https://doi.org/10.26552/com.C.2009.3.15-19



Karol Grondzak – Penka Martincova *

# PARALLEL BACKTRACKING ALGORITHM FOR HAMILTONIAN PATH SEARCH

*The speed of calculations is a common problem to tackle in many areas of scientific research and real life. This paper presents an implementation of a parallel backtracking algorithm. The performance of the proposed algorithm is demonstrated on the problem of Hamiltonian Path search. Obtained results exhibit significant improvement of the parallel algorithm over the sequential one. Different aspects of parallelization of backtracking algorithm are studied and presented.*

## 1. Introduction

Recently, there are still many scientific problems unsolved. One of the areas with many challenging problems is graph theory. Challenge lies in the speed of the used algorithms. Many of them are proven to be NP-complete. For practical usage any possibility of speed-up is appreciated [1], [2].

A lot of problems solved by graph theory can be characterized as combinatorial problems. To solve them, combinatorial search algorithm is quite commonly applied. It can be described as a process of searching a finite mathematical structure (usually set) for a solution satisfying given criteria [3].

To define combinatorial search algorithms, let us consider a discrete set $X$, a function $F: X \rightarrow \Re$ , and a set of feasible solutions $S$, where $S \subseteq X$. The feasible solution is defined in terms of given constraints specific for a particular problem. Generally combinatorial search algorithms can be divided into two sets.

The first is the set of optimization algorithms [4]. The goal of these algorithms is to find the feasible solution $x \in S$ such that its value of function $F$ is extreme. To accomplish this task, the set $S$ must be constructed, evaluating all the elements of the set $X$ and checking given constrains. Simultaneously while constructing the set $S$, values of the function $F$ are calculated for all its elements. Then the element with an extreme (e.g. either minimal or maximal, depending on the character of the problem) value of the function $F$ is the solution of the problem.

The second is the set of solution finding problems [4]. The task is again to iterate through the elements of set $X$, but in this case the goal is to find at least one element of the set $S$. In other words, we are looking for an element $x \in X \wedge x \in S$.

For many practical problems the set $X$ is large, which leads to long execution times of programs solving these problems. To reduce the execution times, many approaches were proposed and implemented. Recently parallelization of the task is a common technique to be applied [5].

## 2. Distributed Backtracking algorithm

While designing the algorithm to solve a combinatorial search problem, a backtracking approach can be used [6]. It is based on the sequential construction of the elements of the set $X$ and evaluation of their feasibility. During this process the elements which are identified as not to produce a feasible solution are pruned. This can significantly reduce the computational time.

To formalize the backtracking approach, let us consider $n$ sets:

$$E_k = \left\{e_1^k, e_2^k, \ldots, e_{m_k}^k\right\}, k = 1 \ldots n\,, \tag{1}$$

where $m_k$ is a number of elements of the set $E_k$ and $e_i^k$ is i-th element of the set $E_k$. Then the above mentioned set $X$ is defined as Cartesian product of the sets $E_i$, $\forall i$:

$$X = E_1 \times E_2 \times \cdots \times E_n\,, \tag{2}$$

with cardinality $Q = \prod_{i=1}^{n} m_i$. It is obvious that the set $X$ consist of $n$-tuples:

$$\left(x_1^c, x_2^c, \ldots, x_n^c\right), c = 1, 2, \ldots Q\,, \tag{3}$$

such that $x_j^c \in E_i$. These $n$-tuples are constructed recursively extending the set of $(n-1)$-tuples.

* **Karol Grondzak, Penka Martincova**
Department of Informatics, Faculty of Management Science and Informatics, University of Zilina, Slovakia, E-mail: Karol.Grondzak@fri.uniza.sk

The backtracking algorithm starts with an empty $n$-tuple. In the stage i the $(i-1)$-tuple is extended using elements of the set $E_i$. Newly obtained $i$-tuples are then checked for feasibility and then expanded to $(i+1)$-tuples to form the set $E_i+1$.

This process is actually a depth-first search of a search-tree. Nodes of the search tree at i-th level consist of $i$-tuples. An example of the search-tree is in Fig. 1.

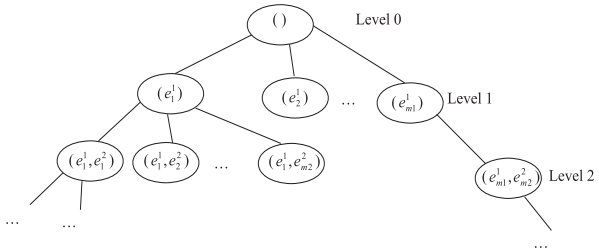


*Fig. 1 Example of a search tree*

It is obvious, that just described the backtracking algorithm is suitable for distributed processing. One strategy to distribute tasks among processors could be as follows. Let us construct all $i$–tuples for some small value of $i$, chosen to correspond the amount of available processors. Then assign those $i$-tuples to processors, such that each processor will perform depth-first search of a sub-tree of the search tree starting from a given $i$-tuple.

The properties of the search-tree and the strategy of search can significantly influence the performance of the backtracking algorithm [7]. When constructing the backtracking algorithm, we do not have usually any a priori information about the structure of the search-tree.

In any stage of the backtracking algorithm, we perform depth-first search of some sub-tree. If there was a solution found in the searched sub-tree, the algorithm finishes. If not, then the backtracking algorithm has to choose another sub-tree to search.

To formalize the properties of a search-tree from the point of view of the backtracking algorithm, let us denote depth of the sub-tree:

$$D_i^k = \sum_{l=1}^{m_k} D_i^{k+1}, k = 1, 2, \ldots n; i = 1, 2, \ldots m_k, \qquad (4)$$

where $n$ is the number of sets used to construct $n$-tuples and $m_k$ is the number of elements in the set $k$. Depth of a sub-tree is a function of search strategy and represents a number of $n$-tuples to construct and process during depth-first search of that sub-tree.

## 3. Hamiltonian Path and Circle

One of the well-known problems of graph theory is the problem of finding a path on a graph which visits each of the nodes exactly once. If the starting and final nodes are different, the problem is known as the Hamiltonian Path problem [6]. Special case when starting and final nodes are identical is known as the Hamiltonian Circle problem.

It is known that both the Hamiltonian Path and Hamiltonian Circle are NP-complete problems. We can characterize them as decision problems – for a given graph the goal is to determine if the Hamiltonian Path or Circle exists.

Among many problems of the Hamiltonian Circle problem let us mention a well-known Knight's Tour problem. The goal is to find a path of knight on a chessboard of a standard dimension $N = 8$, such that will visit all the squares, each exactly once. This problem can be generalized to chessboards of any dimension.

```
WHILE workers not finished
DO
  wait for a message from worker
  IF solution found
    notify all workers
    finish
  ELSE
  IF request for a job
    send job to worker
END
```

(a)

```
REPEAT
  ask for a job from master
  IF there is a job
    process assigned job
    IF solution found
      notify master
UNTIL there is a job
```

(b)

*Fig. 2 Pseudo-code of master (a) and worker (b) processes*

### 3.1 Distributed Hamiltonian Path Search Algorithm

As mentioned above, the Hamiltonian Path problem is a decision problem. The backtracking combinatorial search algorithm can be applied to solve it. The problem is, that even for a relatively small dimension of the graph, the size of the $n$-tuples to be searched is huge. Let us consider the problem of Knight's Tour problem for a chessboard of 64 squares. First let us number the squares of the chessboard starting form 1 to 64. Our goal is to construct a $n$-tuple containing a number of squares visited during the tour. On each of the squares, the knight has at most eight possible ways to move. So there are $8^{64}$ tuples to be searched. This is a rough estimation and many of these tuples can be pruned during the depth-first search of the search-tree. But still there are many of tuples to be checked. There is too much work for a single processor.

To improve the performance of the backtracking algorithm for search of the Hamiltonian path, several processors can be involved. We can expect linear increase of the performance, but for some situations even super-linear increase was reported [7]. It depends on the properties of the search-tree of the solved problem.

Parallelization is quite straightforward, because of the nature of the search-tree. On each level of the search tree, there are nodes to be searched. Let us denote a number of nodes on each level as

$$Q_i = \prod_{k=1}^{i} m_k, \qquad (5)$$

where $m_k$ is the number of elements of the set $E_k$. The nodes form a set of disjunctive sub-trees. These sub-trees can be assigned to at most $Q_i$ processors to perform parallel search on them.

For a given amount of P processors it is reasonable to determine a level of the tree to start parallel search such that:

$$Q_{i-1} < P <= Q_i . \quad (6)$$

Then the maximum amount of processors is involved in search. In fact, when $P = Q_i$, all the processors will search exactly one sub-tree. In other cases some of the processors will search several sub-trees. This situation can lead to better load-balancing comparing the case when $P = Q_i$, because usually the sub-trees are of different depth. Then if all the sub-trees are assigned at the beginning of the algorithm, those processors which finish their task are sitting idle waiting for those processors, whose sub-trees are deeper and require more time to finish the search.

## 4. Experiment and obtained results

To test the proposed distributed algorithm, we have modeled the following routing problem. Let us consider a regular rectangular mesh of cells of the dimension $N$. In this mesh, we have to find a route from a given start point to a given finish point such that each of the cells will be visited exactly once. This problem can be described in terms of graph theory as a problem of finding a Hamiltonian path in a graph of a special regular form (Fig. 3).

This problem can represent a task to route some small maintenance vehicle (e.g. robot) on a set of office cubicles to perform daily routine tasks (e.g. cleaning, etc.). It can also model the route of some machine in a factory to perform a prescribed operation in different parts of some product, e.g. to drill holes, etc. Once the route is found, it can be embedded into a device, which will then use this fixed route regularly. Or in case when the layout is subject to change, the algorithm can be implemented in the device. When the situation changes, the device will apply the algorithm to find a new route corresponding the actual situation.

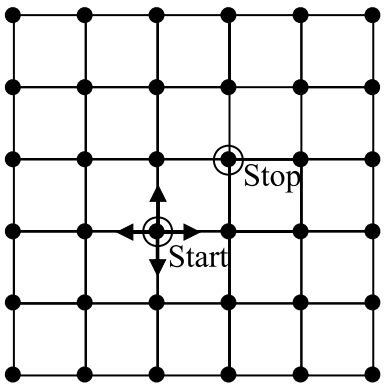


*Fig. 3 Example of graph representation of solved problem*

The algorithm mentioned in section 3, was implemented using OPENMPI Project [8], [9]. It is freely-available, high performance implementation of Message Passing Interface (MPI) standard. MPI is a standard for communication and synchronization of distributed entities. It was proposed by a consortium of companies for high performance on both massively parallel machines and on workstation clusters.

The presented results were obtained on a commodity workstation cluster. The cluster consists of twenty personal computers equipped with a processor Intel Core 2 Duo and 1024MB of RAM memory. Computers are interconnected by 100Mb/s local area network. The connection is sufficient because of the small communication requirements of the application.

The performance was studied for problems of the dimension $N = 6$. The obtained results are summarized in the form of tables. Each table presents data for different study. Data in each table represent amount of iterations necessary to find a solution. This representation was preferred to the time measurement because it represents the performance of the algorithm and is not influenced by the actual load of processors. It corresponds to the depth of the search tree when considering the starting level 0, as it was defined above (4).

Three different characteristics of the problem were studied. First, we studied the influence of the chosen depth-first search strategy to the performance of the algorithm. When solving backtracking problems, the strategy of the next step choice is embedded in the code. It is obvious that for the solved problem there are at most four different directions to continue from a given position. Let us denote those directions L(eft), R(ight), U(p) and D(own). So there are 24 different strategies to choose from, considering a different order of directions (e.g. LURD is one strategy, ULDR is another). Because of some symmetry of the solved problem, the obtained results are same for couples of strategies. This is the reason, why there are only 12 results presented, other 12 results are the same (row Direction in Table 1). When comparing results for different strategies it can be seen that the choice of strategy has significant impact on the performance (Table 1, Table 2 Table 3). For some strategies the number of iterations is significantly smaller than for other ones. There is no a-priori information about the behavior of the strategies, so usually when designing the backtracking algorithm, we randomly choose one of the available strategies.

Secondly, we studied the performance improvement with respect to the number of processors involved in calculation. It can be seen (Fig. 4) that the obtained results are highly non-linear. This figure shows the number of iterations needed to find the solution of the problem. The smaller number of iterations means the better performance. The results are presented for strategy number 4 (DULR). It indicates that the search tree is unbalanced. It is also indicating that a better load-balancing strategy should be applied to distribute load among the nodes.

Third, the study assessed the influence of the depth at which the search starts. It can be seen that the number of iterations needed

Number of iterations and relative speed-up for different strategies for search started in level 1. Table 1
Number of processors involved for parallel search is 5

| Strategy | 0 | 1 | 2 | 3 | 4 | 5 | 12 | 13 | 14 | 15 | 16 | 17 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Direction | DLRU | DLUR | DRLU | DRUL | DURL | DULR | RLDU | RLUD | RDLU | RDUL | RUDL | RULD |
| Sequential | 240 | 821958 | 9638 | 4353 | 101981 | 2515876 | 908134 | 138265 | 264258 | 8476 | 204707 | 6597 |
| Parallel | 240 | 204 | 2018 | 3486 | 19481 | 915 | 24285 | 138265 | 24321 | 858 | 81255 | 6597 |
| Speedup | 1 | 4029.2 | 4.8 | 1.2 | 5.2 | 2750.0 | 37.4 | 1.0 | 10.9 | 9.9 | 2.5 | 1.0 |

Number of iterations and relative speed-up for different strategies for search started in level 2. Table 2
Number of processors involved for parallel search is 17

| Strategy | 0 | 1 | 2 | 3 | 4 | 5 | 12 | 13 | 14 | 15 | 16 | 17 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Sequential | 240 | 821958 | 9638 | 4353 | 101981 | 2515876 | 908134 | 138265 | 264258 | 8476 | 204707 | 6597 |
| Parallel | 240 | 204 | 2018 | 3486 | 19481 | 915 | 1136 | 457 | 1108 | 616 | 559 | 457 |
| Speedup | 1 | 4029.2 | 4.8 | 1.2 | 5.2 | 2750 | 799.4 | 302.5 | 238.5 | 13.8 | 366.2 | 14.4 |

Number of iterations and relative speed-up for different strategies for search started in level 3. Table 3
Number of processors involved for parallel search is 38

| Strategy | 0 | 1 | 2 | 3 | 4 | 5 | 12 | 13 | 14 | 15 | 16 | 17 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Sequential | 240 | 821958 | 9638 | 4353 | 101981 | 2515876 | 908134 | 138265 | 264258 | 8476 | 204707 | 6597 |
| Parallel | 240 | 204 | 3488 | 1350 | 409 | 915 | 1136 | 457 | 1108 | 858 | 559 | 457 |
| Speedup | 1 | 4029.2 | 2.8 | 3.2 | 249.3 | 2750 | 799.4 | 302.5 | 238.5 | 9.9 | 366.2 | 14.4 |

to find a solution is either the same (strategies 0, 1 and 5 ) for all the starting depths, or is improving with the increasing starting depth (strategies 3, 4, 12, 13, 14,16 and 17). For the rest of strategies (2 and 15) the number of iterations to find solution has increased for level 3. It could be caused by the fact, that only 38 processors were involved in the calculation instead of 64, which is the number of sub-trees in level 3. When using at least 64 processors, we would expect to get either the same or better results for strategies 2 and 15.

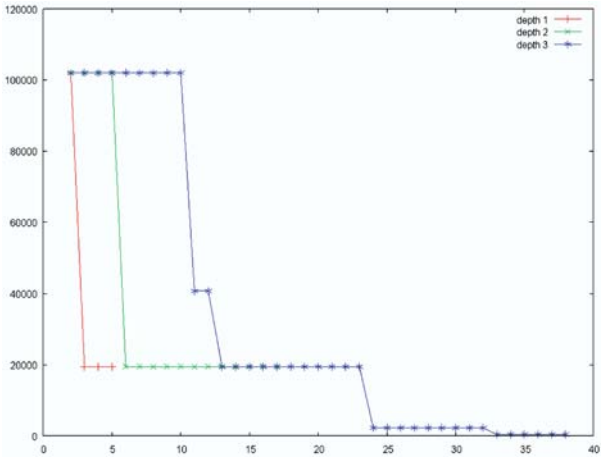


*Fig. 4 Number of processors versus number of iterations needed to find solution.*

Fig. 4 demonstrates also the influence of the depth at which the search starts to the performance of the algorithm. This graph presents the dependence of the number of iterations needed to find a solution with respect to the number of processors for different starting depths. It demonstrates that when the start depth is increasing, the number of iterations decreases (depth 1 curve versus depth 3 curve).

The smaller the number of iterations, the better performance of the algorithm

## 5. Conclusion and Future Work

Parallel paradigm is recently a very popular approach for solving time-consuming scientific problems. It is widely adopted in many scientific areas starting from scientific calculations, through modeling [10] up to computational biology [11].

In this paper, the study of the parallelization of the backtracking algorithm was presented. The general parallel backtracking algorithm was proposed and implemented using MPI implementation OPENMPI. It was tested on the problem of Hamiltonian path search.

The obtained results are in accordance with our expectation [12]. This study helped the authors to better understand the properties of the parallel backtracking algorithm.

We can expect that the performance can be improved by involving more processors into calculation. The obtained results also indicate that the better load balance is achieved when the search starts deeper in the search tree.

The future work will be to modify the algorithm for grid environment. The potential for improving the performance of algorithm is in grid technologies. Study of different load-balancing techniques to the performance of the backtracking algorithm will also be performed.

Acknowledgement: The authors would like to thank Dr. Michal Kaukic for the opportunity to use the laboratory of parallel applications. This work was partially supported by VEGA Grant No. 1/0761/08 "Design of Microwave Methods for Materials Nondestructive Testing" and VEGA Grant No. 1/0796/08 "Large Data Modeling and Processing".

## References

[1] GENDRON, B., CRAINIC, T. G.: *Parallel Branch and Bound Algorithms: Survey and Synthesis,* Operations Research, 42, pp. 1042–1066, 1994.
[2] CRAINIC, T. G., LE CUN, B., ROUCAIROL, C.: *Parallel Branch-and-Bound Algorithms,* Wiley Interscience, October 2006, ch. 1.
[3] MEZMAZ, M., MELAB, N., TALBI, E-G.: *A Grid-enabled Branch and Bound Algorithm for Solving Challenging Combinatorial Optimization Problems,* In Proc. of 21st IEEE Intl. Parallel and Distributed Processing Symp., Long Beach, California, 2007.
[4] QUINN, M. J.: *Parallel Programming in C with MPI and OpenMP.* Mc Graw Hill, 2003. ISBN 007-123265-6.
[5] WILKINSON, B., ALLEN, M.: *Parallel Programming Second Edition.* Pearson Education, 2005. ISBN: 0-13-140563-2.
[6] KUCERA, L.: *Combinatorial algorithms (in Slovak)*, SNTL, 1989.
[7] NAGESHWARA RAO, V., KUMAR, V.: *On the Efficiency of Parallel Backtracking,* IEEE Trans. Parallel Distrib. Syst. 4(4): 427–437 (1993).
[8] http://www.open-mpi.org/
[9] GROPP, W., LUSK, E., SKJELLUM, A.: *Using MPI,* Second Edition. The MIT Press, 1999. ISBN: 978-0-262-57134-0.
[10] KVASNICA, I., KVASNICA, P., IGAZOVA, M.: *Parallel Modeling in Computer Systems,* Acta Avionica, Vol. X, 2008, 16. ISSN 1335-9479, pp. 8–86.
[11] SCHMIDT, M. C., SAMATOVA, N. F., THOMAS K, PARK, B. H.: *A scalable, parallel algorithm for maximal clique enumeration,* J. Parallel Distrib. Comput. 69 (2009), pp. 417–428.
[12] GRONDZAK, K., MARTINCOVA, P., CHOCHLIK, M.: *Performance Analysis of Parallel Algorithm for Backtracking,* Proc. of 4th International Workshop on Grid Computing for Complex Problems, Bratislava, 2008.