a[i-1] son menores o iguales a a[i].
(c) Todos los elementos en a[i+1], ..., a[last] son mayores que a[i].

En este punto se aplica el mismo método recursivamente a los dos subproblemas generados: a[first], ... , a[i-1] y a[i+1], ... , a[last]. El resultado final será una matriz completamente ordenada, y por tanto no hace falta un paso subsiguiente de combinación.

La clase `Problem` estará formada por una variable miembro que representa el vector de enteros a ordenar. Dicha variable se ha declarado con el tipo `vector` de la librería STL de C++. Tanto la clase `SubProblem` como la clase `Solution` también tendrán una variable miembro del mismo tipo, que contiene en el primer caso un vector, de tamaño menor que el original, a ordenar, y en el segundo caso un vector de enteros ordenados que representan la solución parcial a un determinado subproblema.

`initSubProblem()`: El subproblema de partida tendrá el mismo contenido que el problema original. Por tanto, este método de la clase `SubProblem` simplemente copia los elementos del problema original en el subproblema.

```
SubProblem::initSubProblem (const Problem& pbm) {
      for (unsigned i = 0; i < pbm.l.size(); i++)
            l.push_back(pbm.1[i]);
}
```

`easy()`: Se considera que un subproblema es fácil para resolverlo por un algoritmo trivial cuando su tamaño es menor o igual que uno. Por tanto, la implementación del método queda como sigue:

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

`divide()`: La división del subproblema da lugar a dos nuevos subproblemas, donde el primero de ellos contiene los elementos menores que el pivote, mientras que el segundo contendrá los elementos mayores al pivote. A su vez, los elementos iguales al pivote se almacenarán en el objeto `Auxiliar` asociado a dicho subproblema. Se ha escogido como elemento pivote para la división el primer elemento del array de enteros a ordenar.

```
SubProblem::divide (const Problem& pbm,
      vector<SubProblem>& subpbms,
      Auxiliar& aux){
      SubProblem sp1; // Elms. menores al pivote.
      SubProblem sp2; // Elms. mayores al pivote.
      // pivote = primer elemento de la lista.
      const int& pivot = l[0];
      // Division del subproblem.
      for (unsigned i = 0; i < l.size(); i++) {
            if (l[i] < pivot) sp1.l.push_back(l[i]);
            if (l[i] == pivot) aux.l.push_back(l[i]);
            if (l[i] > pivot) sp2.l.push_back(l[i]);
      }
      subpbms.push_back(sp1);
      subpbms.push_back(sp2);
}
```

`combine()`: Para llevar a cabo la combinación, como las dos soluciones a combinar ya contienen los elementos ordenados correspondientes a dos subproblemas para los que los elementos del primero son menores que los elementos del segundo, simplemente hay que concatenar el contenido de la primera solución con el del auxiliar (elementos iguales), y finalmente con los elementos de la segunda solución.

```
Solution::combine (const Problem& pbm,
      const Auxiliar& aux),
      const vector<Solution>& subsols) {
            unsigned i;
            for (i = 0; i < subsols[0].l.size(); i++)
                  l.push_back(subsols[0].l[i]);
```

```
            for (i=0; i<aux.l.size(); i++)
                 l.push_back(aux.l[i]);
            for (i=0; i<subsols[1].l.size(); i++)
                 l.push_back(subsols[1].l[i]);
}
```

4.3. Implementación con BnB. La instanciación con MaLLBa del problema de la mochila queda de la siguiente forma:

`Problem`: se definen los datos del problema, el número de objetos, la capacidad de la mochila y el peso y beneficio de cada objeto.

```
requires class Problem {
      public:
            Number      C, // Capacidad
                        N; // Número de elementos
      vector<Number> p, //beneficios
                     w; //pesos
      Problem (); //Constructor
      inline Direction direction() const
      { return Maximize; }
            ...
}
```

`Solution`: se representa mediante un vector que contiene un verdadero o falso dependiendo de si el objeto pertenece o no a la solución.

```
requires class Solution {
      public:
            vector<bool> s;
                  ...
}
```

`SubProblem`: en cada subproblema se usarán los siguientes datos: la capacidad actual, el siguiente objeto a considerar, el beneficio actual y la solución actual.

```
requires class SubProblem {
      public:
            Number      CRest,      // capacidad actual
                        obj,        // siguiente objeto
                        profit;     // beneficio actual
            Solution sol;           // solucion actual
            SubProblem ();          // constructor
            ...
}
```

`initSubProblem()`: el primer subproblema considera toda la capacidad de la mochila; no se ha insertado ningún objeto, por lo tanto el beneficio obtenido es nulo.

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

`branch()`: partiendo de un subproblema, genera dos nuevos, considerando la inserción o no de otro elemento.

```
void SubProblem::branch (
            const Problem& pbm,
            container<SubProblem>& subpbms){
      SubProblem spNO; SubProblem spYES;

      spNO = SubProblem(CRest,obj+1,profit);
            spNO.sol.update(false, sol);
                  subpbms.insert(spNO);

      Number newC = CRest - pbm.w[obj];
      if (newC >= 0) {
            Number newPf = profit + pbm.p[obj];
            spYES = SubProblem(newC,obj+1,newPf);
            spYES.sol.update(true, sol);
            subpbms.insert(spYES);
      }
}
```

`upper_bound()`: calcula la cota superior incluyendo objetos hasta que la capacidad actual sea alcanzada, y para este último objeto sólo se incluye la proporción de capacidad que quepa.

```
Bound SubProblem::upper_bound (
            Const Problem& pbm){
      Bound upper, weigth, pft; Number i;

      for(i=obj,weigth=0,pft=profit;
                  weigth<=CRest; i++) {
            weigth += pbm.w[i];
            pft += pbm.p[i];
      }
      i--;
      weigth -= pbm.w[i]; pft -= pbm.p[i];
            upper = pft + (Number)((pbm.p[i]*
                  (CRest- eigth))/pbm.w[i]);
      return(upper);
}
```

`lower_bound()`: calcula la cota inferior incluyendo los objetos hasta que la capacidad actual sea alcanzada. El último objeto que haga que se supere la capacidad no será incluido.

```
Bound SubProblem::lower_bound (
            const Problem& pbm, Solution& us){
      Bound lower, weight, pft; Number i, tmp;
      us = sol;

      for(i = obj, weigth = 0, pft = profit;
                  weigth <= CRest; i++){
            weigth += pbm.w[i];
            pft += pbm.p[i];
            us.s.push_back(true);
            i--;
            weigth -= 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);
            lower = pft;
            return(lower);
      }
```

`solve()`: el problema está resuelto cuando no quedan objetos o cuando la mochila está llena.

```
bool SubProblem::solve(const Problem& pbm){
      return ((obj >= pbm.N) || (CRest <= 0));
}
```

5. Conclusiones

Se han presentado los esqueletos de la librería MaLLBa para la estrategia Divide y Vencerás (DnC) y para el paradigma de Ramificación y Acotación (BnB). La herramienta MaLLBa ofrece al usuario libertad para implementar las estructuras de datos que representan su problema, pero proporciona unos patrones de resolución que controlan el flujo de la ejecución.

El principal objetivo de MaLLBa es simplificar la tarea de los investigadores o usuarios que han de implementar un algoritmo usando una técnica algorítmica particular. MaLLBa proporciona al usuario un valor añadido no sólo en términos de la cantidad de código que ha de escribir, que es mucho menor, sino también en modularidad y claridad conceptual. Si se utiliza MaLLBa, o cualquier otro sistema orientado a objetos, el usuario ha de colocar cada pieza de código en la posición correcta.

Otra de las características de MaLLBa es la modularidad: una vez que las estructuras de datos básicas y las funcionalidades se han definido e insertado en el código, el esqueleto proporciona una implementación gratuita en paralelo.

MaLLBa hace un uso equilibrado de la programación orientada a objetos, en el sentido de que proporciona las mínimas plantillas necesarias para conseguir una eficiencia computacional buena y al mismo tiempo hacer el proceso de compilación seguro.

Además el sistema pone a disposición del usuario la posibilidad de generar y experimentar con nuevos algoritmos que permitan la integración de técnicas. En este trabajo se ha mostrado la posibilidad de esta integración mediante el ejemplo del Problema de la Mochila.

Reconocimientos

Han colaborado en la realización del proyecto MaLLBa::DnC y MaLLBa::BnB Isabel Dorta González, Casiano Rodríguez León y Angélica Rojas Rodríguez.

Bibliografía

[1] Alba, E., Almeida, F., Blesa, M., Cabeza, J., Cotta, C., Díaz, M., Dorta, I., Gabarró, J., León, C., Luna, J., Moreno, L., Petit, J., Rojas, A., Xhafa, F.: MaLLBa: A Library of skeletons for combinatorial optimization. En *Proceedings of the International Euro-Par Conference (Paderborn, Germany).* LNCS **2400** (2002), 927-932.

[2] Alba, E., Almeida, F., Blesa, M., Cotta, C., Díaz, M., Dorta, I., Gabarró, J., González, J., León, C., Moreno, L., Petit, J., Roda, J., Rojas, A., Xhafa, F.: MaLLBa*: Towards a Combinatorial Optimization Library for Geographically Distributed Systems.* Actas de las XII Jornadas de Paralelismo (2001), 105-110.

[3] Dorta, I., Rojas, A., León, C., Dorta P.: *Utilización de software en la docencia de técnicas algorítmicas.* Actas de las VII Jornadas de Enseñanza Universitaria de la Informática, 2001 .

[4] Eckstein, J., Phillips, C.A., Hart, W.E.: *PICO: An Object-Oriented Framework for Parallel Branch and Bound.* Rutcor Research Report (2000).

[5] High Performance Fortran Forum: *High Performance Fortran Specification,* 1993.

[6] Hoare, C.A.R., Algorithm 64: Quicksort. *Communications of the ACM*, **4**:321 (1961).

[7] Kielmann, T., Nieuwpoort, R., Bal, H.: *Cilk-5.3 Reference Manual.* Supercomputing Technologies Group, 2000.

[8] Kielmann, T., Nieuwpoort, R., Bal, H.: Satin: *Efficient Parallel Divide-and-Conquer in Java.* Euro-Par 2000 (2000), 690-699.

[9] León, C., Rojas A.: *Un Esqueleto Paralelo para el Paradigma Divide y Vencerás.* Documento de Trabajo Interno. DEIOC-01-01, 2001.

[10] B. Le Cun, C. Roucairol, The PNN Team: *BOB: A unified platform for implementing branch-and-bound like algorithms.* Rapport de Recherche 95/16 (1999).

[11] Martello, S., Toth, P.: *Knapsack Problems: Algorithms and Computer Implementations.* John Wiley & Sons Ltd., 1990.

[12] OpenMP Architecture Review Board: *OpenMP C and* C++ *Application Program Interface, Version 1.0* (1998). [Disponible en http://www.openmp.org].

[13] Shinano, Y., Higaki, M., Hirabayashi, R.: *A Generalized Utility for Parallel Branch and Bound Algorithms*. IEEE Computer Society Press (1995), 392-401.

[14] Tschöke, S., Polzer, T.: *Portable Parallel Branch-and-Bound Library.* User Manual Library Version 2.0, Paderborn, 1995.

En Internet

http://cvs.deioc.ull.es

PCGULL

Parallel Computing Group at the University of La Laguna.

http://www.cs.sandia.gov/~caphill/proj/pico.html

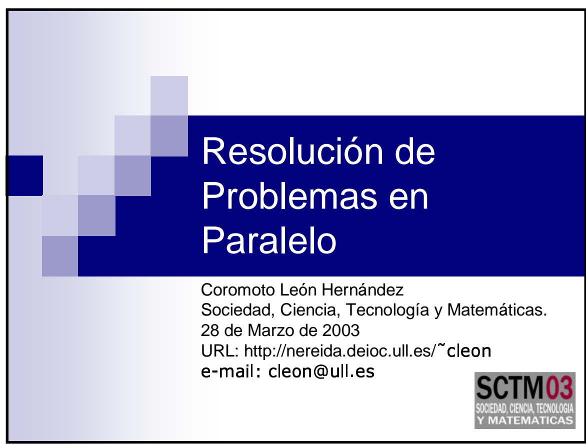
PICO

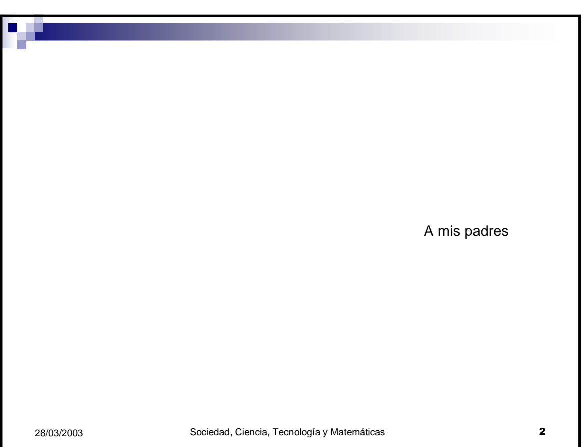
Parallel Integer and Combinatorial Optimization.

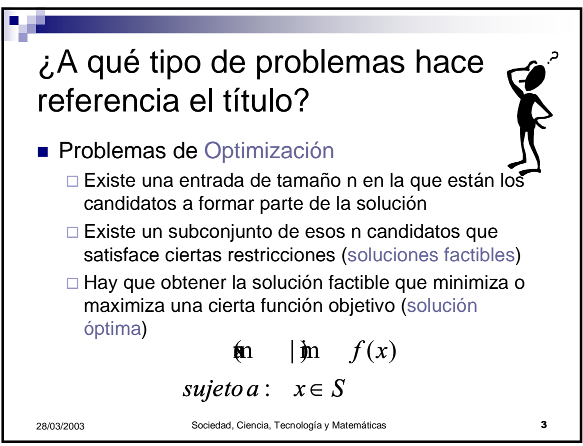
http://www.prism.uvsq.fr/~blec/Research/BOBO

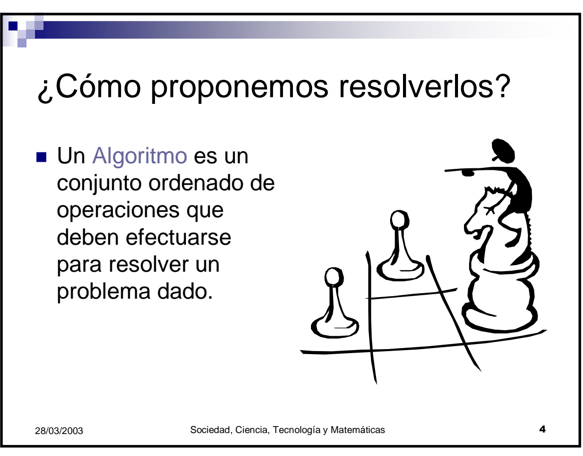
Bob++

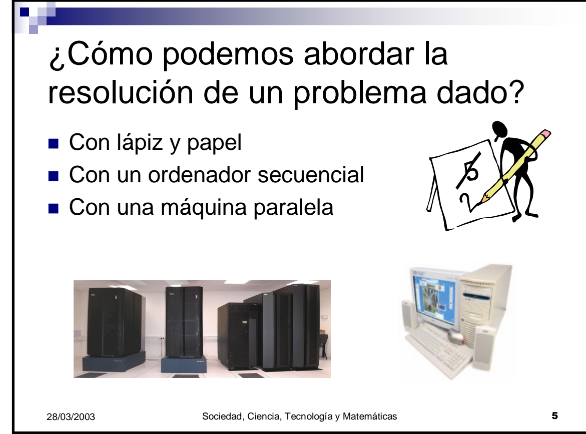
Parallel and sequential parallel branch-and-bound.

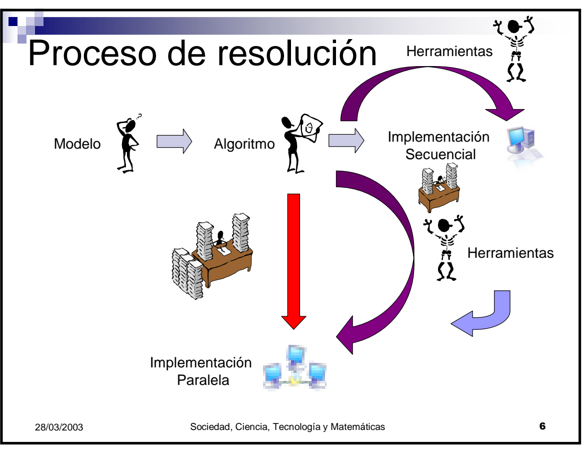

Resolución de Problemas en Paralelo
Coromoto León Hernández
Sociedad, Ciencia, Tecnología y Matemáticas.
28 de Marzo de 2003
URL: http://nereida.deioc.ull.es/~cleon
e-mail: cleon@ull.es
SCTM03
SOCIEDAD, CIENCIA, TECNOLOGÍA Y MATEMÁTICAS

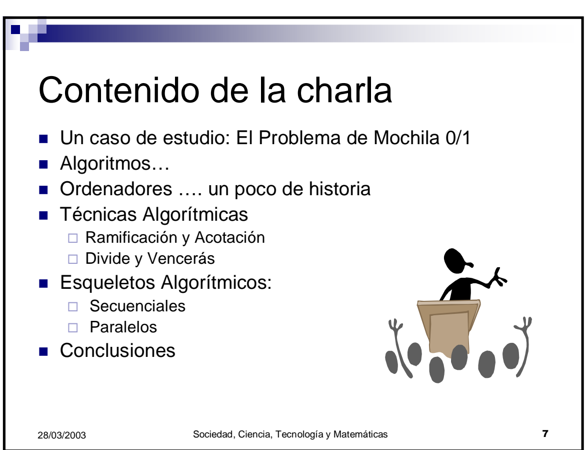

A mis padres
28/03/2003
Sociedad, Ciencia, Tecnología y Matemáticas
2


¿A qué tipo de problemas hace referencia el título?
Problemas de Optimización
Existe una entrada de tamaño n en la que están los candidatos a formar parte de la solución
Existe un subconjunto de esos n candidatos que satisface ciertas restricciones (soluciones factibles)
Hay que obtener la solución factible que minimiza o maximiza una cierta función objetivo (solución óptima)
f(x)
sujeto a : x ∈ S
28/03/2003
Sociedad, Ciencia, Tecnología y Matemáticas
3

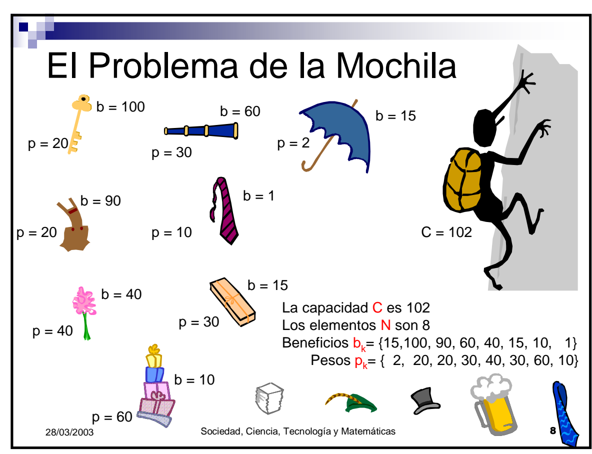

¿Cómo proponemos resolverlos?
Un Algoritmo es un conjunto ordenado de operaciones que deben efectuarse para resolver un problema dado.
28/03/2003
Sociedad, Ciencia, Tecnología y Matemáticas
4


¿Cómo podemos abordar la resolución de un problema dado?
Con lápiz y papel
Con un ordenador secuencial
Con una máquina paralela
28/03/2003
Sociedad, Ciencia, Tecnología y Matemáticas
5

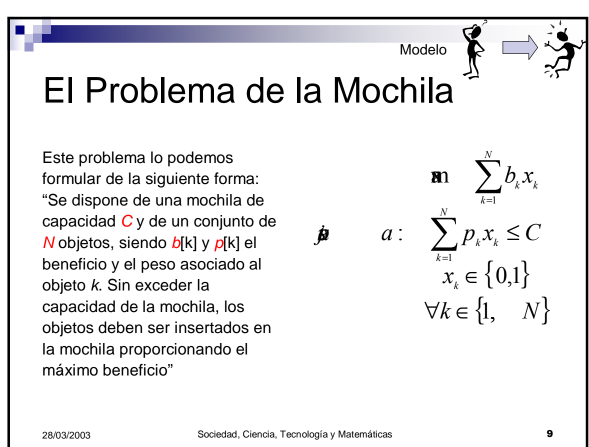

Proceso de resolución
Herramientas
Modelo
Algoritmo
Implementación Secuencial
Herramientas
Implementación Paralela
28/03/2003
Sociedad, Ciencia, Tecnología y Matemáticas
6



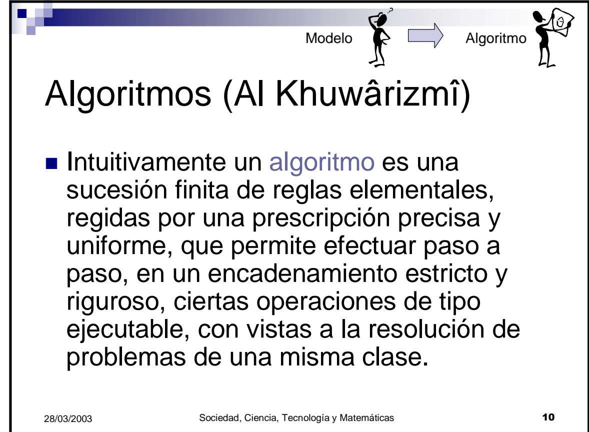
Contenido de la charla
Un caso de estudio: El Problema de Mochila 0/1
Algoritmos…
Ordenadores …. un poco de historia
Técnicas Algorítmicas
Ramificación y Acotación
Divide y Vencerás
Esqueletos Algorítmicos:
Secuenciales
Paralelos
Conclusiones
28/03/2003
Sociedad, Ciencia, Tecnología y Matemáticas
7



El Problema de la Mochila
b = 100
p = 20
b = 60
p = 30
p = 2
b = 15
b = 90
p = 20
p = 10
b = 1
C = 102
b = 40
p = 40
p = 30
b = 15
La capacidad C es 102
Los elementos N son 8
Beneficios b_k = {15,100, 90, 60, 40, 15, 10, 1}
Pesos p_k = { 2, 20, 20, 30, 40, 30, 60, 10}
b = 10
p = 60
28/03/2003
Sociedad, Ciencia, Tecnología y Matemáticas
8

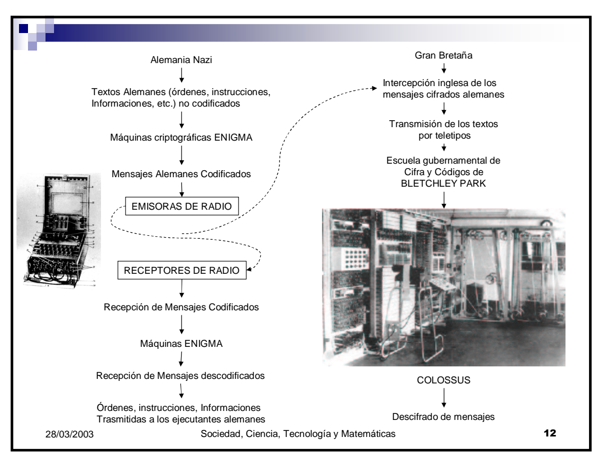


Modelo
El Problema de la Mochila
Este problema lo podemos formular de la siguiente forma: "Se dispone de una mochila de capacidad C y de un conjunto de N objetos, siendo b[k] y p[k] el beneficio y el peso asociado al objeto k. Sin exceder la capacidad de la mochila, los objetos deben ser insertados en la mochila proporcionando el máximo beneficio"
$\max \sum_{k=1}^{N} b_k x_k$
$a: \sum_{k=1}^{N} p_k x_k \le C$
$x_k \in \{0,1\}$
$\forall k \in \{1, N\}$
28/03/2003
Sociedad, Ciencia, Tecnología y Matemáticas
9

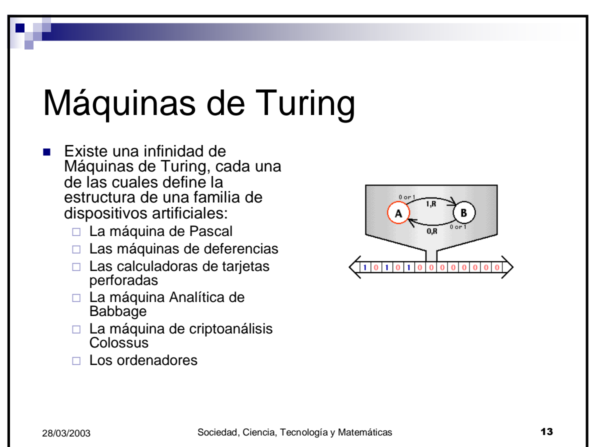


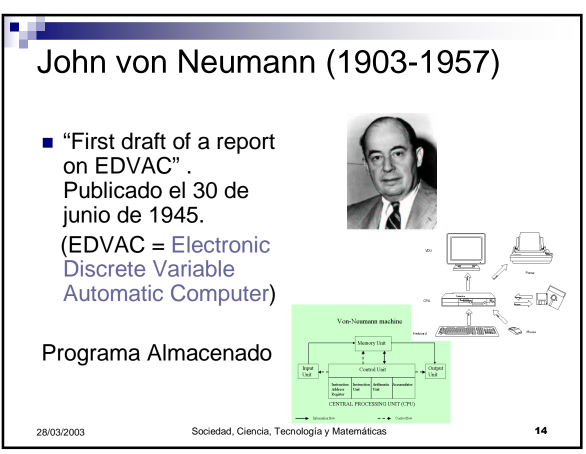
Modelo
Algoritmo
Algoritmos (Al Khuwârizmî)
Intuitivamente un algoritmo es una sucesión finita de reglas elementales, regidas por una prescripción precisa y uniforme, que permite efectuar paso a paso, en un encadenamiento estricto y riguroso, ciertas operaciones de tipo ejecutable, con vistas a la resolución de problemas de una misma clase.
28/03/2003
Sociedad, Ciencia, Tecnología y Matemáticas
10

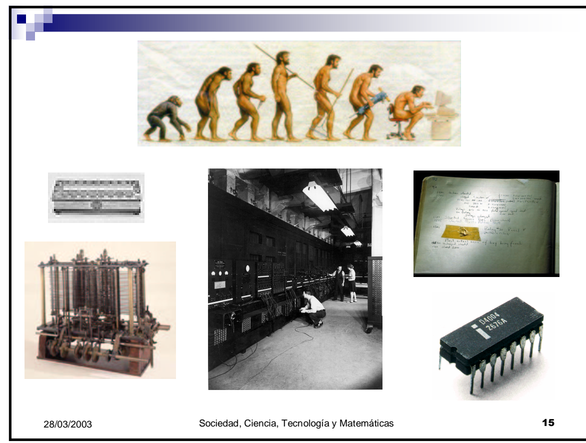
Alan Mathison Turing (1912-1954)
"Sobre los números calculables y su aplicación al problema de la decidibilidad" (1936)
Define rigurosamente un Algoritmo
Introduce los conceptos de
Máquina de Turing y
Autómata Algorítmico Universal
28/03/2003
Sociedad, Ciencia, Tecnología y Matemáticas
11

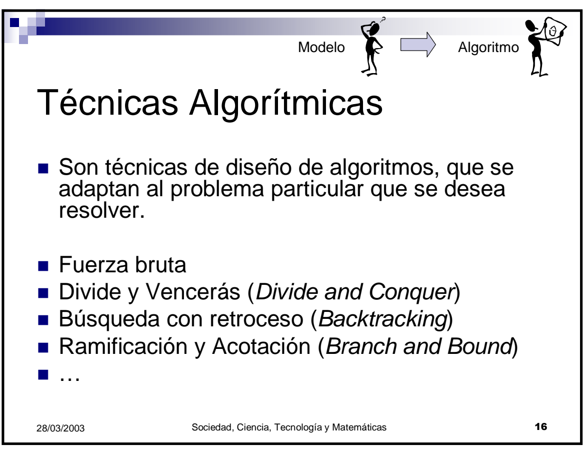


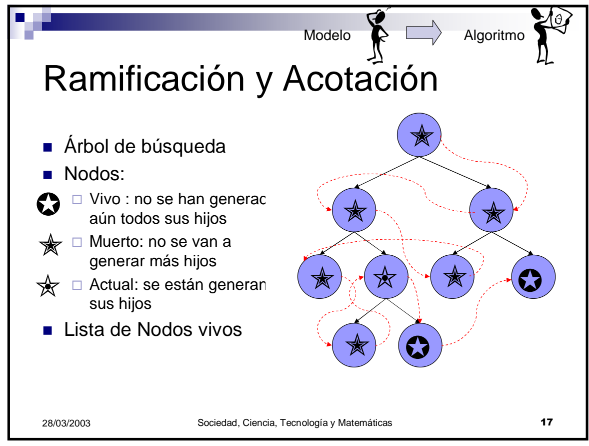
Alemania Nazi
Textos Alemanes (órdenes, instrucciones, Informaciones, etc.) no codificados
Máquinas criptográficas ENIGMA
Mensajes Alemanes Codificados
EMISORAS DE RADIO
RECEPTORES DE RADIO
Recepción de Mensajes Codificados
Máquinas ENIGMA
Recepción de Mensajes descodificados
Órdenes, instrucciones, Informaciones Trasmitidas a los ejecutantes alemanes
Gran Bretaña
Intercepción inglesa de los mensajes cifrados alemanes
Transmisión de los textos por teletipos
Escuela gubernamental de Cifra y Códigos de BLETCHLEY PARK
COLOSSUS
Descifrado de mensajes
28/03/2003
Sociedad, Ciencia, Tecnología y Matemáticas
12

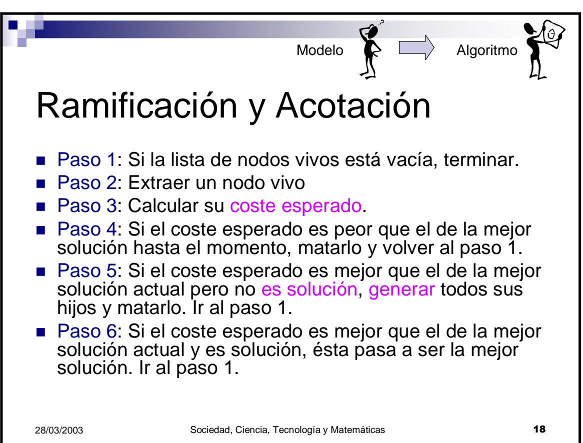

Máquinas de Turing
Existe una infinidad de Máquinas de Turing, cada una de las cuales define la estructura de una familia de dispositivos artificiales:
La máquina de Pascal
Las máquinas de deferencias
Las calculadoras de tarjetas perforadas
La máquina Analítica de Babbage
La máquina de criptoanálisis Colossus
Los ordenadores
28/03/2003
Sociedad, Ciencia, Tecnología y Matemáticas
13

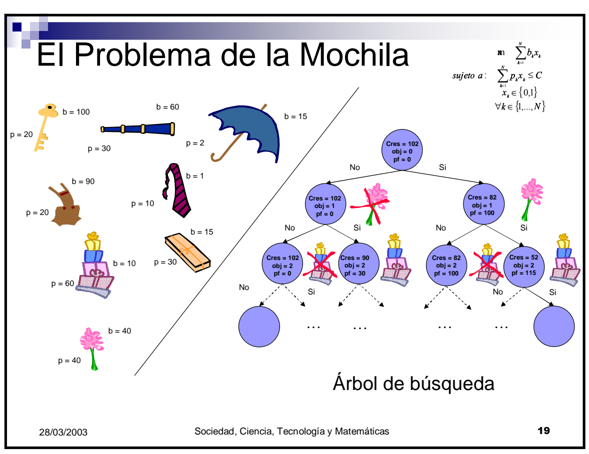

John von Neumann (1903-1957)
"First draft of a report on EDVAC" . Publicado el 30 de junio de 1945.
(EDVAC = Electronic Discrete Variable Automatic Computer)
Programa Almacenado
Von-Neumann machine
28/03/2003
Sociedad, Ciencia, Tecnología y Matemáticas
14

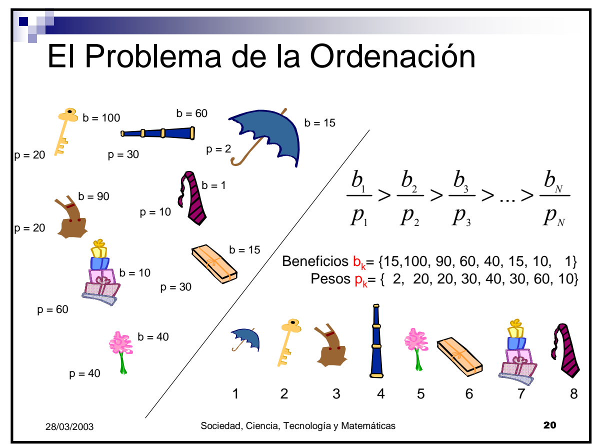

28/03/2003
Sociedad, Ciencia, Tecnología y Matemáticas
15

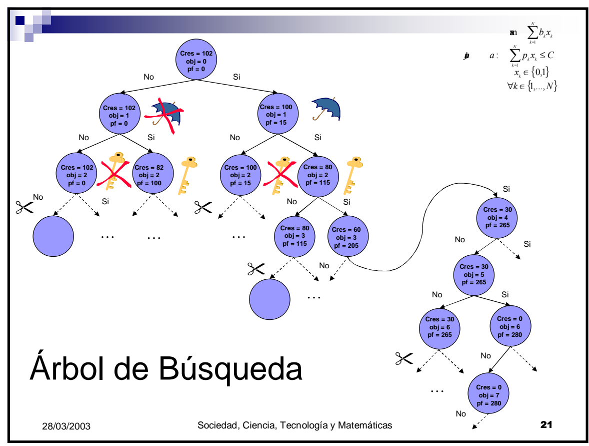

Modelo
Algoritmo
Técnicas Algorítmicas
Son técnicas de diseño de algoritmos, que se adaptan al problema particular que se desea resolver.
Fuerza bruta
Divide y Vencerás (Divide and Conquer)
Búsqueda con retroceso (Backtracking)
Ramificación y Acotación (Branch and Bound)
…
28/03/2003
Sociedad, Ciencia, Tecnología y Matemáticas
16

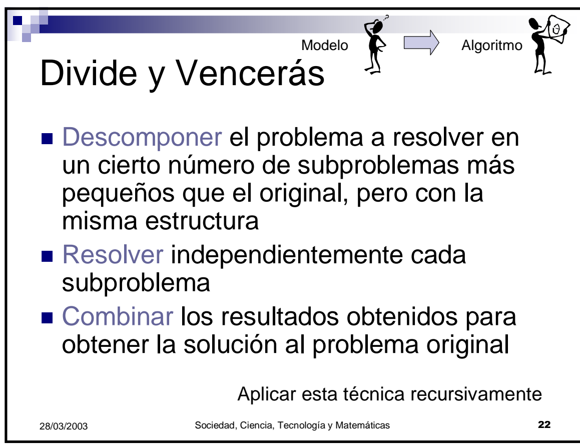

Modelo
Algoritmo
Ramificación y Acotación
Árbol de búsqueda
Nodos:
Vivo : no se han generac aún todos sus hijos
Muerto: no se van a generar más hijos
Actual: se están generan sus hijos
Lista de Nodos vivos
28/03/2003
Sociedad, Ciencia, Tecnología y Matemáticas
17

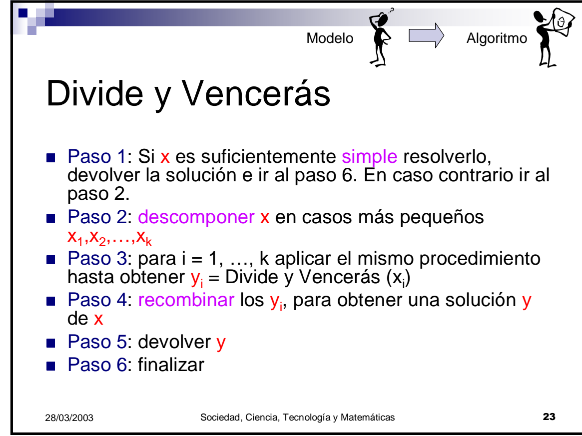

Modelo
Algoritmo
Ramificación y Acotación
Paso 1: Si la lista de nodos vivos está vacía, terminar.
Paso 2: Extraer un nodo vivo
Paso 3: Calcular su coste esperado.
Paso 4: Si el coste esperado es peor que el de la mejor solución hasta el momento, matarlo y volver al paso 1.
Paso 5: Si el coste esperado es mejor que el de la mejor solución actual pero no es solución, generar todos sus hijos y matarlo. Ir al paso 1.
Paso 6: Si el coste esperado es mejor que el de la mejor solución actual y es solución, ésta pasa a ser la mejor solución. Ir al paso 1.
28/03/2003
Sociedad, Ciencia, Tecnología y Matemáticas
18

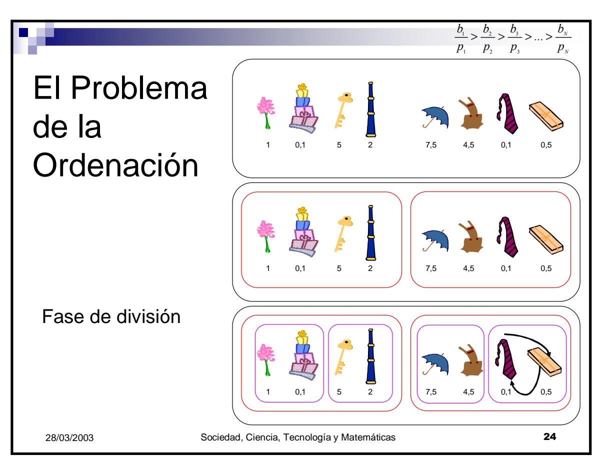

El Problema de la Mochila
sujeto a :
Árbol de búsqueda
28/03/2003
Sociedad, Ciencia, Tecnología y Matemáticas
19

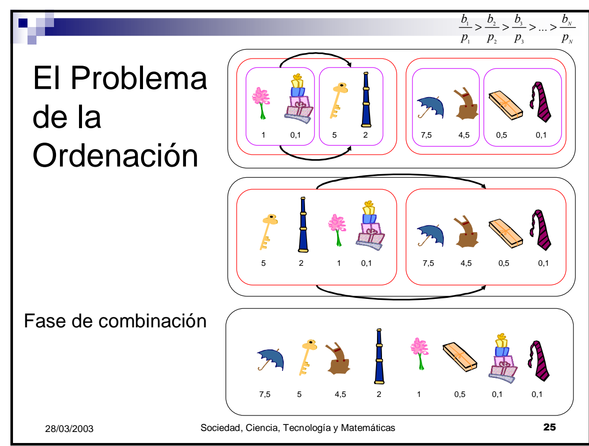

El Problema de la Ordenación
Beneficios b_k = {15,100, 90, 60, 40, 15, 10, 1}
Pesos p_k = { 2, 20, 20, 30, 40, 30, 60, 10}
1 2 3 4 5 6 7 8
28/03/2003
Sociedad, Ciencia, Tecnología y Matemáticas
20

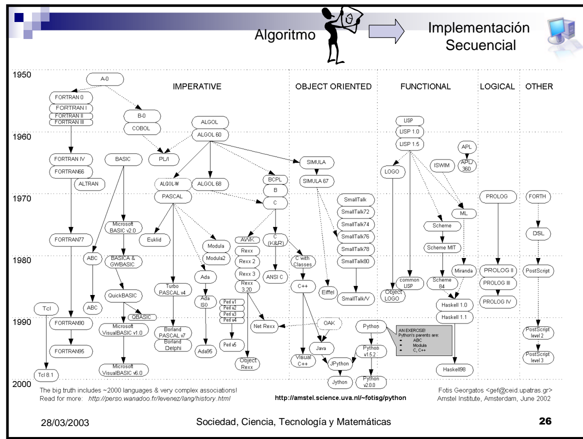

Árbol de Búsqueda
28/03/2003
Sociedad, Ciencia, Tecnología y Matemáticas
21

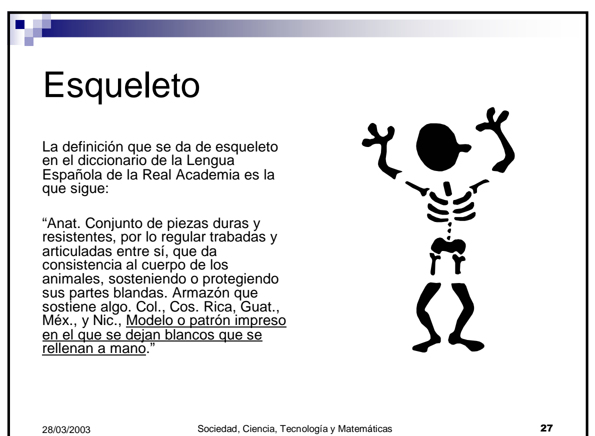

Modelo
Algoritmo
Divide y Vencerás
Descomponer el problema a resolver en un cierto número de subproblemas más pequeños que el original, pero con la misma estructura
Resolver independientemente cada subproblema
Combinar los resultados obtenidos para obtener la solución al problema original
Aplicar esta técnica recursivamente
28/03/2003
Sociedad, Ciencia, Tecnología y Matemáticas
22

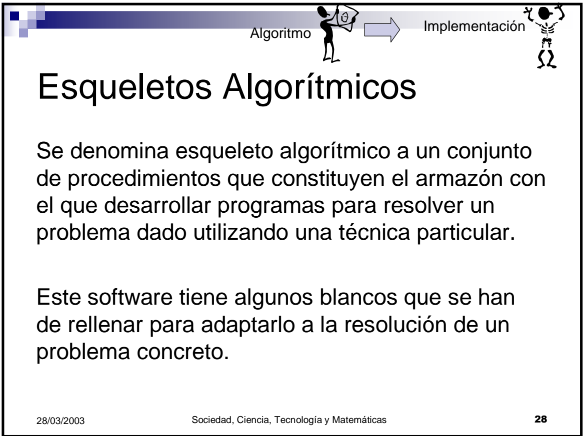

Modelo
Algoritmo
Divide y Vencerás
Paso 1: Si x es suficientemente simple resolverlo, devolver la solución e ir al paso 6. En caso contrario ir al paso 2.
Paso 2: descomponer x en casos más pequeños x1,x2,…,xk
Paso 3: para i = 1, …, k aplicar el mismo procedimiento hasta obtener yi = Divide y Vencerás (xi)
Paso 4: recombinar los yi, para obtener una solución y de x
Paso 5: devolver y
Paso 6: finalizar
28/03/2003
Sociedad, Ciencia, Tecnología y Matemáticas
23

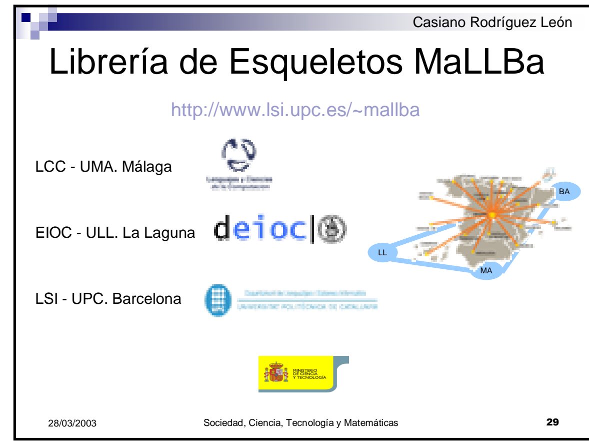

El Problema de la Ordenación
Fase de división
1 0,1 5 2 7,5 4,5 0,1 0,5
28/03/2003
Sociedad, Ciencia, Tecnología y Matemáticas
24

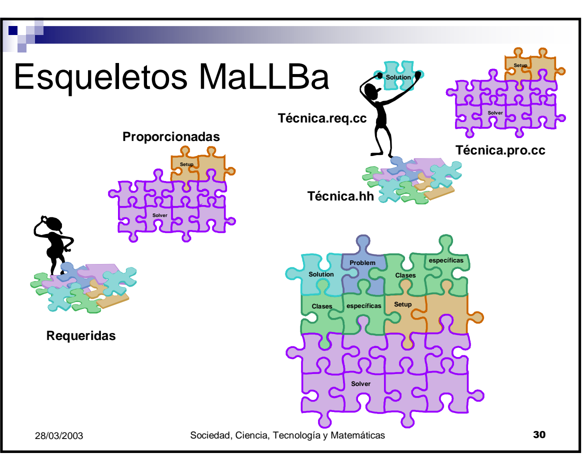

El Problema de la Ordenación
Fase de combinación
28/03/2003
Sociedad, Ciencia, Tecnología y Matemáticas
25


Algoritmo
Implementación Secuencial
IMPERATIVE
OBJECT ORIENTED
FUNCTIONAL
LOGICAL
OTHER
1950
1960
1970
1980
1990
2000
28/03/2003
Sociedad, Ciencia, Tecnología y Matemáticas
26


Esqueleto
La definición que se da de esqueleto en el diccionario de la Lengua Española de la Real Academia es la que sigue:
"Anat. Conjunto de piezas duras y resistentes, por lo regular trabadas y articuladas entre sí, que da consistencia al cuerpo de los animales, sosteniendo o protegiendo sus partes blandas. Armazón que sostiene algo. Col., Cos. Rica, Guat., Méx., y Nic., Modelo o patrón impreso en el que se dejan blancos que se rellenan a mano."
28/03/2003
Sociedad, Ciencia, Tecnología y Matemáticas
27

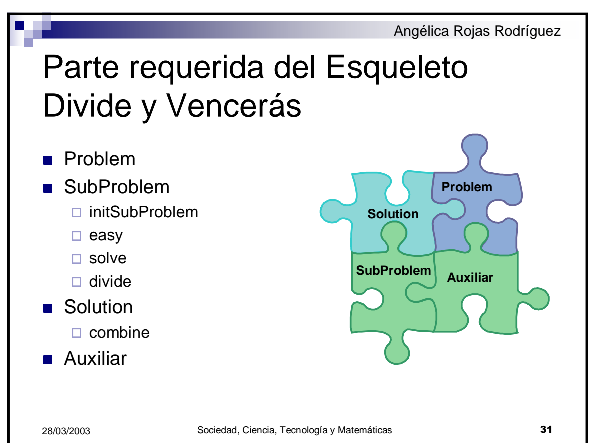

Algoritmo
Implementación
Esqueletos Algorítmicos
Se denomina esqueleto algorítmico a un conjunto de procedimientos que constituyen el armazón con el que desarrollar programas para resolver un problema dado utilizando una técnica particular.
Este software tiene algunos blancos que se han de rellenar para adaptarlo a la resolución de un problema concreto.
28/03/2003
Sociedad, Ciencia, Tecnología y Matemáticas
28


Casiano Rodríguez León
Librería de Esqueletos MaLLBa
http://www.lsi.upc.es/~mallba
LCC - UMA. Málaga
EIOC - ULL. La Laguna
deioc
LSI - UPC. Barcelona
BA
LL
MA
28/03/2003
Sociedad, Ciencia, Tecnología y Matemáticas
29


Esqueletos MaLLBa
Técnica.req.cc
Proporcionadas
Técnica.pro.cc
Técnica.hh
Requeridas
Solver
Setup
Solution
Problem
Clases
especificas
28/03/2003
Sociedad, Ciencia, Tecnología y Matemáticas
30

Parte requerida del Esqueleto Divide y Vencerás

- Problem
- SubProblem
  - initSubProblem
  - easy
  - solve
  - divide
- Solution
  - combine
- Auxiliar



$\frac{b_1}{p_1} > \frac{b_2}{p_2} > \frac{b_3}{p_3} > \ldots > \frac{b_N}{p_N}$

El problema de Ordenación

```
void SubProblem::initSubProblem (const Problem& pbm) {
  for (unsigned i = 0; i < pbm.l.size(); i++)
    l.push_back(pbm.l[i]);
}

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

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


$\frac{b_1}{p_1} > \frac{b_2}{p_2} > \frac{b_3}{p_3} > \ldots > \frac{b_N}{p_N}$

El problema de Ordenación

```
void SubProblem::divide (const Problem& pbm,
                           vector<SubProblem>& subpbms,
                           Auxiliar& aux){
  SubProblem sp1, sp2;
  unsigned i, middle = l.size() / 2;

  for (i = 0; i < middle; i++)
    sp1.l.push_back(l[i]);
  for (i = middle; i < l.size(); i++)
    sp2.l.push_back(l[i]);
  subpbms.push_back(sp1);
  subpbms.push_back(sp2);
}
```

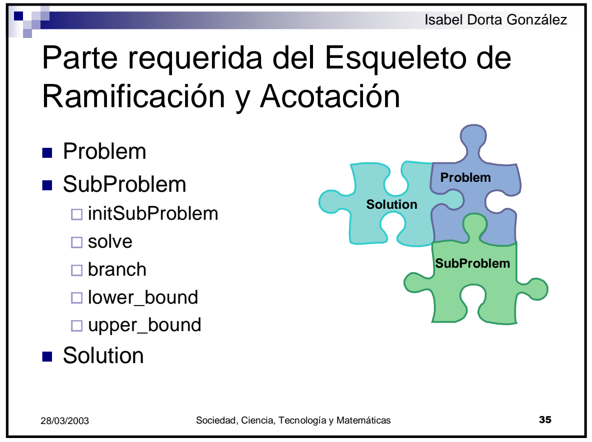

$\frac{b_1}{p_1} > \frac{b_2}{p_2} > \frac{b_3}{p_3} > \ldots > \frac{b_N}{p_N}$

El problema de Ordenación

```
void Solution::combine (const Problem& pbm,
                        const Auxiliar& aux,
                        const vector<Solution>& subsols) {
  vector<int>::const_iterator i = subsols[0].l.begin();
  vector<int>::const_iterator j = subsols[1].l.begin();
  while ((i!=subsols[0].l.end())&&(j!=subsols[1].l.end())) {
    if (*i < *j) { l.push_back(*i); i++; }
    else { l.push_back(*j); j++; }
  }
  while (i != subsols[0].l.end()) { l.push_back(*i); i++;}
  while (j != subsols[1].l.end()) { l.push_back(*j); j++;}
}
```


Parte requerida del Esqueleto de Ramificación y Acotación

- Problem
- SubProblem
  - initSubProblem
  - solve
  - branch
  - lower_bound
  - upper_bound
- Solution



$\max \sum_{k=1}^{N} b_k x_k$

$s.a: \sum_{k=1}^{N} p_k x_k \leq C$

$x_k \in \{0,1\}$

$\forall k \in \{1,\ldots,N\}$

El problema de la Mochila

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

Bound SubProblem::upper_bound(const Problem& pbm, Solution& sl){
   Number weigth, pft, i;
   for(i = obj, weigth = 0, pft = profit;
       weigth <= CRest; i++){
     weigth += pbm.w[i];
     pft += pbm.p[i];
   }
   if (i != obj){
     i--; weigth -= pbm.w[i]; pft -= pbm.p[i];
   }
   if ((i == obj) && (obj == (pbm.N - 1))) return (Bound)pft;
   return (Bound)(pft);
 }
```

El problema de la Mochila

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


El problema de la Mochila

```
void SubProblem::branch (const Problem& pbm,
    branchQueue<SubProblem>& subpbms){
   SubProblem spNO, spYES;

   Number nextObject = obj + 1;
   if (nextObject <= pbm.N ) {
     spNO = SubProblem(CRest, nextObject, profit);
     spNO.sol.update(false, sol);
     subpbms.insert(spNO);
     Number newC = CRest - pbm.w[obj];
     if (newC >= 0) {
       Number newPf = profit + pbm.p[obj];
       spYES = SubProblem( newC, nextObject, newPf);
       spYES.sol.update(true, sol);
       subpbms.insert(spYES);
     }
   }
 }
```


Parte Proporcionada

- SetUp
  - Por ejemplo, el tipo de recorrido del espacio de búsqueda.
- Solver
  - Implementaciones secuenciales
  - Implementaciones paralelas

Setup

Solver

Esqueleto Divide y Vencerás Secuencial

M Memoria

P Procesador

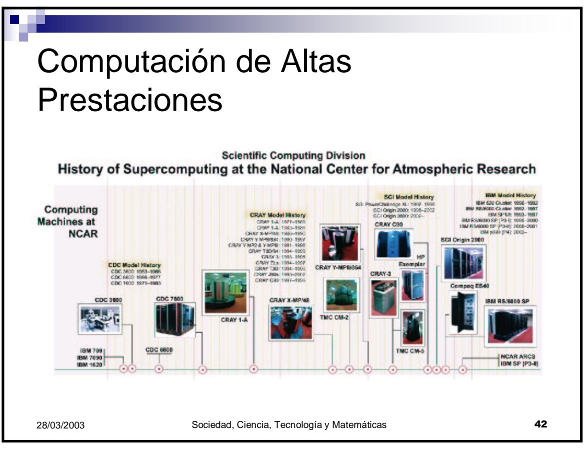
```
procedure DandC(sp, sol) {
  if (sp.easy()) {
    sol = sp.solve();
  }
  else {
    sp.divide(<array>subpbm);
    <array>subsol;
    for i := 1 to subpbm. numProblem() do
      DandC(subpbm[i], subsol[i]);
    }
    sp.combine(subsol[i], sol);
  }
}
```

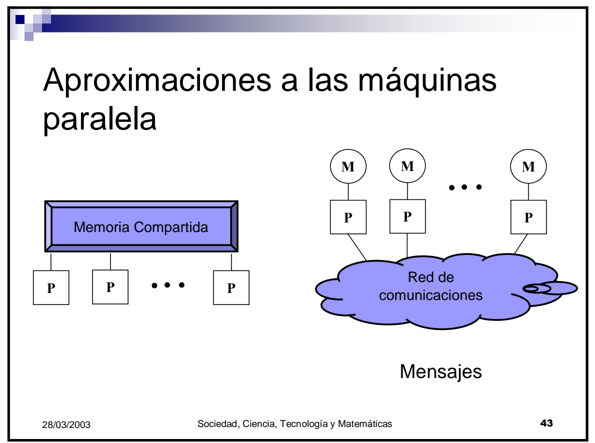

Esqueleto de Ramificación y Acotación Secuencial

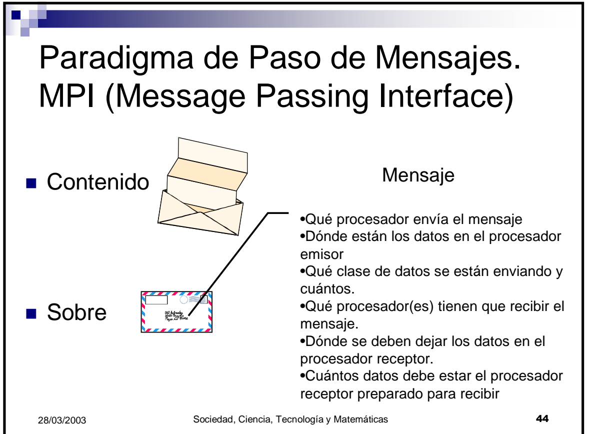
```
Bound BB (const Problem& pbm, const SubProblem& sp, Solution& sol){
  branchQueue<SubProblem> bqueue;
  Solution sl;
  Bound high, low;
  SubProblem sp;
  bqueue.insert(sp);
  while (!bqueue.empty()) {
    sp = bqueue.remove();
    high = sp.upper_bound (pbm, sl);
    if (high > bestSol){
      low = sp.lower_bound(pbm, sl);
      if (low > bestSol) {
        bestSol = low;
        sol = sl;
      }
      if (!sp.solve(pbm)) sp.branch(pbm, bqueue);
    }
  }
  return(bestSol);
}
```

Maximización

Computación de Altas Prestaciones

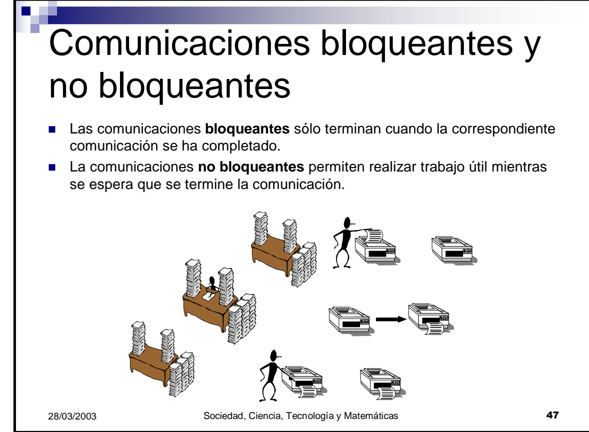



Aproximaciones a las máquinas paralela
Memoria Compartida
P
P
• • •
P
M
M
• • •
M
P
P
P
Red de comunicaciones
Mensajes
28/03/2003
Sociedad, Ciencia, Tecnología y Matemáticas
43


Paradigma de Paso de Mensajes. MPI (Message Passing Interface)
Contenido
Mensaje
Sobre
•Qué procesador envía el mensaje
•Dónde están los datos en el procesador emisor
•Qué clase de datos se están enviando y cuántos.
•Qué procesador(es) tienen que recibir el mensaje.
•Dónde se deben dejar los datos en el procesador receptor.
•Cuántos datos debe estar el procesador receptor preparado para recibir
28/03/2003
Sociedad, Ciencia, Tecnología y Matemáticas
44

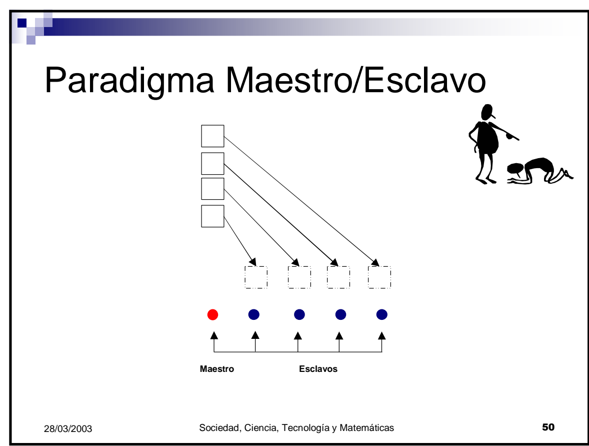
Comunicaciones Síncronas
Una comunicación síncrona no se completa hasta que el mensaje ha sido recibido
beep
28/03/2003
Sociedad, Ciencia, Tecnología y Matemáticas
45

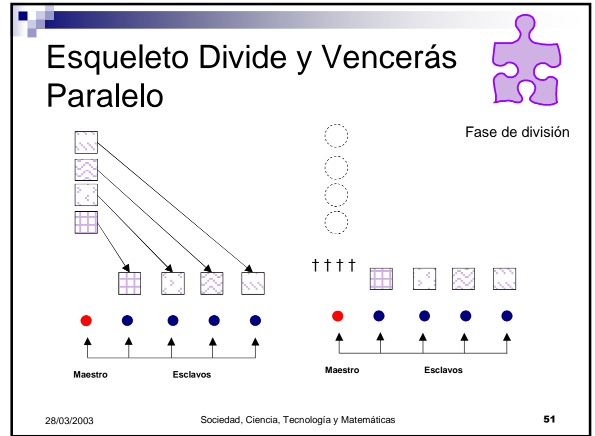
Comunicaciones Asíncronas
Una comunicación asíncrona se completa tan pronto como el mensaje se ha puesto en el medio de transporte.
28/03/2003
Sociedad, Ciencia, Tecnología y Matemáticas
46

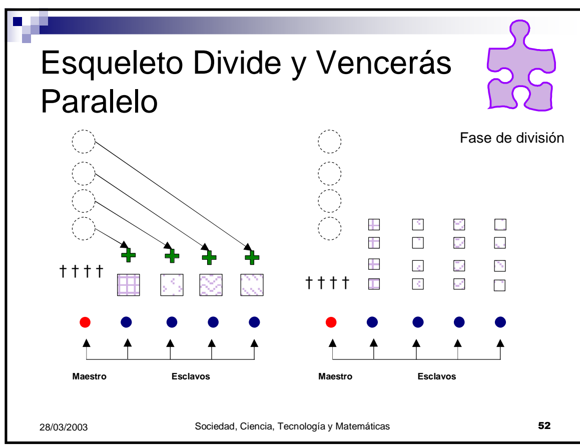

Comunicaciones bloqueantes y no bloqueantes
Las comunicaciones bloqueantes sólo terminan cuando la correspondiente comunicación se ha completado.
La comunicaciones no bloqueantes permiten realizar trabajo útil mientras se espera que se termine la comunicación.
28/03/2003
Sociedad, Ciencia, Tecnología y Matemáticas
47

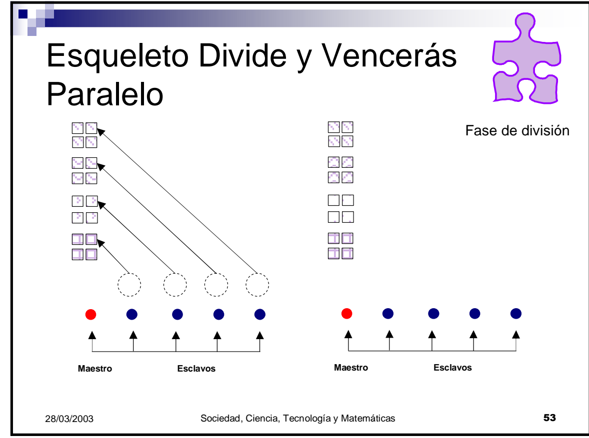
Comunicaciones colectivas (1)
Una barrera es una operación que permite sincronizar procesos. No hay intercambio de datos pero la barrera impide el paso hasta que todos los procesadores participantes lleguen ahí.
BARRIER
BARRIER
BARRIER
28/03/2003
Sociedad, Ciencia, Tecnología y Matemáticas
48

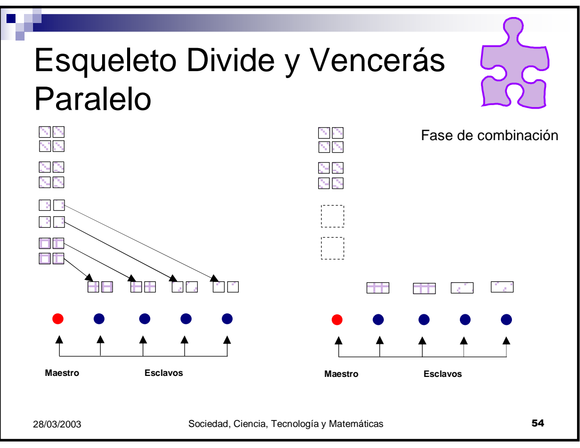
Comunicaciones colectivas (2)
La **emisión** (broadcast) es una comunicación uno-a-muchos. Con una sola operación un procesador envía el mismo mensaje a varios.
28/03/2003
Sociedad, Ciencia, Tecnología y Matemáticas
49

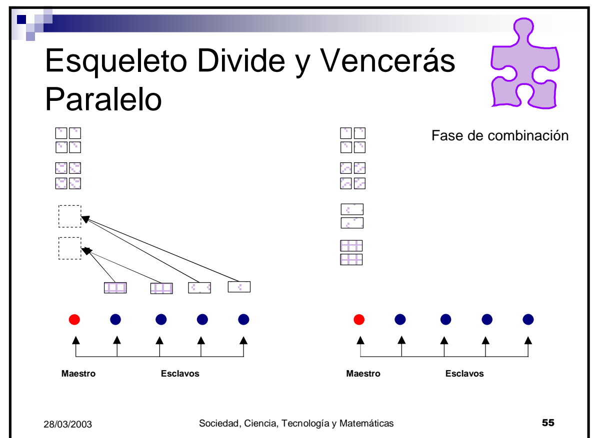

Paradigma Maestro/Esclavo
Maestro
Esclavos
28/03/2003
Sociedad, Ciencia, Tecnología y Matemáticas
50

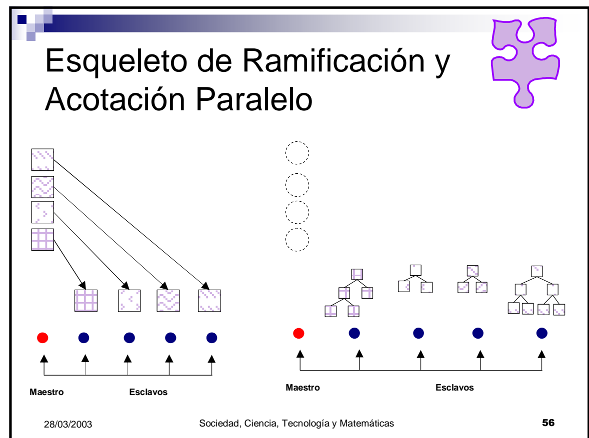

Esqueleto Divide y Vencerás Paralelo
Fase de división
Maestro
Esclavos
Maestro
Esclavos
28/03/2003
Sociedad, Ciencia, Tecnología y Matemáticas
51

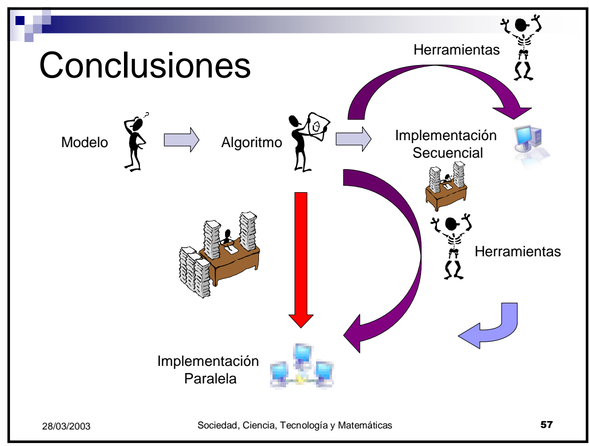

Esqueleto Divide y Vencerás Paralelo
Fase de división
Maestro
Esclavos
Maestro
Esclavos
28/03/2003
Sociedad, Ciencia, Tecnología y Matemáticas
52

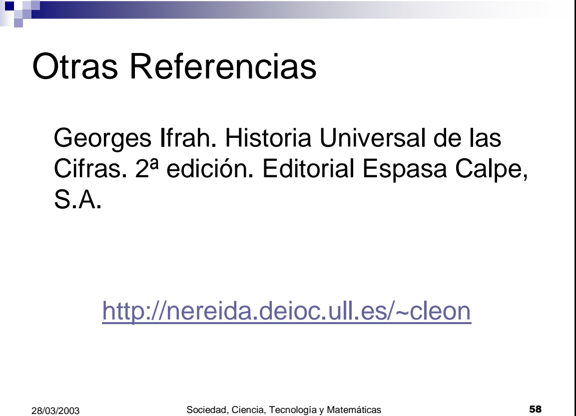

Esqueleto Divide y Vencerás Paralelo
Fase de división
Maestro
Esclavos
Maestro
Esclavos
28/03/2003
Sociedad, Ciencia, Tecnología y Matemáticas
53


Esqueleto Divide y Vencerás Paralelo
Fase de combinación
Maestro
Esclavos
Maestro
Esclavos
28/03/2003
Sociedad, Ciencia, Tecnología y Matemáticas
54


Esqueleto Divide y Vencerás Paralelo
Fase de combinación
Maestro
Esclavos
Maestro
Esclavos
28/03/2003
Sociedad, Ciencia, Tecnología y Matemáticas
55


Esqueleto de Ramificación y Acotación Paralelo
Maestro
Esclavos
Maestro
Esclavos
28/03/2003
Sociedad, Ciencia, Tecnología y Matemáticas
56


Conclusiones
Herramientas
Modelo
Algoritmo
Implementación Secuencial
Herramientas
Implementación Paralela
28/03/2003
Sociedad, Ciencia, Tecnología y Matemáticas
57


Otras Referencias
Georges Ifrah. Historia Universal de las Cifras. 2ª edición. Editorial Espasa Calpe, S.A.
http://nereida.deioc.ull.es/~cleon
28/03/2003
Sociedad, Ciencia, Tecnología y Matemáticas
58