Computers and Artificial Intelligence, Vol. 6, 1987, No. 3, 195—208 

# THE RING MACHINE

Burkhard MONIEN, Oliver VORNBERGER

*Universität Gesamthochschule Paderborn,*
*Postf. 1621, 4790 Paderborn, FRG*

**Abstract.** A ring-structured network of asynchronous processors is constructed with personal computers (SIRIUS I). Each computer in the ring has its own microprocessor, local memory for code and data, and can communicate via two serial ports with its neighbours, i.e. can exchange information with its predecessor and its successor in the ring. This distributed system allows the parallel execution of sequential backtracking algorithms, i.e. a general-purpose package is implemented together with a user interface for the specific application. To demonstrate the performance of the ring machine the Hamiltonian cycle problem is solved for 50 graphs with up to 16 ring members.

**Кольцевая ЭВМ**
Б. Моньен, О. Форнбергер

**Резюме.** Кольцеобразная сеть асинхронных процессоров построена с ПЭВМ (СИРИУС I). Каждая машина в кольце имеет свой микропроцессор, локальную память для кода и данных и с помощью двух последовательных портов может общаться с соседями, т. е. может обмениваться информацией со своим предшественником и преемником в кольце. Эта распределенная система позволяет параллельную обработку последовательных алгоритмов бэктрекинга, т. е. используется общецелевой пакет вместе с интерфейсом пользователя для специфического применения. Для демонстрации работы кольцевой ЭВМ проблема гамильтоновского цикла решена для 50-ти графов с помощью 16-ти устройств кольца.

## 1. INTRODUCTION

In the beginning of the seventies COOK [1] and KARP [4] introduced the notion of NP-completeness, which since then has been the centre of many research efforts [3]. This term characterizes a class of combinatorial problems (members are e.g. Hamiltonian cycle, vertex cover, satisfiability) that behave — relative to their inherent solution complexity — equivalent in the following sense:

1) Any algorithm that solves any of the NP-complete problems fast (i.e. in polynomial time) can be transformed to fast algorithms for all NP-complete problems.

2) The proof that a specific NP-complete problem cannot be solved in polynomial time can be extended to the whole class.

Since no one has ever found such a fast algorithm, the term of NP-completeness can, despite the absence of a formal proof, be used as a synonym for intractability. So, NP-complete problems will — at least in the worst case — consume an intolerable amount of execution time, even for modest-sized problems.

One way to overcome this difficulty is to use approximation algorithms that produce near optimal solutions in short running times. Another way is parallelism:

Since the performance of sequential computers cannot be improved because of electronic principles, in recent years computer architectures have evolved that consist of a system of several processing units which are connected by a communication network. The goal of such super computer is to produce additional processing power by the use of additional processing units.

In general, it is not obvious how to take advantage of the "parallel hardware", since it requires a special "parallel software". The philosphy of the von Neumann computer, modifying an exclusive memory by sequential steps, has tuned programmers to produce tricky 1-processor-code and only very slowly they adjust to the need of programs that exploit parallelism.

In this paper we present an approach that bypasses these two difficulties, namely to have access to special hardware and to write parallel software. We present a general-purpose package together with a user interface that allows the parallel computation of sequential backtracking algorithms on a set of personal computers.

The organization of the paper is as follows:
In Section 2 we relate our approach to previous work. In Section 3 the hardware configuration is presented. Sections 4 and 5 illustrate sequential and parallel backtracking respectively. In Section 6 we describe the user interface, in Section 7 we discuss the computing times and speedup resulting from a typical implementation. Section 8 closes with suggestions for further research.

## 2. RELATED WORK

Several authors have used parallelism to speed up the computation of NP-complete optimization problems. J. MOHAN [5] solved the travelling salesman problem on the CM*, a multiprocessor system built at Carnegie Mellon University, consisting of a hierarchical network of LSI-11 computers. Using 16 processors, MOHAN got 1/8 of the 1-processor running time for 30-node instances. R. FINKEL und U. MANBER [2] proposed DIB, a general-purpose package for distributed backtracking implementations on the crystal multicomputer, a collection of 16 VAX-11/750 computers connected by a 10 Mb sec token ring. MANIP, a parallel machine for processing NP-hard problems was introduced by WAH and MA [10]. Their theoretical analysis for a suitable interconnection network was confirmed by simulation results. LAI and SAHNI [5] considered anomalies in parallel branch and bound algorithms by presenting conditions under which $k > 1$ processors need more time than 1 processor.

The common point to the implementations of MOHAN, FINKEL and WAH is a

multicomputer, i.e. a special designed hardware that provides global memory to all processors or allows fast communication between the processors. Both are necessary for a parallel backtracking algorithm that uses a single subproblem stack. Our attempt, on the other hand, will start from standard hardware: the personal computer, a mass product, available at low rates and powerful enough to act as an integrated part of a (loosely coupled) parallel system.

Since the main philosophy of our architecture is to handle independent subproblems of asynchronous processes, it is also very well suited to organize parallel branch-and--bound algorithms. The experiences relevant to these topics have been reported in [9]. Our ring machine has also been used to study superlinear speedups produced by a parallel implementation of testing Boolean formulas for satisfiability [7].

## 3. THE HARDWARE

The basic ingredients of any multiprocessor network are processing units (called nodes) and connections between them (called links). Most universities, banks and insurance companies are equipped with a lot of personal computers that are used for programming tasks and administration work. Each personal computer (like the SIRIUS I at the University of Paderborn) has

— its own microprocessor (Intel 8086)
— its own main memory for program code & data (256 KB)
— two serial ports to the outside world: (RS 232).

Each port can

- send one byte
- receive one byte

} (at a rate of 9600 bits per second).

These properties obviously qualify such a personal computer to act as a processing unit in our parallel system. How do we design the connections? There are mainly two possibilities:

*Centralized version*: A supervisor (with special hardware and software) is introduced which is connected to all the nodes. It acts as an operating system and is responsible for balancing the work load (see Fig. 1).

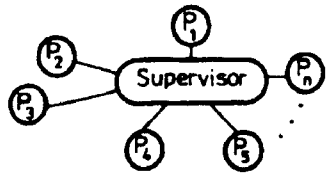


Fig. 1.

*Decentralized version*: All nodes form a ring by connecting the left port of a computer to its left neighbour and its right port to its right neighbour (see Fig. 2).

There are several advantages inherent to the decentralized version:

— simple concept

— no special designed hardware for supervisor
— not only parallel computation but also parallel communication (in the centralized version the supervisor forms a bottleneck, since it can manage only one pair of nodes talking to each other at a time).

We will see later that the main disadvantage, namely the very restrictive communication pattern, will take effect only partly when this network is running our backtracking strategy.

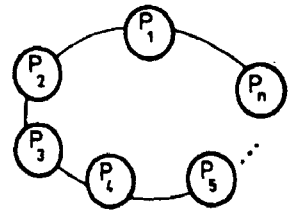


Fig. 2.

## 4. SEQUENTIAL BACKTRACKING

One solution technique to solve NP-complete problems is backtracking. Here the total solution space is passed through in a systematic manner until either a solution is found or the search turns out to be unsuccessful. Let us demonstrate for the Hamiltonian cycle problem how to implement a sequential backtracking.
(To recall the problem: Given a directed graph $G = (V, E)$ with node set $V$ and edge set $E$. the question is whether $G$ has a Hamiltonian cycle, i.e. a cycle running over each node exactly once.)

In order to solve the Hamiltonian cycle problem we have to check for each edge the "consequences" when it is excluded from or included into the solution cycle. So, given the problem $G = (V, E)$ we can choose an edge $e = (x, y)$ and can form the subproblems $G_{\bar{e}}$ and $G_e$ (meaning: $e$ does not belong to the solution and $e$ belongs to the solution respectively) by either deleting $e$ (resulting in $G_{\bar{e}}$) or by shrinking the endpoints of $e$ into a new node (resulting in $G_e$); see Fig. 3.

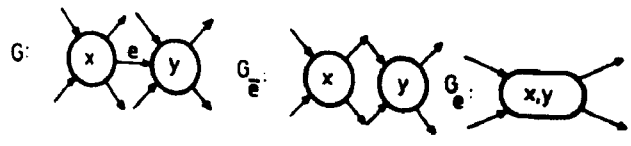


Fig. 3.

Notice that $G_e$ has also lost the edges leaving the (former) node $x$ and going into the (former) node $y$. This is justified by the fact that the explicit inclusion of an edge $e$ into the solution cycle implies an explicit exclusion of the adjacent edges.

It is easy to see that $G$ has a Hamiltonian cycle iff $G_{\bar{e}}$ has a Hamiltonian cycle or $G_e$ has a Hamiltonian cycle. Furthermore, the solution for $G$ can be constructed from the solution from $G_{\bar{e}}$ or $G_e$.

The two graphs resulting from the splitting may allow us to "simplify" them: a node $x$ of outdegree one forces its only outgoing edge $e$ into the solution cycle and a node $y$ of indegree one forces its only incoming edge $e$ into the solution cycle. This leads to a sequence of implications that only stop when there are no longer nodes of outdegree or indegree one. Clearly, if a graph results with a node of degree zero, it can be excluded from further examinations.

If we store the still unsolved subproblems in a stack, the backtracking procedure looks like this:

```
push   input graph G on the stack
repeat
       pop top graph G from the stack
       simplify G
       if G is small
       then begin
                  if G is Hamiltonian then report cycle
            end
       else begin
                  choose an edge e
                  split G and push G_i and G_e on the stack
            end
until  the stack is empty or a solution has been found.
```

## 5. PARALLEL BACKTRACKING

By adding a communication procedure at the beginning of the repeat loop we can extend in a very natural way the sequential backtracking procedure to a parallel program for our ring structure.

```
procedure communicate
begin
      if the stack is empty then ask your left neighbour for a problem;
      if the right neighbour asks for a problem and the stack is not empty
      then cut the bottom graph from the stack and send it
end;
```

The repeat-condition has to be changed to
"until all stacks are empty or a solution has been found".

Notice that the size of the subproblems on the stack decreases from the bottom to the top. This stems from the fact that splitting a graph $G$ results always in graphs with fewer edges and/or fewer nodes. So the topmost subproblem in the stack is the smallest (measured in "free variables") and therefore the easiest to solve; the bottom graph in the stack represents the largest subproblem. Since we wish to supply an idle processor with work that lasts again for a while (until the next request is necessary), it makes sense for

a processor to get rid of its largest problem first and therefore send its bottom stack element.

At the beginning only one node in the ring (master) has the input graph in its stack, the other nodes (slaves) start with an empty stack. Starting with the master the original graph will be split into smaller and smaller pieces which are spread by requests of idle nodes over the ring. By this technique the solution space is divided into disjoint subspaces.

What has been described so far had been the first version of our parallel backtracking procedure. After we had conducted several experiments it became clear that some tuning was necessary, mainly to compensate the following two effects:

1) At the beginning the work is distributed extremely unevenly: the master has the whole input problem, all slaves are idle. So they all start to ask their neighbours for work at the same time and it takes quite a while until all ring members are supplied with work. Even then they start with problems of very different size, since slave $i$ starts with a graph that is a restriction of the graph of slave $i-1$.
2) An idle processor is supplied with work from its neighbour. However, it often takes only a few iterations to completely solve this subproblem and the processor asks once more for work. This leads to high interaction and results in loss of performance, since each time the processors have to synchronize for their communication.

So in order to support a better balanced work load the implementation is augmented by two heuristics:

1) To postpone the first need of processor communication it is important to provide in the beginning all ring members with roughly the same amount of work. First, starting with the master each processor sends a copy of $G$ to its right neighbour. Assume there are $k = 2^l$ nodes in the ring (if $k$ is not a power of 2, analogous considerations are possible). Consider edges $e_1, e_2, \ldots, e_l$. Then the set $\mathcal{G} := \{G_{x_1 x_2 \ldots x_l} | x_i \in \{e_i, \bar{e}_i\}, 1 \leq i \leq l\}$ consists of $k$ graphs which form a partition of $G$ into disjoint subproblems. So if all ring members agree on the selection of $e_1, e_2, \ldots, e_l$, each process r can — on receiving $G$ — compute its start problem and then enter the main repeat-until loop.
2) Upon request not only one problem is sent but several depending on some heuristic arguments such as the total number of problems currently in the stack.

Let us summarize the main philosophy of our implementation. Parallelism is organised by a ring of independent stand alone computers, each of which can exchange data with its 2 immediate neighbours at a low transmission rate. If it is possible to reduce the need of communication by keeping the ring members busy (i.e. keep them working on their local stack), sending a few slow bytes once in a while will not affect the total efficiency too much.

## 6. USER INTERFACE

Our parallel system provides an interface that requires from the user four procedures. These procedures are application-specific and must be formulated in a sequential manner, i.e. as if they were used in a conventional single processor system. In fact, any

sequential backtracking program for this specific user application would basicly consist of these four components:

*procedure create-problem*; (only used by the master)
The user generates (either randomly according to specific requirements or by reading the terminal input) one instance of his problem class and stores it in a suitable encoding on a file.

*procedure init-stack*; (only used by the master)
The user reads a problem instance from a file and pushes it as the first subproblem onto the stack.

*procedure split-problem*; (used by both the master and the slave)
This is the basic part of any backtracking algorithm. The user removes the topmost subproblem $P$ from the stack. If $P$ turns out to have a solution, then this solution is put in a buffer $B$, provided by the system. If $P$ turns out to have no solution, this is a "dead end" and nothing is done. If $P$ must be divided for further examination, then the user splits $P$ into disjoint subproblems $P_1, P_2, \ldots, P_r$ and pushes these onto the stack. It has to be guaranteed that $P$ has a solution iff $P_1, P_2, \ldots, P_{r-1}$ or $P_r$ has a solution.

*procedure show-answer*; (only used by the master)
The content of the buffer $B$ is displayed.

It should be emphasized that our approach requires the sequential backtracking algorithm to be formulated in a specific manner (see below). It is clear that our method is not a general "parallelizer" in the sense that any backtracking algorithm (in what notion so ever) is automatically transformed to a parallel version.

However, the requirements we impose are very weak:

1) The algorithm has to be formulated in an iterative version, i.e. the basic "backtracking step" has to be specified: Pop a subproblem from the stack, split it into subproblems and push them onto the stack.
2) The encoding of the subproblem has to provide all information necessary to let this subproblem be solved independently of other subproblems. This requires e.g. that subproblems contain those parts of the solution vector that have been found so far.

These four routines together with the system utilities for communication in the ring form the parallel system. Notice that at the end of the computation the system collects the solutions (if any were found) from the local buffers and moves one into the master buffer. So if the master finds its buffer empty there is no solution. How do we detect the end of the computation if no solution is found? A request for a subproblem is expressed by the master with a "master request". A request for a subproblem by a slave is expressed as a "slave request" unless the slave itself was asked for a subproblem and could not provide one: in this case it formulates its need by a "master request". Clearly, when the master receives a "master request" he can conclude that all slaves (and he himself) are idle and therefore initiate a stop signal.

Table 1. Execution times in seconds.

| | yes/no | Number of processors | | | | |
|---|---|---|---|---|---|---|
| | | 1 | 2 | 4 | 8 | 16 |
| Graph No. 01 | − | 40353 | 20573 | 10288 | 5152 | 2901 |
| Graph No. 02 | + | 3554 | 3605 | 425 | 322 | 330 |
| Graph No. 03 | + | 5148 | 5184 | 5172 | 1126 | 852 |
| Graph No. 04 | − | 11711 | 5900 | 2975 | 1521 | 863 |
| Graph No. 05 | + | 541 | 33 | 35 | 39 | 41 |
| Graph No. 06 | + | 130 | 134 | 21 | 22 | 26 |
| Graph No. 07 | − | 30764 | 15489 | 7741 | 3899 | 2000 |
| Graph No. 08 | − | 3906 | 1979 | 1011 | 535 | 375 |
| Graph No. 09 | + | 57 | 60 | 61 | 61 | 61 |
| Graph No. 10 | + | 1250 | 36 | 38 | 23 | 25 |
| Graph No. 11 | + | 21 | 25 | 17 | 20 | 24 |
| Graph No. 12 | + | 583 | 16 | 16 | 17 | 23 |
| Graph No. 13 | − | 27221 | 13841 | 6909 | 3486 | 1929 |
| Graph No. 14 | − | 25611 | 13028 | 6518 | 3293 | 1994 |
| Graph No. 15 | + | 1327 | 54 | 55 | 61 | 53 |
| Graph No. 16 | + | 1283 | 1321 | 1322 | 15 | 20 |
| Graph No. 17 | + | 20607 | 21067 | 17104 | 145 | 42 |
| Graph No. 18 | + | 12 | 15 | 17 | 16 | 20 |
| Graph No. 19 | − | 13 | 9 | 11 | 14 | 19 |
| Graph No. 20 | − | 186 | 123 | 82 | 74 | 61 |
| Graph No. 21 | − | 412 | 229 | 139 | 100 | 79 |
| Graph No. 22 | + | 26 | 16 | 16 | 17 | 23 |
| Graph No. 23 | + | 404 | 14 | 16 | 20 | 27 |
| Graph No. 24 | − | 2345 | 1192 | 639 | 341 | 222 |
| Graph No. 25 | + | 12 | 15 | 17 | 21 | 27 |
| Graph No. 26 | + | 26 | 29 | 22 | 22 | 21 |
| Graph No. 27 | − | 6106 | 3083 | 1573 | 822 | 520 |
| Graph No. 28 | − | 4152 | 2105 | 1077 | 568 | 323 |
| Graph No. 29 | + | 14 | 15 | 16 | 19 | 21 |
| Graph No. 30 | + | 57 | 59 | 62 | 63 | 51 |
| Graph No. 31 | − | 722 | 382 | 204 | 139 | 103 |
| Graph No. 32 | − | 2429 | 1253 | 637 | 354 | 283 |
| Graph No. 33 | + | 13 | 15 | 17 | 19 | 25 |
| Graph No. 34 | − | 11718 | 5900 | 2968 | 1505 | 848 |
| Graph No. 35 | − | 4630 | 2338 | 1188 | 613 | 401 |
| Graph No. 36 | + | 12 | 14 | 16 | 16 | 20 |
| Graph No. 37 | + | 1032 | 1038 | 517 | 46 | 47 |
| Graph No. 38 | − | 3071 | 1569 | 801 | 429 | 267 |
| Graph No. 39 | + | 65 | 68 | 70 | 23 | 27 |
| Graph No. 40 | − | 3066 | 1576 | 809 | 422 | 282 |
| Graph No. 41 | + | 440 | 41 | 44 | 17 | 20 |
| Graph No. 42 | + | 20 | 21 | 24 | 24 | 24 |
| Graph No. 43 | + | 23 | 25 | 27 | 24 | 29 |
| Graph No. 44 | + | 144 | 62 | 39 | 35 | 36 |
| Graph No. 45 | + | 15 | 17 | 17 | 19 | 23 |
| Graph No. 46 | + | 21 | 23 | 24 | 28 | 31 |
| Graph No. 47 | + | 7376 | 7455 | 7453 | 7370 | 2933 |
| Graph No. 48 | + | 780 | 794 | 21 | 27 | 32 |
| Graph No. 49 | − | 755 | 394 | 223 | 145 | 112 |
| Graph No. 50 | + | 1088 | 318 | 320 | 80 | 44 |
| per problem: | + | 1486 | 1341 | 1065 | 314 | 160 |
| per problem: | − | 9430 | 4787 | 2410 | 1232 | 714 |
| Total: | | 225252 | 132552 | 78814 | 33169 | 18560 |
| per problem: | | 4505 | 2651 | 1576 | 663 | 371 |

To help the user to evaluate the efficiency of his program and to measure the performance several statistical data are collected during the computation (see Section 7).

## 7. EXPERIMENTAL RESULTS

50 directed random graphs having 100 nodes each were generated. All graphs were forced to have in- and outdegree in the range of 2 to 4, because previous experiments had shown that without this restriction the graphs become "too easy to solve": Almost any random walk through the graph establishes a Hamiltonian cycle. This effect has been analysed in a paper by PÓSA [8], where he proved that a random graph with $n$ vertices and $c \cdot n \cdot \log n$ edges contains a Hamiltonian cycle with probability approaching 1 (as $n \to \infty$).

Table 1 shows the running times in seconds for the 50 graphs when solved on 1, 2, 4, 8, 16 nodes in the ring. The + and − respectively indicate whether the graph has a Hamiltonian cycle or not. The execution time covers the total work period, beginning with the distribution of the graph and ending with the display of the solution at the master. So communication overhead and idle times are included.

Notice that sometimes additional processors do not help in finding the solution more quickly (e.g. for graph No. 16, 1, 2 and 4 processors need roughly the same amount of time) and sometimes one additional processor speeds up the algorithm very much (e.g. 2 processors compute a Hamiltonian cycle for graph No. 10 about 30 times faster than 1 processor). The reason for this is obvious: sometimes the additional processors search "in vain" their solution spaces, sometimes they are "lucky" and detect a cycle early. For certain problem classes (e.g. satisfiability) anomalies have been observed and explained [7].

One of the most important criteria to measure the performance of a multicomputer system is the speed-up and the efficiency.

Let $T_k(P)$ be the execution time for problem $P$ using $k$ processors.

Then $S_k(P) := T_1(P)/T_k(P)$ denotes the speedup for problem $P$,

$AS_k := \sum_P T_1(P) \Big/ \sum_P T_k(P)$ denotes the average speedup and

$E_k := AS_k/k$ is called the efficiency, a measure for the utilization of the $k$ processors.

A few comments on problems related to the notion of speedup are in order. First, the average speedup is not defined as $\frac{1}{n} \cdot \sum_P S_k(P)$, where $n$ denotes the number of problem instances. This is motivated by the fact that in general several problems have to be solved in a row and one is interested in the total gain of speed. Second, we were unable to predict the average speedup by mathematical analysis. Note that in each iteration of the main while loop we can simplify the current graph. Sometimes, after several implications due to nodes of degree 1, the graph "vanishes", i.e. that branch of the solution space turns out to be non-Hamiltonian at an early stage of the computation. So it seems that the very irregular execution times for graphs of similar sizes can only be recorded by experiments.

Table 2. Speedup factor.

| | yes/no | Number of processors | | | | |
|---|---|---|---|---|---|---|
| | | 1 | 2 | 4 | 8 | 16 |
| Graph No. 01 | − | 1.00 | 1.96 | 3.92 | 7.83 | 13.91 |
| Graph No. 02 | + | 1.00 | 0.99 | 8.36 | 11.04 | 10.77 |
| Graph No. 03 | + | 1.00 | 0.99 | 1.00 | 4.57 | 6.04 |
| Graph No. 04 | − | 1.00 | 1.98 | 3.94 | 7.70 | 13.57 |
| Graph No. 05 | + | 1.00 | 16.39 | 15.46 | 13.87 | 13.20 |
| Graph No. 06 | + | 1.00 | 0.97 | 6.19 | 5.91 | 5.00 |
| Graph No. 07 | − | 1.00 | 1.99 | 3.97 | 7.89 | 15.38 |
| Graph No. 08 | − | 1.00 | 1.97 | 3.86 | 7.30 | 10.42 |
| Graph No. 09 | + | 1.00 | 0.95 | 0.93 | 0.93 | 0.93 |
| Graph No. 10 | + | 1.00 | 34.72 | 32.89 | 54.35 | 50.00 |
| Graph No. 11 | + | 1.00 | 0.84 | 1.24 | 1.05 | 0.88 |
| Graph No. 12 | + | 1.00 | 36.44 | 36.44 | 34.29 | 25.35 |
| Graph No. 13 | − | 1.00 | 1.97 | 3.94 | 7.81 | 14.11 |
| Graph No. 14 | − | 1.00 | 1.97 | 3.93 | 7.78 | 12.84 |
| Graph No. 15 | + | 1.00 | 24.57 | 24.13 | 21.75 | 25.04 |
| Graph No. 16 | + | 1.00 | 0.97 | 0.97 | 85.53 | 64.15 |
| Graph No. 17 | + | 1.00 | 0.98 | 1.20 | 142.12 | 490.64 |
| Graph No. 18 | + | 1.00 | 0.80 | 0.71 | 0.75 | 0.60 |
| Graph No. 19 | − | 1.00 | 1.44 | 1.18 | 0.93 | 0.68 |
| Graph No. 20 | − | 1.00 | 1.51 | 2.27 | 2.51 | 3.05 |
| Graph No. 21 | − | 1.00 | 1.80 | 2.96 | 4.12 | 5.22 |
| Graph No. 22 | + | 1.00 | 1.63 | 1.63 | 1.53 | 1.13 |
| Graph No. 23 | + | 1.00 | 28.86 | 25.25 | 20.20 | 14.96 |
| Graph No. 24 | − | 1.00 | 1.97 | 3.67 | 6.88 | 10.56 |
| Graph No. 25 | + | 1.00 | 0.80 | 0.71 | 0.57 | 0.44 |
| Graph No. 26 | + | 1.00 | 0.90 | 1.18 | 1.18 | 1.24 |
| Graph No. 27 | − | 1.00 | 1.98 | 3.88 | 7.43 | 11.74 |
| Graph No. 28 | − | 1.00 | 1.97 | 3.86 | 7.31 | 12.85 |
| Graph No. 29 | + | 1.00 | 0.93 | 0.88 | 0.74 | 0.67 |
| Graph No. 30 | + | 1.00 | 0.97 | 0.92 | 0.90 | 1.12 |
| Graph No. 31 | − | 1.00 | 1.89 | 3.54 | 5.19 | 7.01 |
| Graph No. 32 | − | 1.00 | 1.94 | 3.81 | 6.86 | 8.58 |
| Graph No. 33 | + | 1.00 | 0.87 | 0.76 | 0.68 | 0.52 |
| Graph No. 34 | − | 1.00 | 1.99 | 3.95 | 7.79 | 13.82 |
| Graph No. 35 | − | 1.00 | 1.98 | 3.90 | 7.55 | 11.55 |
| Graph No. 36 | + | 1.00 | 0.86 | 0.75 | 0.75 | 0.60 |
| Graph No. 37 | + | 1.00 | 0.99 | 2.00 | 22.43 | 21.96 |
| Graph No. 38 | − | 1.00 | 1.96 | 3.83 | 7.16 | 11.50 |
| Graph No. 39 | + | 1.00 | 0.96 | 0.93 | 2.83 | 2.41 |
| Graph No. 40 | − | 1.00 | 1.95 | 3.79 | 7.27 | 10.87 |
| Graph No. 41 | + | 1.00 | 10.73 | 10.00 | 25.88 | 22.00 |
| Graph No. 42 | + | 1.00 | 0.95 | 0.83 | 0.83 | 0.83 |
| Graph No. 43 | + | 1.00 | 0.92 | 0.85 | 0.96 | 0.79 |
| Graph No. 44 | + | 1.00 | 2.32 | 3.69 | 4.11 | 4.00 |
| Graph No. 45 | + | 1.00 | 0.88 | 0.88 | 0.79 | 0.65 |
| Graph No. 46 | + | 1.00 | 0.91 | 0.88 | 0.75 | 0.68 |
| Graph No. 47 | + | 1.00 | 0.99 | 0.99 | 1.00 | 2.51 |
| Graph No. 48 | + | 1.00 | 0.98 | 37.14 | 28.89 | 24.38 |
| Graph No. 49 | − | 1.00 | 1.92 | 3.39 | 5.21 | 6.74 |
| Graph No. 50 | + | 1.00 | 3.42 | 3.40 | 13.60 | 24.73 |
| Average: | + | 1.00 | 1.11 | 1.40 | 4.72 | 9.26 |
| Average: | − | 1.00 | 1.97 | 3.91 | 7.65 | 13.19 |
| Average: | | 1.00 | 1.70 | 2.86 | 6.79 | 12.14 |
| Efficiency: | | 1.00 | 0.85 | 0.71 | 0.85 | 0.76 |

Table 3. Statistical details for graph No. 13, solved with 16 processors.

Solution: There is no Hamiltonian circuit

| | CPU | TRANS | TOTAL | % | > 0 | > 5 | WAIT | REQ | IN | OUT | STACK | ITERAT |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Master | 1690 | 239 | 1929 | 87 | 11 | 1447 | 16 | 656 | 57 | 68 | 2699 | 16796 |
| Slave 1 | 1635 | 294 | 1929 | 84 | 10 | 1178 | 12 | 935 | 68 | 72 | 2740 | 15960 |
| Slave 2 | 1619 | 310 | 1929 | 83 | 11 | 1180 | 14 | 1368 | 72 | 70 | 2344 | 15475 |
| Slave 3 | 1581 | 346 | 1927 | 82 | 35 | 1088 | 14 | 1796 | 70 | 68 | 2721 | 14965 |
| Slave 4 | 1563 | 365 | 1928 | 81 | 551 | 1091 | 14 | 2808 | 68 | 55 | 2402 | 16215 |
| Slave 5 | 1596 | 333 | 1929 | 82 | 810 | 1356 | 25 | 3345 | 55 | 45 | 2812 | 16467 |
| Slave 6 | 1685 | 243 | 1928 | 87 | 699 | 1621 | 26 | 2231 | 45 | 37 | 2629 | 16545 |
| Slave 7 | 1694 | 234 | 1928 | 87 | 1207 | 1604 | 25 | 2305 | 37 | 21 | 3452 | 17235 |
| Slave 8 | 1724 | 204 | 1928 | 89 | 687 | 1716 | 21 | 2552 | 21 | 11 | 3133 | 17213 |
| Slave 9 | 1773 | 156 | 1929 | 91 | 1692 | 1718 | 25 | 2809 | 11 | 3 | 2606 | 19659 |
| Slave 10 | 1876 | 52 | 1928 | 97 | 1656 | 1906 | 23 | 840 | 3 | 3 | 2985 | 19187 |
| Slave 11 | 1880 | 48 | 1928 | 97 | 1867 | 1908 | 12 | 432 | 3 | 8 | 3792 | 20476 |
| Slave 12 | 1861 | 66 | 1927 | 96 | 1385 | 1909 | 14 | 505 | 8 | 16 | 3107 | 19449 |
| Slave 13 | 1839 | 89 | 1928 | 95 | 1383 | 1911 | 15 | 668 | 16 | 28 | 3472 | 18791 |
| Slave 14 | 1805 | 122 | 1927 | 93 | 1263 | 1912 | 13 | 528 | 28 | 43 | 3329 | 18453 |
| Slave 15 | 1759 | 169 | 1928 | 91 | 1172 | 1791 | 23 | 616 | 43 | 57 | 3208 | 18310 |
| Total | 27580 | 3270 | 30850 | | | | | 24394 | 605 | 605 | 47431 | 281196 |
| Average | 1723 | 204 | 1928 | 88 | 902 | 1583 | 18 | 1524 | 37 | 37 | 2964 | 17574 |

Table 2 shows speedup and efficiency. The average speedup is also listed for graphs with Hamilton cycle (+) and for graphs without Hamilton cycle (−). The individual speedup ranges from 0.44 (the communication overhead for 16 processors solving graph No. 25 outweighs by far the absolute short execution time for 1 processor) to 490.64 (slave No. 11 found a solution in the 16-processor ring for graph No. 17 after 42 seconds).

Tables 3-5 show statistical data in detail for graphs No. 13, 44, 47 when solved on a 16-node ring.

For each processor the following is given:

| | |
|---|---|
| CPU | time in seconds spent on working on the graph |
| TRANS | time in seconds spent on waiting for a problem or transmitting a problem |
| TOTAL | total time in seconds, i.e. CPU + TRANS |
| % | work load in percent, i.e. 100 * CPU/TOTAL |
| > 0 | first moment that the node has run out of work |
| > 5 | first moment that the node has to wait more than 5 seconds to get work from his neighbour |
| WAIT | maximal time period the node has spent for waiting to get work |
| REQ | number of requests |
| IN | number of problems received from the left neighbour |
| OUT | number of problems sent to the right neighbour |
| STACK | maximal stack size in bytes |
| ITERAT | number of iterations of the repeat-until loop. |

Table 4. Statistical details for graph No. 44, solved with 16 processors.

Solution (found by slave No. 13): There is a Hamiltonian circuit:

| | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 99 | 58 | 19 | 91 | 82 | 18 | 51 | 76 | 80 | 33 | 21 | 34 | 74 | 94 | 25 | 89 | 12 | 100 | 5 |
| 13 | 10 | 4 | 35 | 64 | 81 | 7 | 48 | 57 | 16 | 95 | 70 | 77 | 3 | 60 | 79 | 30 | 17 | 67 | 52 |
| 87 | 6 | 92 | 98 | 53 | 43 | 50 | 14 | 26 | 44 | 45 | 46 | 24 | 22 | 20 | 54 | 40 | 85 | 2 | 71 |
| 61 | 29 | 90 | 97 | 32 | 65 | 15 | 28 | 47 | 9 | 66 | 56 | 72 | 39 | 78 | 49 | 55 | 83 | 86 | 96 |
| 41 | 11 | 38 | 73 | 42 | 8 | 93 | 68 | 31 | 88 | 37 | 84 | 36 | 63 | 75 | 69 | 23 | 27 | 62 | 59 |

| | CPU | TRANS | TOTAL | % | > 0 | > 5 | WAIT | REQ | IN | OUT | STACK | ITERAT |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Master | 27 | 9 | 36 | 75 | 6 | | 3 | 1 | 1 | 1 | 1880 | 260 |
| Slave 1 | 24 | 11 | 35 | 68 | 5 | | 5 | 31 | 1 | 1 | 2083 | 221 |
| Slave 2 | 21 | 14 | 35 | 60 | 6 | 12 | 6 | 95 | 1 | 1 | 1648 | 199 |
| Slave 3 | 20 | 15 | 35 | 57 | 8 | 15 | 7 | 117 | 1 | 0 | 2874 | 193 |
| Slave 4 | 25 | 10 | 35 | 71 | | | 0 | 0 | 0 | 0 | 1962 | 261 |
| Slave 5 | 23 | 12 | 35 | 65 | | | 0 | 0 | 0 | 0 | 1635 | 251 |
| Slave 6 | 21 | 14 | 35 | 60 | | | 0 | 0 | 0 | 0 | 2511 | 241 |
| Slave 7 | 18 | 17 | 35 | 51 | | | 0 | 0 | 0 | 1 | 1766 | 192 |
| Slave 8 | 14 | 21 | 35 | 40 | 14 | | 3 | 1 | 1 | 1 | 1730 | 122 |
| Slave 9 | 13 | 21 | 34 | 38 | 14 | | 4 | 19 | 1 | 1 | 1268 | 97 |
| Slave 10 | 12 | 22 | 34 | 35 | 14 | 20 | 6 | 76 | 1 | 1 | 863 | 82 |
| Slave 11 | 10 | 24 | 34 | 29 | 12 | 22 | 10 | 112 | 1 | 0 | 1528 | 59 |
| Slave 12 | 21 | 13 | 34 | 61 | | | 0 | 0 | 0 | 0 | 2619 | 228 |
| Slave 13 | 22 | 13 | 35 | 62 | | | 0 | 0 | 0 | 0 | 3195 | 251 |
| Slave 14 | 28 | 6 | 34 | 82 | | | 0 | 0 | 0 | 0 | 2623 | 276 |
| Slave 15 | 28 | 5 | 33 | 84 | | | 0 | 0 | 0 | 1 | 2475 | 300 |
| Total | 327 | 227 | 554 | | | | | 452 | 8 | 8 | 32660 | 3233 |
| Average | 20 | 14 | 34 | 58 | 22 | 30 | 2 | 28 | 0 | 0 | 2041 | 202 |

**Note**: If the condition for "> 0" or "> 5" never occurred, the no number is listed and the maximum possible value contributes to the average.

Graph No. 13 exhibits a "good behaviour". As the column % shows, the average workload is about 88%. No node had to wait for more than 26 seconds in a row. Nodes 7 and 9—15 ran out of work for the first time after more than 1000 seconds. Most nodes had to wait more than 5 seconds only close to the end of the whole computation. Since the graph has no solution, every iteration of a single processor search was also performed by one of the processors in the 16-node ring. So the speedup shown in Table 2 for graph No. 13 is about 14 = 88% of 16.

Graph No. 44 produces an average work load of only 58%. This results mainly from the unfavourable relation between communication and execution times. Shortly after the initial distribution phase, slave No. 13 found a solution.

The work load of graph No. 47 is excellent: 99%. This stems from the non-existing communication. Aside from the initial distribution (which took at most 15 seconds) the

Table 5. Statistical details for graph No. 47, solved with 16 processors.

Solution (found by slave No. 1): There is a Hamiltonian circuit:

| | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 57 | 42 | 38 | 88 | 86 | 89 | 10 | 54 | 6 | 58 | 45 | 55 | 1 | 29 | 76 | 81 | 24 | 64 | 80 | 9 |
| 82 | 37 | 72 | 31 | 50 | 13 | 19 | 77 | 34 | 85 | 98 | 67 | 59 | 33 | 60 | 52 | 83 | 92 | 26 | 51 |
| 66 | 97 | 17 | 75 | 27 | 3 | 5 | 100 | 56 | 39 | 25 | 8 | 99 | 23 | 94 | 53 | 91 | 47 | 11 | 20 |
| 61 | 69 | 21 | 63 | 84 | 79 | 15 | 78 | 14 | 73 | 22 | 74 | 18 | 36 | 40 | 12 | 49 | 2 | 46 | 96 |
| 35 | 4 | 30 | 7 | 70 | 44 | 32 | 93 | 68 | 90 | 16 | 95 | 41 | 43 | 65 | 71 | 87 | 28 | 62 | 48 |

| | CPU | TRANS | TOTAL | % | >0 | >5 | WAIT | REQ | IN | OUT | STACK | ITERAT |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Master | 2928 | 5 | 2933 | 99 | | | 0 | 0 | 0 | 0 | 3037 | 32711 |
| Slave 1 | 2927 | 6 | 2933 | 99 | | | 0 | 0 | 0 | 0 | 3652 | 31585 |
| Slave 2 | 2927 | 6 | 2933 | 99 | | | 0 | 0 | 0 | 0 | 3321 | 38064 |
| Slave 3 | 2925 | 7 | 2932 | 99 | | | 0 | 0 | 0 | 0 | 3832 | 35885 |
| Slave 4 | 2923 | 9 | 2932 | 99 | | | 0 | 0 | 0 | 0 | 3272 | 27128 |
| Slave 5 | 2921 | 11 | 2932 | 99 | | | 0 | 0 | 0 | 0 | 4369 | 27088 |
| Slave 6 | 2919 | 12 | 2931 | 99 | | | 0 | 0 | 0 | 0 | 3953 | 30624 |
| Slave 7 | 2917 | 14 | 2931 | 99 | | | 0 | 0 | 0 | 0 | 4320 | 30816 |
| Slave 8 | 2917 | 15 | 2932 | 99 | | | 0 | 0 | 0 | 0 | 3597 | 29079 |
| Slave 9 | 2918 | 13 | 2931 | 99 | | | 0 | 0 | 0 | 0 | 3750 | 27912 |
| Slave 10 | 2918 | 13 | 2931 | 99 | | | 0 | 0 | 0 | 0 | 4197 | 29898 |
| Slave 11 | 2919 | 12 | 2931 | 99 | | | 0 | 0 | 0 | 0 | 4333 | 29145 |
| Slave 12 | 2921 | 10 | 2931 | 99 | | | 0 | 0 | 0 | 0 | 4830 | 26649 |
| Slave 13 | 2923 | 8 | 2931 | 99 | | | 0 | 0 | 0 | 0 | 5629 | 26519 |
| Slave 14 | 2924 | 6 | 2930 | 99 | | | 0 | 0 | 0 | 0 | 4828 | 27466 |
| Slave 15 | 2926 | 5 | 2931 | 99 | | | 0 | 0 | 0 | 0 | 5206 | 29126 |
| Total | 46753 | 152 | 46905 | | | | | 0 | 0 | 0 | 66126 | 479695 |
| Average | 2922 | 9 | 2931 | 99 | 2931 | 2931 | 0 | 0 | 0 | 0 | 4132 | 29980 |

processors did not need to exchange problems. After about 50 minutes, slave No. 1 found a solution, causing only a speedup of 2.5, because the single processor finished after 2 hours.

## 8. CONCLUSION

We have presented an implementation of a parallel backtracking strategy for a set of personal computers. This strategy was tuned to the specific requirement of the hardware environment: no global memory, restricted routing (ring), slow transmission. Our experimental results show that suitable software can cope with these handicaps and produce astonishingly high speedup.

With the more and more intense use of personal computers LANs (local area networks) become commercially available. Via a common bus they allow clique-like connections. It is the goal of our next project to use communication routines of such an LAN

to handle requests for subproblems more effectively: the donator can be any network member and the transmission is much faster (1 Megabit/sec).

**Acknowledgement.** Thanks to E. SPECKENMEYER for inspiring discussions and to R. FUNKE and M. UTERMÖHLE for technical support.

## REFERENCES

[1] COOK, S. A.: The complexity of theorem-proving procedures. In: Proc. 3rd Ann. ACM Symposium on Theory of Computing, New York 1971, pp. 151—158.
[2] FINKEL, R.—MANBER, U.: DIB — a distributed implementation of backtracking. Computer Science Technical Report No. 583, University of Wisconsin, Madison 1983, pp. 1—7.
[3] GAREY, M.—JOHNSON, D.: Computers and Intractability. A Guide to the Theory of NP--Completeness. Freeman and Company, San Francisco 1979, 338 pp.
[4] KARP, R. M.: Reducibility among combinatorial problems. In: Complexity of Computer Computations. Plenum Press, New York 1972, pp. 85—103.
[5] LAI, T.-H.—SAHNI, S.: Anomalities in parallel branch-and-bound algorithms. Communications of the ACM, Vol. 27, 1984, No. 6, pp. 594—602.
[6] MOHAN, J.: A Study in Parallel Computation: the Travelling Salesman Problem. Technical Report CMU-CS-82-136 (R), Dept. of Computer Science, Carnegie-Mellon University, 1983, 21 pp.
[7] MONIEN, B.—SPECKENMEYER, E.—VORNBERGER, O.: Superlinear Speedup for Parallel Backtracking. Technical Report No. 30, Dept. of Computer Science, University of Paderborn, 1986, 22 pp.
[8] PÓSA, L.: Hamiltonian circuits in random graphs. Discrete Math., Vol. 14, 1976, pp. 359—364.
[9] VORNBERGER, O.: Implementing Branch-and-Bound in a Ring of Processors. Technical Report No. 29, Dept. of Computer Science, University of Paderborn, 1986, 16 pp.
[10] WAH, B. W.—EVA MA, Y. W.: MANIP — a multicomputer architecture for solving combinatorial extremum-search problems. IEEE Transactions on Computers, Vol. C-33, 1984, No. 5, pp. 377—390.

Received April 22, 1986 — revised version July 23, 1986