Parallel computing in combinatorial optimization

C. Roucairol

INRIA

UNITE DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tel. (1) 39 63 55 11

Rapports de Recherche

N° 979

Programme 2

PARALLEL COMPUTING IN COMBINATORIAL OPTIMIZATION

Catherine ROUCAIROL

Février 1989

RR-0979

PARALLEL COMPUTING IN COMBINATORIAL OPTIMIZATION

CALCUL PARALLELE EN OPTIMISATION COMBINATOIRE

Catherine Roucairol

Abstract

The aim of this paper is to show the interest and the impact of parallel computers on the field of non numerical algorithms. Many problems arising in Combinatorial Optimization need much faster computers than those presently available. In order to illustrate this idea, we present examples of parallel algorithms designed for problems of different difficulty :

- "easy" problems (polynomial in complexity) coming from scheduling or graph theory,
- non polynomial problems solved by enumerative methods such as Branch and Bound procedures.

Keywords : parallel computing, combinatorial optimization, parallel algorithms

Résumé

Notre but est de montrer l'intérêt et l'impact du calcul parallèle dans le domaine de l'algorithmique non numérique. En effet, des problèmes d'optimisation combinatoire peuvent être résolus par les machines actuelles (vectorielles, multiprocesseurs) plus rapidement et ainsi conduire au traitement d'examples pratiques de plus grande taille que ceux mis en oeuvre sur une machine séquentielle. Nous donnons des exemples d'algorithmes conçus pour :

- des problèmes "faciles" (polynomiaux en complexité) issus du domaine des graphes et des ordonnancements.
- des problèmes "difficiles" (non polynomiaux) résolus par des méthodes énumératives (Branch and Bound).

Mots clés : calcul parallèle, optimisation combinatoire, algorithme parallèle.

presented at School on numerical methods for vector computers, Chexbres, Switzerland, September 1988

PARALLEL COMPUTING IN COMBINATORIAL OPTIMIZATION

Catherine ROUCAIROL
INRIA & Université Paris VI

Contents

INTRODUCTION

The aim of this paper is to show the interest and the impact of parallel computers on the field of non numerical algorithms. Traditionally, emphasis has been put on numerical problems due to numerous applications demand in computational aerodynamics, weather prediction, satellite image processing, military uses ...

But, many problems arising in artificial intelligence or combinatorial optimization, also need much faster computers than presently available.

In combinatorial optimization problems, the need for higher performance computers has two main purposes :

- to solve problems faster,
- to solve larger problems.

This demand is due to the fact that pratical applications in operations research (such as, for example, layout of VLSI circuit, traveling salesman tours, theorem proving) are more and more complex and solutions must be searched faster and faster, even sometimes in real time.

The above goals can be reached not only by exploiting performances (speed and wide space) of supercomputers but above all by designing efficient algorithms well suited to the architecture of parallel computer .

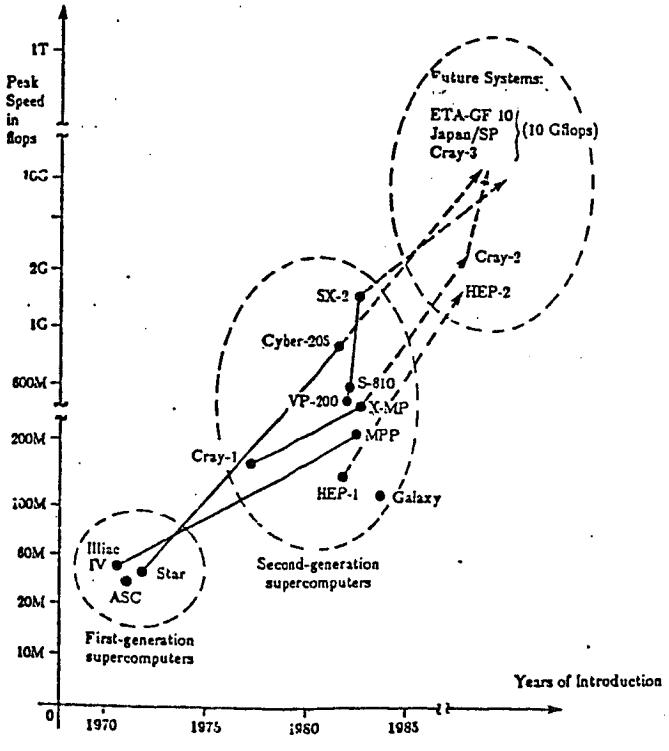


Figure 1. Space of supercomputers [HW84]

The paper is organized as follows.
Section I gives definition and examples of combinatorial optimization problems (I.1);

- proposes a few general statements about the process of designing a parallel algorithm (I.2) and some notions of parallel computing complexity for these problems (I.3)
- defines two measures of the efficiency of parallel algorithms implemented on multiprocessors (I.4).

Sections II and III respectively present examples of parallel algorithms to solve combinatorial optimization problems of different complexity :

- polynomial problems coming from scheduling or graph theory,
- non polynomial problems solved by enumerative methods (Branch and Bound procedures).

The first ones are designed to run on SIMD machines (often, with unbounded parallelism) whereas the second are implemented on commercial MIMD machines (Cray X-MP, Cray 2).

Primary sources of material are the book of M.J. Quinn [QUI87] and [ROU87].

I - Parallel algorithms for combinatorial optimization

1. Combinatorial optimization problem

1.1. Definition

A combinatorial optimization problem can be put into the form of a constrained optimization :

find a solution x subjet to a set U of constraints that optimizes some criterion function f(x)

$$\begin{cases} \min\ f(x) \\ x \in U \end{cases}$$

A solution that lies in U is called a feasible solution and a feasible solution for which f(x) is optimized is called an optimal solution, denoted by f*.

The solutions space is combinatorially large : its cardinality is finite but not enumerable.
The table below shows the time necessary to enumerate all the solutions of the solutions space with different cardinality, whenever one solution is examined by microsecond.

Table 1 Comparaison of several polynomial and exponential time complexity functions (GJ79)

Time complexity function	Size n					
	10	20	30	40	50	60
n	.00001 second	.00002 second	.00003 second	.00004 second	.00005 second	.00006 second
n^2	.0001 second	.0004 second	.0009 second	.0016 second	.0025 second	.0036 second
n^3	.001 second	.008 second	.027 second	.064 second	.125 second	.216 second
n^5	.1 second	3.2 seconds	24.3 seconds	1.7 minutes	5.2 minutes	13.0 minutes
2^n	.001 second	1.0 second	17.9 minutes	12.7 days	35.7 years	366 centuries
3^n	.059 second	58 minutes	6.5 years	3855 centuries	2×10^8 centuries	1.3×10^{13} centuries

1.2. Examples

Representative examples of such problems are :

- minimum spanning tree (MST)
  find a minimum cost connecting communication network between n processors in a distributed system

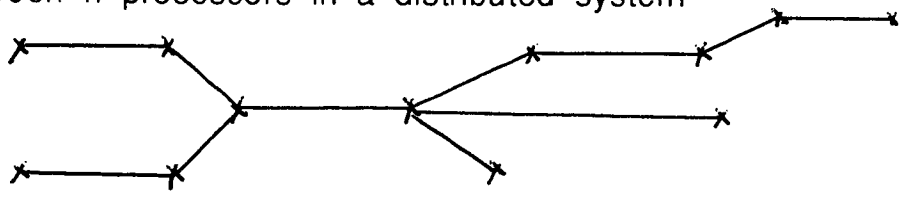


Figure 2.

- linear assignment problem (LAP)
  find a minimum cost assignment of n jobs to n men where each job is assigned to a different man

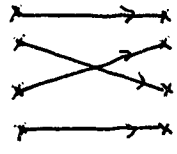


Figure 3.

- traveling salesman problem (TSP)
a traveling salesman must make a tour of n cities (1, 2,...,n) beginning and ending at city 1. His objective is to minimize the total cost or distance of his tour.

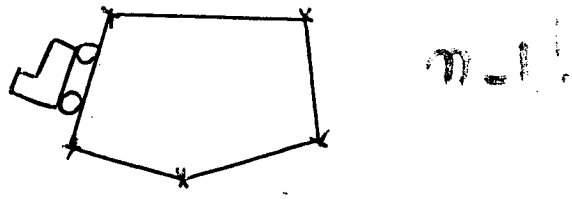


Figure 4.

- quadratic assignment problem (QAP)
assign n plants to n locations in such a way that the total cost of interplant transportation is minimized.

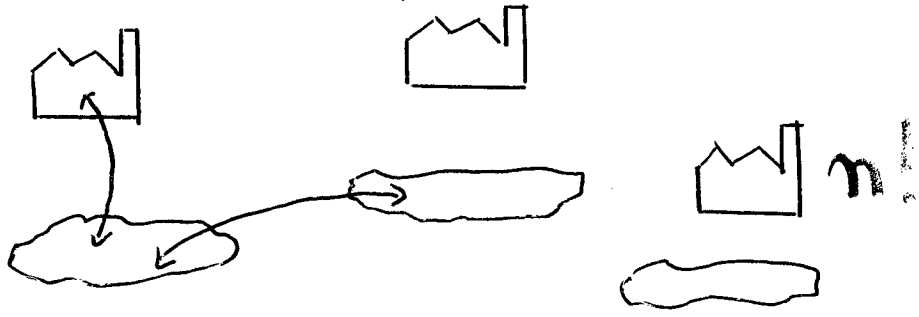


Figure 5.

According to the complexity theory for sequential computations (table 2), the former problems belong to class P (solvable by a sequential machine in polynomial time), while the latter are problems for which no polynomial time algorithm is known (NP hard).
More precisely, these two problems are in NP complete class ; they have the property that it can be solved in polynomial time iff all other NP complete problems can also be solved in polynomial time.

Table 2.

Problem	Number of solutions	Class	Computation time of algorithm
MST	n^{n-2}	P	o(n)
LAP	n!	p	o(n)
TSP	(n-1)!	NP	exponential
QAP	n!	NP	exponential

We will see in section 3 how using parallel computers, can affect the complexity of problems in P or NP classes.

3. Design of parallel algorithms

From the programmer's point of view, existing computers offer different form of parallelism (Figure 6) :

- local parallelism (at low level) an instruction is put instead of a sequence of program's instructions which consist in repeating the same operation on several different data.
  SIMD machines, Cray 1, Cyber 205 ...

- global parallelism (at high level) parallel execution of independent sequences of program's instructions(called processes)
  MIMD machines, distributed networks ...

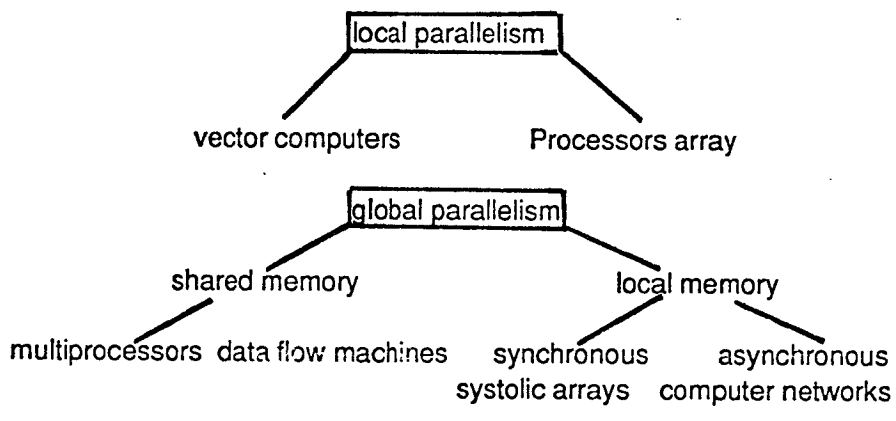


Figure 6. Local and global parallelisms

Of course, in both cases, the architecture of the selected computer will influence the design of parallel algorithms which are implemented .

In the first class (local parallelism), vector instructions provided by software must be fully used.
In the second, a decomposition of work in independant processes must be find; necessary synchronization and communication mecanisms between concurrent processes must be established in order to assure a good trade-off between communication and computation.

There are different ways to design a parallel algorithm and solve a problem:
- parallelization of any sequential algorithm
- design of a new algorithm,
- adaptation of another parallel algorithm that solves a similar program.

But, a serial algorithm has not necessarily an obvious or inherent parallelization.

For example, Hanoï's towers problem

Figure 7. Hanoî's towers
(minimize the number of necessary moves to rank disks in decreasing order of diameters on the three "towers").

More, the fastest serial algorithm does not necessarily make the most parallel efficient algorithm :
in a graph with n nodes and non negative weighted arcs, Dijkstra's algorithm finds a shortest path from one node to all the other in a time proportional to at most n^2 but does not lead to an efficient parallelized algorithm, unlike Pape, d'Esopo and Moore's algorithm whose complexity ($0(n^3)$) is larger (Parallel shortest path algorithm (COR85)).

Furthermore, the architecture often requests a new approach.
As said before , what we expect solving combinatorial optimization problems by using parallel computers is

to **increase**
- the speed of finding optimal solutions,

- the size of exactly solved problems.

This seems obvious for problems belonging to class NP; algorithms associated are exponential in time and because their computational requirements grow exponentially with the problem size, problems (even of pratical size) cannot be solved exactly. This is due to excessive running time and memory requirements (as we will see for Branch & Bound procedures in section III)
But, it is also a demand for problems in class P due to different reasons. For example, a new adherent wants to know if he can join an exchange market. Its problem is equivalent to the search of transitive closure in the large graph of adherents (n vertices) in order to test if there exists a possible circuit between all the participants. A serial algorithm, with a worst case complexity of $0(n^3)$, exists but, in this pratical case, n equals 20 000 nodes and the answer has to be given quickly..

A vectorized algorithm, implemented on a Cyber 205, ([FRA84]), computes the same problem in about 5 minutes for this size.

An other example is the linear assignment problem. This problem is a relaxation (some constraints are obmitted) of various combinatorial optimization problems such as Traveling Salesman problem, Quadratic Assignment problem... Its solution yields a lower bound on the optimal solution of these problems which is used several times in their resolution algorithm. So, even if a polynomial algorithm (worst case complexity $0(n^3)$) is known, for great value of n, a gain in time is very appreciable.

Table 3. A vectorized algorithm for LAP [LMR88]

Time in milliseconds on Cray 2 with one processor

Size	100	200	300	400	500	600
Serial algorithm	72	434.	1014	2386	3957	5165
Vectorized	15	77	157	375	612	790
Speed-up	4.7	5.6	6.4	6.3	6.4	6.5

3. Notions of parallel complexity
([KL86], [LA86])

A selection of results on parallel computation which is relevant for combinatorial optimization, is presented in this section.
A standard theoretical model of parallel computations is the PRAM (Parallel Random Access Machine). It is a synchronized machine with an unbounded number of processors and a shared memory (unbounded sequence of registers) ; each processor is a RAM (one-address computer).
Other variants of this model are EREW-PRAM (exclusive read, exclusive write) or CREW-PRAM (concurrent read, exclusive write).
An hypothesis known as the parallel computation thesis :
time bounded parallel machines are polynomially related to space bounded sequential machines
holds, for example with the PRAM machine model.
The class of problems solvable by a PRAM in polynomial time is equal to PSPACE , the class of problems solvable by a sequential machine in polynomial space.
Many problems in P can be solved in **polylogarithmic parallel time** (0(logn)) - i.e. in time that is polynomially bounded in the logarithm of the problem size n.
So, easy problems become "very easy problems" :

a sorting algorithm requires $0(\log n)$ with $0(n^2/\log n)$ processors.
a shortest path $0(\log^2 n)$ with $0(n/\log n)$.

But, some problems in P do not admit solution in polylog parallel time; they have been shown to be log space complete for P - i.e - they belong to P and any other problem in P is reducible, by a transformation using logarithmic work space: linear programming and maximum flow.

4. Performance measures

Important measures of the efficiency of parallel algorithms implemented on multiprocessors are speed-up and efficiency.

Speed-up

The speed-up S achieved by a parallel algorithm A executing an instance of a problem π, is defined as the ratio between T_1 and T_n:

$$S(A,\pi) = \frac{T_1}{T_n}$$

where T_n is the time taken by the parallel computer executing the parallel algorithm using n processors, T_1 is the time taken by the execution of the optimal serial algorithm (best average complexity) with only one processor.

Remarks

(i) This time is measured
- either theoretically (average number of operations, or number of iterations between two synchronizations in case of parallel synchronous algorithm);
In case of asynchronous machine, a parallel algorithm is linked with several executions times, which is due to non-determinism; thus, it would be preferable to compute the average time;
- or experimentally (dedicated CPU time).

(ii) Some authors take as T_1, the time of the parallel algorithm with one processor. But, synchronization and communication statements are included in the parallel program and penalize the serial execution. It just gives an easy way to compare experimentally parallel executives with an increasing number of processors.

(iii) Average speed up on all the same instances size of a combinatorial optimization problem can be usefully considered.

Efficiency

The efficiency of a parallel algorithm running on n processors is the speed-up divided by n.

Speed-up S is limited by a number of factors. Amdahl's law gives the formula:

$$S = \frac{1}{(1-f) + f/p}$$

where f is the inherently sequential fraction of a computation to be solved by p processors.
But this law assume that each processor will carry out identical amounts of computation.
We will see in section III that superlinear speed-up are possible; because it is not always possible to choose for a given problem instance the best serial algorithm and because in parallel extra processors may enable the parallel algorithm to find the optimal solution very quickly. Of course , one can say it is not possible because a sequential computer can always emulate a parallel computer.

II. Graph algorithms

In this section, we examine a number of parallel algorithms developed to solve problems in graph theory . Section 1 presents the graph terminology used in the rest of the paper; Section 2 a simple example of preemptive scheduling; Section 3 some parallel algorithms for shortest path problem and their implementation on a Cray X-MP computer.

Primary references for this section are (KL86), (DK83), (COR85).

1. Graph terminology

A **graph** G =(V,E) consists of a finite set V of **nodes** (vertices) and a set E of **arcs** (edges) joining the nodes.

In a **directed** graph, every arc is an ordered pair (i,j) and j is the successor node of i.

A **path** $\{v_1,v_2,\ldots,v_j\}$ from v_1 to v_j is a sequence of nodes such that (v_1,v_2), $(v_2,v_3),\ldots,(v_{j-1}, v_j)$ are arcs of G. The **length** of a path is the number of arcs on it.

A **cycle** (circuit in directed graph) is a path where first and last nodes are identical.

A graph is **connected** if there is a path between any two nodes of the graph.

A **tree** is a connected undirected graph with no cycle.

In a **weighted** graph, a real number is assigned to each arc; this number (which represents a length, a time, a cost or a probability...) is the value of the arc.

2. Scheduling

(DK83)(KL86)

Classical methods of designing parallel algorithm for SIMD machines, as recursive doubling and broadcasting of results, are illustrated on a relatively simple example: preemptive scheduling.

The problem is to find a schedule that minimizes the finish time of n jobs J_j, with processing time p_j, when m identical machines are available. Preemption of jobs is allowed; it means that a job can be process on a machine and the remainder further on another machine (nonoverlapping intervals of time).

McNaughton's rule gives an optimal schedule of jobs in o(n) time. Using the computation of an obvious lower bound of the finish time t*,

$$t^* = \max \{\max\{ p_j \;/\; 1 \le j \le n\}, \text{sum}\{ p_j \;/\; 1 \le j \le n \} \;/\; m \}$$

an optimal schedule with a minimum finish time of t*, is constructed.

Job 1 is scheduled on machine 1 from 0 to p_1 and job 2 from p_1 to $\min \{ p_1+ p_2 , t^* \}$. If $p_1+ p_2 > t^*$, then the remainder of job 2 is done on machine 2 starting at time 0. If $p_1+ p_2 +p_3 > t^*$, then job 3 is scheduled on machine 1 from p1+ p2 to $\min \{ p_1+ p_2 + p_3 , t^* \}$.

If the triple (M_i, s, t) indicates that job J_j is to be done by machine M_i from time s to time t, the sequential algorithm can be formally written:

$t^* \leftarrow \max\{\max\{ p_j \mid 1 \le j \le n\}, \text{sum}\{ p_j \mid 1 \le j \le n \} / m \}$

$s \leftarrow 0 ; i \leftarrow 1;$

for $j \leftarrow 1$ to n **do**

if $s + p_j \le t^*$ **then** assign (M_i, s, $s + p_j$) to J_j,

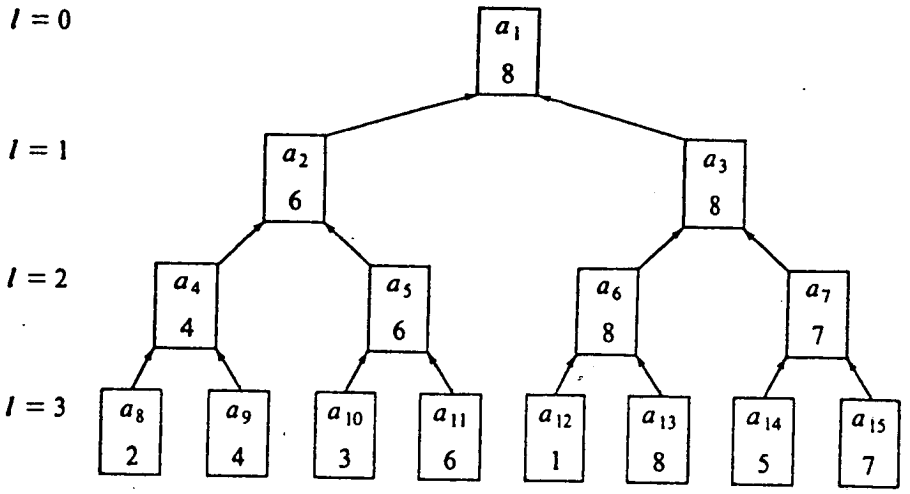
$s \leftarrow s + p_j$

else assign (M_i, s, t^*) and (M_{i+1}, 0, $p_j - (t^* - s)$) to J_j,

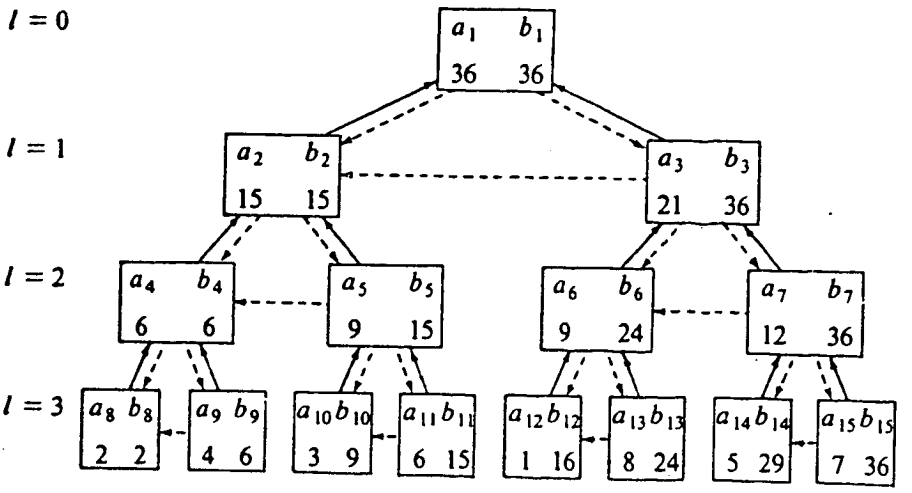
$s \leftarrow p_j - (t^* - s)$; $i \leftarrow i + 1$;

endfor

Parallel algorithms using recursive doubling (see binary computation tree below) computes maximum $\{\max\{ p_j \mid 1 \le j \le n\}$ as well as partial sums $\text{sum}\{ p_j \mid 1 \le j \le n \}$ in o(logn) time .



Maximum finding: an instance with $n = 8$.



Partial sums: an instance with $n = 8$.

Figure 8. Max and partial sums (KL86)

As all starting time and machines indices can be calculated simultaneously, the following parallel algorithm requires o(logn) time with (n/ logn) processors.

$t^* \leftarrow \max \{\max\{ p_j \ / \ 1 \le j \le n\} , \text{sum}\{ p_j \ / \ 1 \le j \le n \} \ / \ m \}$

for j ← 1 to n **do in parallel** $q_j \leftarrow \text{sum}\{ p_k \ / \ 1 \le k \le j - 1 \}$

for j ← 1 to n **do in parallel**

$s_j \leftarrow q_j \bmod t^*$; $i_j \leftarrow \lfloor q_j / t^* \rfloor + 1$;

if $s_j + p_j \le t^*$ **then** assign $(Mi_j , s_j, s_j + p_j)$ to J_j

else assign (Mi_j, s_j, t^*) and $(Mi_j + 1, 0, p_j - (t^* - s_j))$ to J_j

endfor

3. Shortest path

Parallel algorithms for two kinds of problems are reported:
(1) finding the shortest path between every pair of nodes,
(2) finding the shortest path from a specified node - source s - to all others nodes in the graph.

3.1. All-pairs shortest path

Given a weighted directed graph with n nodes, Floyd's algorithm finds path between all-pair (i,j) node, if the graph does not contain a negative circuit (sum of arcs on it is negative).
Matrix D, $D = (d_{ij})$, gives the length d_{ij} of every arc (i,j).

```
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
        d(i,j) ← min {d(i,j) , d(i,k) + d(k,j) }
        endfor
    endfor
endfor
```

Deo proposes the following algorithm. For a given k, the two inner loops access to line k and column k only for reading while they access other coefficients of D for writting. Furthermore, these computations are independant: new value depends on the precedent value and elements value in range k. So these two loops can be parallelized.

```
for k ← 1 to n do
    for 1 ≤ (i , j) ≤ n do in parallel
        d(i,j) ← min {d(i,j) , d(i,k) + d(k,j) }
    endfor
endfor
```

This yields to an $o(n^3/p)$ parallel CREW algorithm for p processors ($p \le n^2$).

3.2.1. Implementation on a Cray X-MP

The Cray X-MP is configured with four identical CPUs that share an eight million 64-bit word ECL bipolar central memory arranged in 64 interleaved banks; each CPU has a 9.5 ns clock and a memory bank cycle time of 38 ns.

Processors communicate by simultaneous reading or exclusive writing on the shared central memory.

For our experiments, codes have been written in Fortran 77 and designed in order to take advantage of **automatic vectorization** (it concerns the do -statements which do not include any if or goto-statement).

A **multitasking library** provides different mechanisms for parallel processing:

-to create or wait for termination of created task(s)TSKSTART(-),TSKWAIT(-)
-to control event created for synchronization EVPOST (-),EVWAIT (-),EVCLEAR(-)
-to control communication LOCKON (-), LOCKOFF (-)

3.2.2. Algorithm

This algorithm avoids synchronization with busy-waiting processes at the end of each iteration (new value of k).

Definition of variables:
NBACTIF is the number of active tasks
MUTEX semaphore for mutual exclusion on the share variable NBACTIF
DERNIER contains the name of the last active task
EVENT(-) event posted or cleared
IEVENT index of blocking event

Initially, nbtask is the number of tasks created,

NBACTIF ← nbtask
MUTEX ← OFF
DERNIER ← 0
EVENT(0) ← cleared
EVENT(1) ← posted
IEVENT ← 0

Process(p)

```
For k ← 1 to n do
        for i ← p to n by step of nbtask do
            if d(i,k) <+∞ then
                        for j ← 1 to n do
                            d(i,j) ← min {d(i,j) , d(i,k) + d(k,j) }
                        endfor
            endif
        endfor
        LOCKON (MUTEX)
            NBACTIF ← NBACTIF – 1
            if NBACTIF = 0 then p is the last active process
                                DERNIER ← p
            endif
            n°event ← IEVENT
        LOCKOFF(MUTEX)
```

```
if DERNIER = p then
                    DERNIER ← o ;
                    NBACTIF ← nbtask;
                    IEVENT ← 1 - IEVENT;
                    EVCLEAR (EVENT (IEVENT));
                    EVPOST (EVENT (1 - IEVENT))
                else
                    EVWAIT(EVENT(n°event))
        endif
endfor
```

3.3. Single-source shortest path
(QUI87)(COR85)

Pape, d'Esopo and Moore proposes an algorithm where:
- a queue contains nodes for which further searching must be done; initially, it contains s.
- while queue is non empty, node u from the head of the queue is removed, all arcs (u,v) are examined. Length (here for value) of the path from s to v is compared to length of a path from s to u arc (u,v) . If it is updated, v is added to the tail of the queue (if not already in queue).
-algorithms terminate when queue is empty.

```
global  distance,   {Element i contains distance from s to i}
        n,          {Number of vertices in graph}
        s,          {Source vertex}
        weight      {Contains weight of every edge}
begin
  for i ← 1 to n do
    INITIALIZE(i)
  endfor
  insert s into the queue
  while the queue is not empty do
    SEARCH
  endwhile
end

SEARCH:
local   new_distance,   {Distance to v if pass through u}
        u,              {Examined edge leaves this vertex}
        v,              {Examined edge enters this vertex}
begin
  dequeue vertex u
  for every edge {u, v} in the graph do
    new_distance ← distance(u) + weight({u, v})
    If new_distance < distance(v) then
      distance(v) ← new_distance
      If v is yet in the queue then
        enqueue vertex v
      endif
    endif
  endfor
end
```

Deo's parallel version is based upon a shared queue and a number of asynchronous processes (QUI87)

```
global   distance,   {Element i contains distance from s to vertex i}
         halt,       {Set to true when it is time for processes to stop}
         n,          {Number of vertices in graph}
         p,          {Number of processes}
         s,          {Source vertex}
         weight      {Contains weight of every edge}
begin
  for all P_i, where 1 ≤ i ≤ p do
    for j ← i to n step p do
      INITIALIZE(j)
    endfor
  endfor
  enqueue s
  halt ← false
  for all i, 1 ≤ i ≤ p do
    repeat SEARCH (i) until halt
  endfor
end


SEARCH (i):
parameter  i              {Process number}
local      new_distance,  {Distance to v if go through u}
           u,             {Edge is directed from this vertex}
           v              {Edge is directed to this vertex}
begin
  lock the queue
  if the queue is empty then
    waiting(i) ← true
    if i = 1 then
      halt ← waiting(2) and waiting(3) and ... and waiting(p)
    endif
    unlock the queue
  else
    dequeue u
    waiting(i) ← false
    unlock the queue
    for every edge {u, v} in the graph do
      new_distance ← distance(u) + weight({u, v})
      lock (distance(v))
      if new_distance < distance(v) then
        distance(v) ← new_distance
        unlock (distance(v))
        if v is not in the queue then
          lock the queue; enqueue v; unlock the queue
        endif
      else unlock (distance(v))
      endif
    endfor
  endif
end
```

An other version of Pape, d'Esopo and Moore's algorithm has been implemented on Cray X-MP(COR85).

III.Tree search

1. Introduction

Branch and Bound (B&B) algorithms are the most efficient know means for solving many NP-hard problems. They can also be viewed as the most general technique for the search for solutions in a combinatorially large problem space, which is a major problem in Operations Research and also Artificial Intelligence. Backtracking, dynamic programming, decision trees like AND-OR tree, α-β search are variations of B&B algorithms. But as the computational requirements of these algorithms (time and space) grow exponentially in the problem size, possibilities of overflowing storage and running out of time can stop the program before reaching the optimal solution. So, the idea to realize a parallel implicit enumeration of the solutions has naturally emerged.
Section 2 gives a brief description of sequential B&B principles while section 3 discusses different ways to parallelize them, and gives two methods to design parallel asynchronous algorithms. Computational results obtained on a Cray X-MP with a wellknown combinatorial optimization problem - the traveling salesman Problem - are reported in section 4 .
Proofs (theoretical and experimental) that they can exhibit anomalous speed-up are presented in section 5.

2. Sequential B&B algorithm

Let us recall that the goal of the B&B algorithm is to solve a constrained optimization problem :

$$\begin{cases} \min f(x) \\ x \in X \\ \text{where} \end{cases}$$

X represents the domain of optimization
x is a solution, x is feasible iff $x \in X$
f(x) is the value of the solution

The B&B method is based on the idea of intelligently enumerating all the feasible solutions of a combinatorial optimization problem ; we have seen in section I.1. that it is hopeless to look at all the solutions !
The underlying idea of the B&B algorithm is to decompose a given problem into two or more subproblems of smaller size. Then, this partitionning process is repeatedly applied to the generated subproblems until each unexamined one is either partitioned, solved or shown not to yield an optimal solution of the original problem.

The process to exclude a subproblem from further consideration, is based upon the computation of a lower bound on the value of solutions within each subset: subproblems whose bounds exceeds the value of some known solution can be discarded.

The state of the partitionning process at any time can be represented as a partial tree in which subproblems are represented by nodes and the decomposition of a problem into subproblems by arcs from father node to each son node.

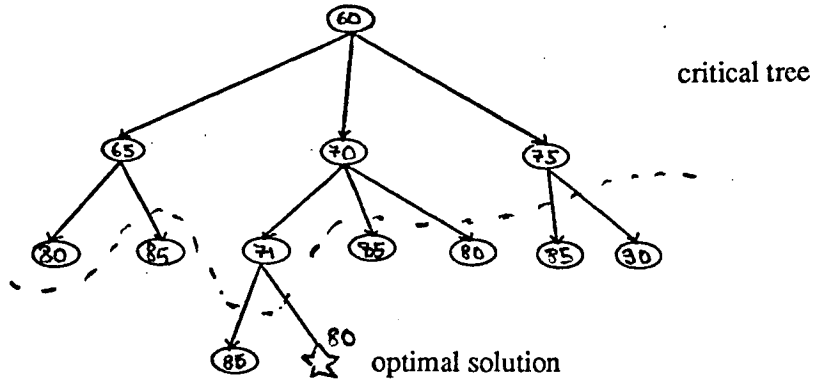


Figure 9.

More precisely, a B & B algorithm has three major components:
a branching scheme, a bounding function, a search strategy.

Let B=(S, Γ) be the tree built, S the set of nodes and Γ the application successor.

Branching principle Γ

The root of the tree S_0 is the set X (or a set of solutions U with X included in U). Successors $\Gamma(S_k)$ of every node S_k are defined by the branching scheme, which must satisfy three conditions:

(c1) union of partitionned subsets is equal to the initial set

(c2) number of node partitions necessary to reach an optimal solution is finite

(c3) a node -associated to a subset of solutions- cannot be partitionned (expanded) when:
-this subset is reduced to only one feasible solution
-this subset does not contain any feasible solution or feasible solutions whose cost is greater than the cost value of the best known feasible solution. Such a node is called a leaf of the B&B tree.

Bounding function v

A lower bound function v assigns to each subset of solutions a real number representing a lower bound cost for all complete solutions in the set.It has the following properties:

(p1) $v: S \rightarrow \mathbb{R} \cup \{\infty\}$

(p2) for every S_k node of B associated with a set of solutions

$v(S_k) \leq \min \{ f(s) / s \in S_k \}$

(p3) for every S_k leaf node of B, $v(S_k) = \min \{ f(s) / s \in S_k \}$

(p4) if none feasible solution belongs to S_k, $v(S_k) = +\infty$

(p5) v is non decreasing on S: if S_l is a successor of S_k then $v(S_l) \geq v(S_k)$

(p6) let suppose that an upper bound UB of f^* is known (for example UB is the value of a feasible solution found by any heuristic method or the cost of the cheapest solution yet seen); whenever S_k is such that

$v(S_k) \geq UB$ then S_k is not selected further for expansion.

Search strategy h
A strategy is a rule for choosing to which of the currently active nodes the branching principle should be applied. For conceptual simplicity and uniformity of notation, we assume that a heuristic function h , h: S→ℕ , gives a priority to each node and thus the selected node is the one with the smallest heuristic value.

The choice of a heuristic function depends on the application; the most used are:
depth first $h(S_i)= l(S_i)$ where $l(S_i)$ is the level of node S_i - length of the path from S_0 to S_i-
best first $h(S_i)= v(S_i)$.

First strategy leads quickly to a feasible solution and is space-saving: a node at a lower level will always be examined before a node at a higher level so the set of active nodes can be maintened in a FIFO list.
Second strategy is not as blind; good solutions are first searched but all active nodes not yet examined must be stored.
However, we will see further (in section 5) that the total number of nodes expanded is minimum in the sense that any branching operation performed under this strategy must also be performed under other strategies , provided some condition on bound's values.

Dominance relation may also be introduced in B& B algorithms; it is a relation on subproblems such that if S_i dominates S_j, then S_j cannot contain a better solution than S_i and thus S_j can be eliminated .

Procedure B&B
liveset = set of active nodes
```
begin
    liveset = { root};
    if root is a solution node then UB = f(root) else UB =∞;
    while liveset contains a node S with v(S) < UB do
            begin
            x = node in liveset with min h( );
            delete x from liveset;
            / branching scheme /
            create successors nodes of x;
            / evaluation /
            for each successor y of x do
                    begin
                    if y is a solution and f(y) < UB then UB= f(y);
                    if y is not a leaf and v(y) < UB then add y to liveset;
                    end
            end
end
```

3. Differents ways to parallelize B&B algorithm

While dividing the work - expanding parts of the B & B tree - among a bounded number of processes, we have to avoid the following situations in order to reduce enumeration:
- several processes expand the same part of the tree,
- a process is working on node of the subtree whose lower bound 's value is greater than the value of a solution found by an other process in an other part of the tree.

So, each process must broadcast all information it obtained and upper bound UB must be constantly updated. This can influence the choice of the next expanded node and some branch which was developed with a sequential algorithm, can be pruned.
It clearly appears that to synchronize processes for information exchange (processes wait for others which have not completed their tasks) is a solution more easy to handle but it will increase the overall complexity of the algorithm. This hypothesis have been verified by experiments with parallel synchronous B & B algorithms, conducted by Mohan on Cm* at Carnegie Mellon (MOH 84), and Kindervater on ICL DAP at Manchester (KIN 85).
We retain two ways to distribute the B & B algorithm's operations on a bounded number of processors(LAROU84,85, ROU87).

Asynchronous parallel B & B algorithms

Vertical

Every process search a complete subtree independantly of the others.
The information exchanged between various processes is reduced to one value per complete solution. As soon as this bound is communicated, this the responsability of each process to eliminate nodes with lower bound greater than this value.
The principal difficulty comes from necessary synchronizations between processes which have terminated their enumeration in order to obtain quickly and consistently some other work from their partners.
This kind of algorithm is dedicated to computer networks or multiprocessors with limited memory space and slow interprocessor communication.
The principal criticism is that some processes may be working on node of the subtree that could be eliminated if a better communication (last upper bound) was designed.

Horizontal

Each process works on one node of the tree:
it generates iteratively the sons of one node (*branching*),
it calculates a lower bound cost for all complete solutions that can be generated from this successor node (*bounding*).
A specialized process -the sheduler process- keeps track of live nodes, maintains the current value of the best known feasible solution, schedules the processes which expand either a live node one or two levels or a small number of nodes before returning for re-scheduling.
The advantage is that the bound information is kept reasonably up-to-date.
But as the number of processors grows the scheduler process and message transport mechanism will be bootlenecks.
So, this algorithm is rather dedicated to a system with a modest number of processors and an efficient message passing scheme, like a multiprocessors with a shared memory.

4. An horizontal parallel B & B algorithm for the Cray X-MP

We now give an adaptation of the previous ideas to a shared memory multiprocessors machine.

4.1. Programming facilities on Cray X-MP

The Cray X-MP is configured with four identical CPUs that share an eight million 64-bit word ECL bipolar central memory arranged in 64 interleaved banks; each CPU has a 9.5 ns clock and a memory bank cycle time of 38 ns. Processors communicate by simultaneous reading or exclusive writing on the shared central memory.
For our experiments, codes have been written in Fortran 77 and designed in order to take advantage of **automatic vectorization** (it concerns the do -statements which do not include any if or goto-statement).
A **multitasking library** provides different mechanisms for parallel processing:
-to create or wait for termination of created task(s)TSKSTART(-),TSKWAIT(-) -to control event created for synchronization EVPOST (-),EVWAIT (-),EVCLEAR(-)
-to control communication LOCKON (-), LOCKOFF (-)

4.2. Main characteristics of the algorithm

The *distribution of work* among the different processes (one process assigns to one of the four processors) is done by giving access to a shared list which contains information about every node to be expanded.
This list is implemented as a heap (binary tree) in order to use fast existing algorithms to insert, sort and remove items.
A priority (adapted to the problem to be solved) is associated to each node. Nodes are ranking by decreasing priority: every active process finds at the top of the list the node it has in charge.
Insertion of items is done whenever the expansion of a node generates several successors whose evaluation is less than the best known upper bound. The best known upper bound (BKUB) is a shared variable which is updated whenever a local upper bound (lub), less than BKUB, is found at a node to be expanded.
Items are suppressed either at the beginning or at the end of the list. The former case occurs whenever an inactive process looks for a new job, the latter case occurs whenever a lub is less than BKUB: every node whose evaluation is greater than lub is eliminated because it cannot lead to an optimal solution.
The algorithm terminates whenever the list is empty and all the processes are inactive.

4.3. Definition of variables

Variables used in multitask code can be categorized as follows:

shared variables
LIST: list of items implemented
is
as a heap;

NBLIST: number of nodes in LIST,
BKUB: value of best known feasible solution,
empty.
COUNT: number of active processes.

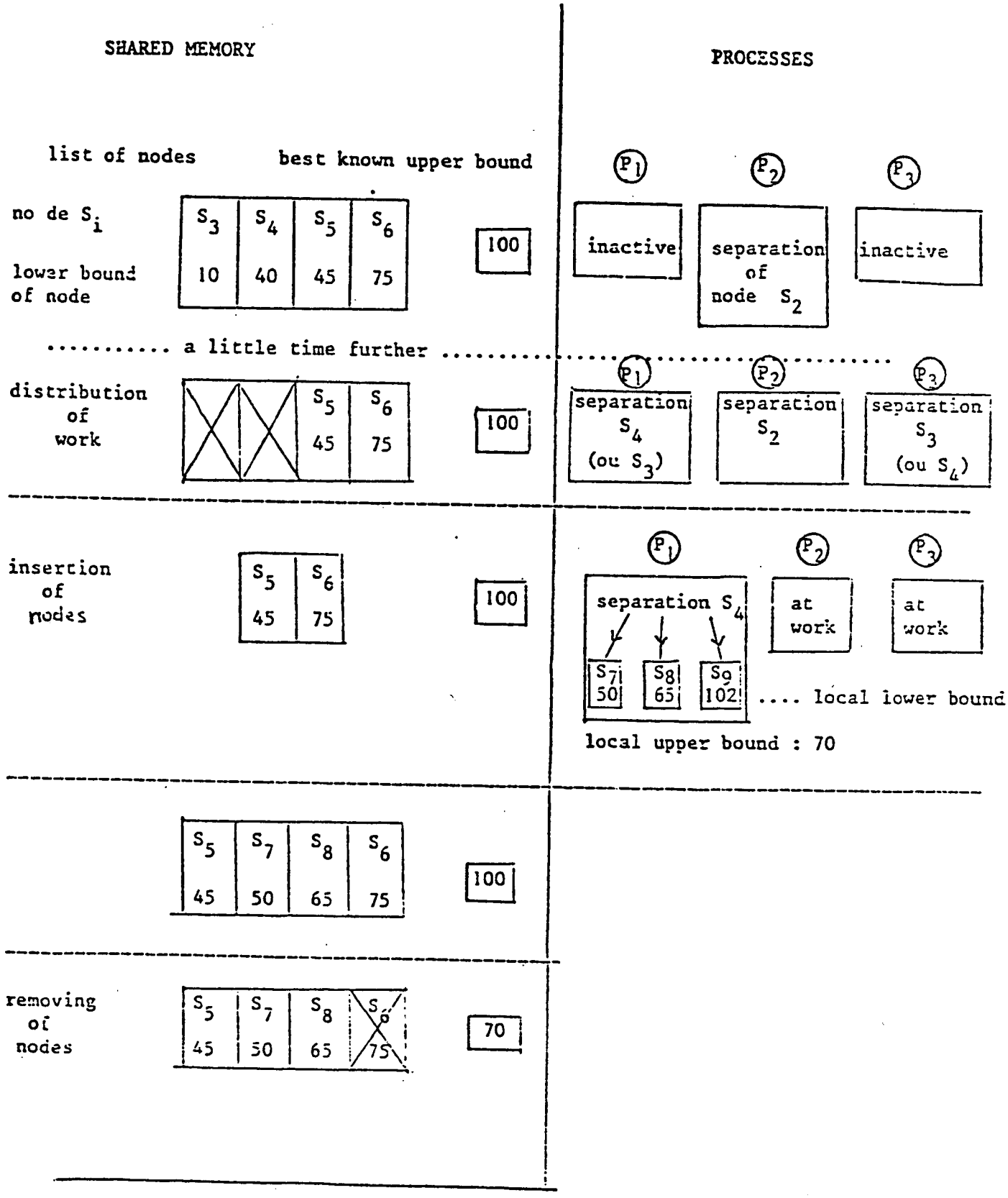
local variables
lub: upper bound

llb: lower bound associated with a node.

event variable
INSER: this event

sent when a process
inserts items in the
list that was

Example (minimization problem - three processes)



4.4. The program

```
Initial process
begin
COUNT = root of thetree; FIN = false
branch to this node and create its successors
compute its upper bound (lub)
compute lower bound (llb) for each successor of the root
BKUB = lub
for each successor node do
                    if llb < BKUB then insert this node in LIST
                    NBLIST = NBLIST +1
start concurrently several node processes
end
Node process
begin
    end = false
    while end = false do
        lock (NBLIST)
        if  NBLIST = 0 then
                            if COUNT = 0 then
                                        end = true
                                        unlock (NBLIST)
                                        send event (INSER)
                                        else
                                        unlock (NBLIST)
                                        wait event (INSER)
                        else
                        COUNT = COUNT + 1
                        select the node at the top of LIST
                        delete top of LIST
                        NBLIST = NBLIST- 1
                        unlock (NBLIST)
                        create the successors of this elected node
                        compute upper bound (lub)
                        compute lower bound (llb) for each successor node
                        read BKUB
                        if lub < BKUB then
                                lock (NBLIST)
                        suppress in LIST each node
                                            with llb ≥ BKUB
                                decrease NBLIST ; BKUB = lub
                                unlock (NBLIST)
                        lock (NBLIST)
                        for each successor node  do
                                if lub < BKUB  then
                                            if NBLIST = 0 then send  (INSER)

                                            insert this successor in LIST
                                           NBLIST = NBLIST + 1
                                COUNT = COUNT - 1
                                unlock (NBLIST)
end
```

4.5. Parallelization of a B & B algorithm for Traveling Salesman Problem

We have experimented this algorithm for various combinatorial problems: Traveling salesman(ROU86), Quadratic assignment (ROU87), Multiknapsack

(PR87, PRV88), but we present here results obtained with the parallelization of the most efficient algorithm for the asymmetric TSP.

4.5.1. Serial B & B algorithm

Given a complete oriented and weighted graph in which the weight of an arc (i,j) represents the cost of traveling from i to j, the traveling salesman problem is to find a circuit of minimum cost that goes through every node exactly one.
The TSP can also be formulated as:

$$\begin{cases} \min \Sigma_i \Sigma_j c_{ij} x_{ij} \\ \Sigma_j x_{ij} = 1 \quad i \in J & (1) \\ \Sigma_i x_{ij} = 1 \quad j \in J & (2) \\ \Sigma_{i \in S} \Sigma_{j \in J/S} x_{ij} \geq 1 \quad \forall S \subset J & (3) \\ x_{ij} = 1 \text{ if arc (i,j) belongs to tour with minimum cost,} & (4) \\ \quad = 0 \text{ otherwise.} \\ \text{where } J=\{1,2,\ldots,n\} \text{ and } c_{ij} \in N \quad , c_{ij} \neq c_{ji}. \end{cases}$$

Carpateno and Toth (CT80) proposes an efficient algorithm for this asymmetric problem($c_{ij} \neq c_{ji}$). It is based on subtour elimination approach.
At a node k of the B & B tree, the *lower bound* is obtained by solving the linear assignment problem defined by constraints (1), (2), (4) and two sets of additional constraints: E_k set of arcs excluded from the solution belonging to node k, I_k set of arcs included in.
To compute this bound, the Hungarian algorithm (polynomial, $O(n^3)$) is used.
The branching scheme is what we call a **polytomic branching scheme** (ROU84):
a node is split into more than two nodes.
Let us suppose that the solution of the LAP at a node k of the subtree contains several subtours(figure below). The subtour that is selected to be eliminated, is one with the minimum of arcs not yet included in the current solution. From node k, three successors nodes will be created:

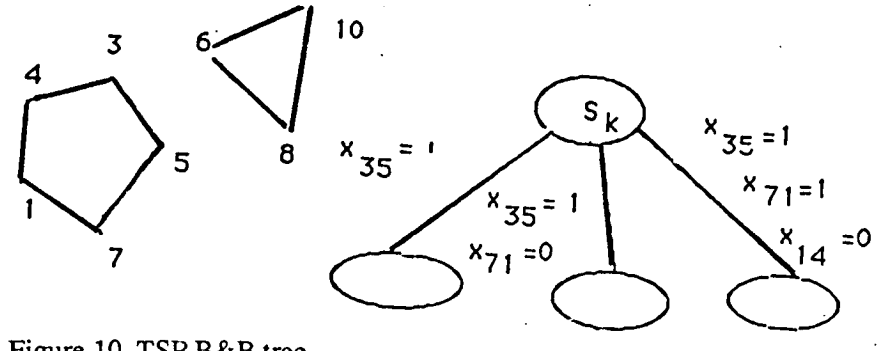


Figure 10. TSP B&B tree

We can observe that this kind of branching scheme increase the number of parts that can be carried out in parallel.

4.5.2. Experimental results

We report here our experiments with the horizontal parallelization (other results in (ROU86)).

First results have been obtained on an emulator called CREM running on a Multics machine BULL DPS68 at INRIA, France and last results on the Cray X-MP 48 at CRAY RESEARCH, Minneapolis, USA.

Two kinds of tests were conducted:
- in the first series the number of processes is increased from 1 to 3 on CREM, from 1 to 4 on Cray in order to see the improvement;
- the second series compare various search strategies in the B & B algorithm for a given number of processes.

Speed-up
Coefficients of matrix C are generated from a uniform distribution in a range $[\alpha,\beta[$. Best first strategy is used. Speed-up is computed here as: T_1 / T_i where T_1 is the solution time for one processand T_i the solution time for i processes.

Number of processors	Average simulated time in seconds (CREM)	Average number of nodes in the B & B tree
1	373	10
3	150	11
Average speed-up = 2.5		Average increase of nodes $=N_1/N_2 = 1.1$

Table 4. 40 TSPs of size 200, $c_{ij} \varepsilon [0,100[$.

Moreover , with one processor it was impossible to find optimal solution for 6 problems of size 200 and 21 of size 150 in a time limited to 900 seconds.

Number of processors	Average simulated time in seconds (CREM)	Average number of nodes in the B & B tree
1	> 687	> 8
3	424	11.5
Average speed-up >1.6		Average increase of nodes $=N_1/N_3 < 1.4$

Table 5. 40 TSPs of size 150, cij ε [0,1000[.

But, these results hide a part of reality: worst speed-up is 0.5, best 12.5. The next table shows the differences between problem instances.

Anomaly	Class	Number of TSPs	Average speed-up S	N_3/N_1
	S < 0.8	3	0.7	2.8
detrimental	S ε [0.8, 1.2]	11	1	2.8
deceleration	S ε [1.2, 2.8]	11	1.8	1.5
	S ε [2.8, 4]	7	3.1	0.9
acceleration	S >4	8	6.9	0.5

Table 6. Anomalies

This table indicates that time is directly related to the number of nodes expanded in the search tree.

For problems of class (1) and (2), the number of examined nodes is increased by a factor 3 so there is no improvement by using parallelism.
For problems of class (5), this number is divided by 2 and the experimental speedup is greater than the number of processes used; it is a case of **"super-linear" speed-up !**
Such effects of parallel B & B algorithms with a given problem's instance are called **anomalies**:

detrimental anomaly	$S_p < 1$	where speed-up S_p equals T_1 / T_p
deceleration anomaly	$S_p < p$	ratio of time with 1 and p processors
acceleration anomaly	$S_p > p$	for the parallel B & B algorithm

These anomalies are due to the fact that, for a problem's instance, the search strategy which leads to the best serial B & B algorithm is not known "à priori". Causes will be studied in more details in the next section.

	Number of nodes				Speed-up			Time (seconds) dedicated time	
Size	Processor P_1	P_2	P_3	P_4	P_2	P_3	P_4	T_1	T_4
50	18	20	1	1	1.8	16.2	14.4	0.21	0.014
100	2	5	5	7	0.5	0.5	0.4	0.04	0.01
200	6	13	11	14	1.6	1.6	1.6	1.8	1.3

Table 7 .Running time for some problems on Cray X-MP

Testing of strategies
According to the fact that the nodes in the list are ranked in increasing order of their lower bounds, four strategies are used:
"best lower bound first" selects a node at the top of the list, the second "deepest parent node first" at the bottom, the third randomly, the fourth chooses a node at the bottom with probability p_0= (BKUB - binf)/BKUB and at the top with probability 1-p_0, where binf= v(first node inlist).

	Best		Bottom		Random		Binomial	
Size	N	T	N	T	N	T	N	T
50	6.3	10.5	8	17.2	7.3	12	6	11.6
100	9.3	40	15.3	76.7	12.7	60.8	9.3	57.8
150	13	96	18.7	363	27	221	15	111
200	11.7	148	19	273	27.3	385	16.3	188
250	14.7	421	19.3	484	13.3	291	15	432

Table 8. Strategies with 3 processes on CREM, N = number of nodes, T= simulated time in seconds.

It is interesting to stress two points:
- even with a single processor machine and a simulated parallelism, optimal results have been obtained for problems which were not solved in sequential. It means that just asynchronous or merely non determinism of choice can improve the performance of a pure sequential and deterministic algorithm. Of course, performances can be improved with a real parallel machine.
- speed-up is almost linear with respect to the number of processors. However, this have to be refine a bit; there exist speed-up which are more than linear and these cases are more frequent in this experiment than sub-linear speed-up.

5. Anomalies

Anomalious behaviours of B &B algorithms under parallel processing have been studied by many authors:
The computational efficiency of these algorithms depends on the bounding function, the data structure and the search strategy.

Let call *B & B tree* the tree generated by the B & B algorithm,
critical tree subtree whose nodes S_i are such that $v(S_i) < f^*$,
minimum tree subtree whose nodes S_i are such that $v(S_i) \leq f^*$
(contains minimal number of nodes to expand in order to find an optimal solution)

A B &B algorithm, A = (Γ bounding principle, v bounding function, h strategy), is *h-optimal* if the B & B tree is minimum.

Theorem: if T^*_1 is the time with a h-optimal B & B (called A) to expand all the nodes of the minimum tree, T_p the time with a parallelization of A on p processors, the speed-up is:

$$T^*_1 / T_p \leq p$$

This theorem shows that anomaly can occurs when the serial B & B algorithm is not h-optimal.

Let define now some properties of the bounding function:

v is discriminating if $\forall (S_i, S_j)\ v(S_i) \neq v(S_j)$

h consistent with v $h(S_i) \leq h(S_j) \Rightarrow v(S_i) \leq v(S_j)$

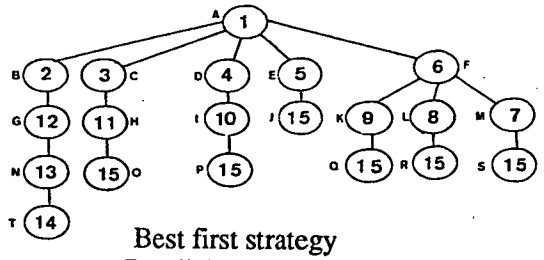
v weak $\forall S_i$ which is not an optimal solution $v(S_i) \neq f^*$.

Theorical results have ben obtained under some assumptions:
- horizontal parallelization,
- all processors are synchronized.

Time is measured as number of iterations required; at each *iteration*, p processes expand in parallel the p first priority nodes of the list (provided this list contains at least p nodes) and add their children to the share list.
We summarize the contains of several theorems whose proofs can be found in Lai and Sahni(Lais83), Lai and Sprague(LAI85),Li and Wah(LIW84),Wah and Yu(WAY82,85), (Rou87):
- for a sequential B & B algorithm, if v is discriminating, a best first strategy h build the critical tree,
- with a parallel B &B algorithm, if h is a best first strategy and v is discriminating, the speed-up $S_p \leq p$,
- moreover, under these conditions, this strategy is robust $S_p \equiv p$,

($p - t(p-1)/Tp \leq T1 / Tp \leq p$, t height of the B & B tree with 1 processor)
- necessary condition for good anomalies is : h inconsistent with v ,
- sufficient conditions for bad anomalies are : v discriminating and h consistent with v.

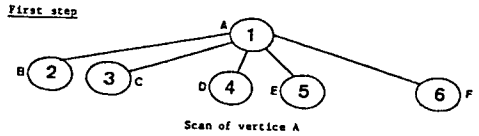
Next figures give examples of anomaly when increasing the number of processors increase the number of iterations.

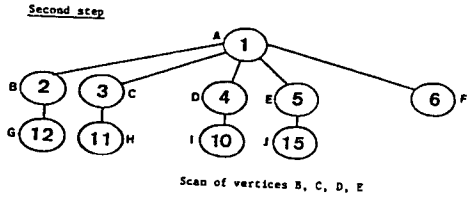


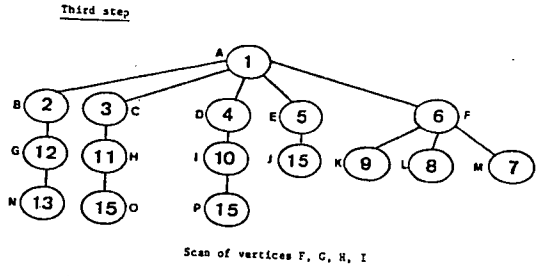
(number in a node = lower bound)

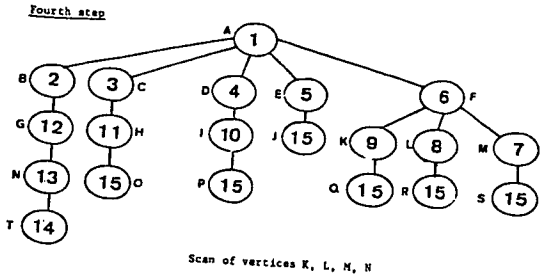
Best first strategy
Parallel horizontal synchronous algorithm

4 processors

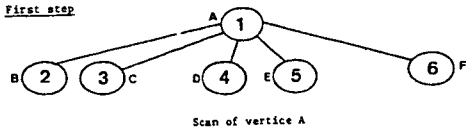


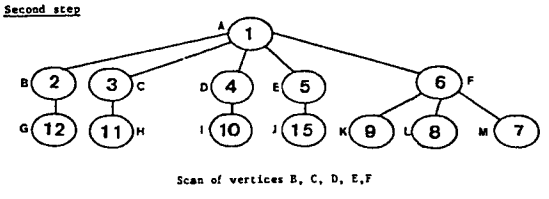


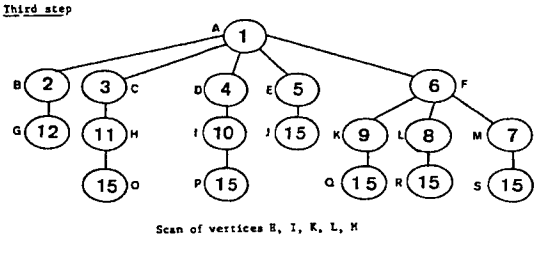


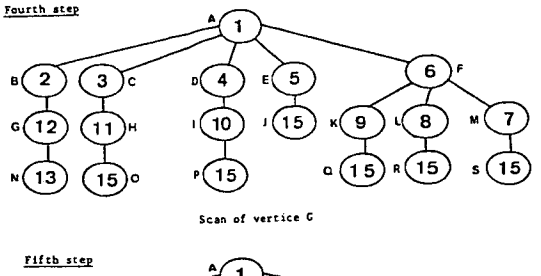


5 processors









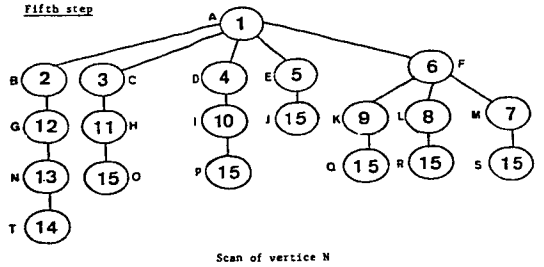


Figure 11. Anomalies

REFERENCES

(COR85) Cornelus P., Algorithmes parallèles de cheminement dans les graphes Mémoire d'ingénieur, IIE-CNAM, Evry 1986.
(FRA84) Fraisse P.,Vectorisation d'algorithmes de théorie des graphes, RR LRI, Bat 490, Univ. d'Orsay, 1984.

(GJ79) Garey M.R and Johnson D.J., Computers and untractibility: a guide to the theory of NP-completeness, W.H.Freeman and company, N.York, 1979.

(HW84) Kai Hwang, Evolution of modern supercomputers, in Supercomputers: design and applications, IEEE Catalog number EH02 19-6, 5-8, 1984.

(KIN85) Kindervater G., H.Trinekens, Experiments with parallel algorithms for combinatorial problems, Report OS-R8512, Center for Mathematics and Computer Science, Amsterdam,1985.

(KL86) Kindervater G.A.P. and Lenstra J.K., An introduction to parallelism in combinatorial optimization, Discrete Applied Math. 14, 135-156, 1986.

(LA86) Lavault C., Parallel and distributed complexity : new notions and developments", presented at EURO VIII, Lisboa, Portugal, 1986.

(LAS84) Lai H., Sahni S., Anomalies in Branch and Bound , Com. of ACM, Vol 27, n°6, 594-602, 1984.

(LASP85) Lai T., Sprague, Performance of Branch and Bound algorithms, IEEE Trans. on Comp., C-34, n°10,194-201,1985.

(LAROU 85) Lavallée I., Roucairol C., Parallel Branch and Bound algorithms, presented at Euro VIII Congress, Bologna, Italy, RR MASI n° 112, Univ. Paris VI,1985.

(LIW84) Li G., Wah B., Computational efficiency of parallel approximate branch and bound algorithms, Proc. of the Int. Conf. on Parallel Processing, 473-480,1984.

(LMR 88) Lichnevsky A.,Marchand D. and C.Roucairol, A fast vectorized version of a linear assignment algorithm, RR INRIA, à paraître 88.

(MOH84) Mohan J., A study in parallel computation: the traveling salesman problem, Carnegie Mellon University, CMU-CS-82-136, 1982.

(QUI87) Quinn M.J., Designing efficient algorithms for parallel computers, MC Graw Hill Series in Supercomputing and Artificial Intelligence.

(ROU84) Roucairol C., An efficient branching scheme in branch and bound procedure, presented at TIMS XXV, Copenhagen, RR Université Paris VI 79-4, 1983.

(ROU84) Roucairol C., A parallel branch and bound algorithm for the quadratic assignment problem, Disc. Apl. Math. 18,211-225,1987.

(ROU86) Roucairol C., Experiments with parallel algorithms for the asymetric salesman problem, presented at EUROVIII,Lisboa, Portugal.

(ROU87) Roucairol C., Du séquentiel au parallèle: la recherche arborescente et son application à la programmation quadratique en variables 0-1, Thèse d'état, Université Paris VI, Juin 1987.

(WAY82) Wah B., Yu C., Probabilistic modeling of branch and bound algorithms, Proc. COMPSAC, 647-653,1982.

(WAY85) Wah B., Yu C., Stochastic modeling of branch and bound algorithms with best first search, IEEE Trans. on Software Engineering, Vol SE-11, n°9,922-934,1985.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique