ROMANIAN JOURNAL OF INFORMATION SCIENCE AND TECHNOLOGY
Volume **13**, Number 3, 2010, 241–254

An implementation of the backtracking algorithm for multicore systems

Claudiu-Florin GIURGIU

National College for Informatics *Traian Lalescu*
23 Victoriei Str., Hunedoara, Romania

E-mail: claudiu.giurgiu@yahoo.com

Abstract. The emergence of inexpensive parallel computers powered by multicore chips combined with high clock rates raises new challenges for software engineering. Since the performance improvements will not come from increased clock rates, high performance applications will need to be parallelized. In addition to operating system support, adjustments to existing software are required to maximize utilization of the computing resources provided by multicore processors. The ability of multicore processors to increase application performance depends on the use of multiple threads within applications. We will describe an algorithm that allows backtracking problems to be solved by means of such multicore systems. A multicore algorithm meant for backtracking problems and able to meet certain constraints is made of multiple search threads with concurrent operation and distinct processing parts of the solution space. The proposed algorithm is scalable.

1. Introduction

In last years, the processor design industry has shifted away from increasing the performance of monolithic, centralized processor designs. In the past, processor generations could scale performance by increasing clock frequency and designing more complex structures. Thermal issues and increased power dissipation have become the main design constraints, forcing a change to decentralized multicore designs. On june 1, 2009, AMD announced availability of the worlds first six-core server processor with Direct Connect Architecture for two-, four- and eight-socket servers[1]. Intel also have six-core processor[2], so the race continues. Multicore processors lessen issues by using multiple simpler cores and tightly integrating them together on a single die. This allows for an increase in throughput capabilities of the processor but does

not necessarily increase performance. The ability of multicore processors to increase application performance depends on the use of multiple threads.

A wide number of algorithms have been created and there are various debates whether one or another algorithm is recommended for the solving of distributed constraint problems (DisCSPs). The synchronous algorithms can exchange updated information with a reduced degree of parallelism. The asynchronous algorithms use less updated information, with a higher degree of parallelism. The hybrid algorithms combine the two versions. A comparison between such algorithms can be found in [3], [4].

In the last twenty years, the interest in the agent-based distributed solving of problems has increased. The different parts of a problem are kept by different agents that have an independent behavior and collaborate among themselves in order to reach a global solution. World Wide Web offers possibilities of solving real problems, using agents. There are also approaches of distributed solving of problems without agents. Algorithms capable of meeting the distributed constraints have been suggested by different authors. They suggested algorithms for asynchronous backtracking (ABT) as well as for asynchronous weak-commitment backtracking search (WCS) [5][6][7]. These algorithms require a total ordering of the agents, static for the ABT and dynamic for WCS. The alternatives have been studied in [8]. When a dead-end has been reached, messages are generated and exchanged among the agents and these messages need to be stored. This involves a consistent transmission of messages and supplementary memory to store them. Different approaches are represented by the distributed backtracking algorithm (DIBT) [9] and by the interleaved distributed intelligent backtracking (IDIBT)[10]. The agents are ordered hierarchically and according to some identifiers. Another algorithm was also presented, under the name of asynchronous aggregation search (AAS) [11]. Further details can be found in [12].

The performance obtained using agents for n-queen problem have been studied in [13] using a Sun Fire T2000 with 8 cores. Tests showed a performance speedup of 4.57 to 4.71. The speedups were obtained with respect to the execution time of the sequential version of the benchmarks. A program-level partitioning of the memory operations is analyzed in [14] to divide the memory accesses across the data caches. A parallelization of some problems demonstrates that significant algorithm engineering may be required to achieve speedup and that the available parallelism can be limited by the problem itself, not the number of available cores or processors. Other problems provides ample parallel work, but the caches and memory accesses become a bottleneck as the number of threads increases. Further details can be found in [15]. A distributed version of algorithm is presented in [16].

The paper is organized as follows. Section 2 describes the sequential backtracking, while Section 3 presents the new multicore backtracking algorithm. Finally some remarks are drawn in the conclusion section.

2. The sequential Backtracking algorithm

In the real world, a situation that arises frequently is that in which the solving of a problem leads to the determination of vectors of the form $x = (x_1, x_2, ..., x_n)$, where

- each component x_i belongs to a certain finite set V_i;
- there are certain relation requirements that have to be met by the components of the vector, called restraints, so that x represents a solution to the problem if and only if these constraints are met by the components $x_1, x_2, ..., x_n$ of the vector.

The problem to be solved now is to find all the vectors of form $x = (x_1, x_2, ..., x_n)$ that meet these constraints.

The Cartesian product $V_1 \times V_2 \times ... \times V_n$ is called the space of possible solutions. The solutions of the problem are those possible solutions that meet the constraints.

One might say that the work speed of modern computers is so high that the exponential time shouldn't bother. In order to clear up this aspect, we acknowledge the fact that computers work at a very high speed, indeed. High values of work time can be reached even for values of n that are relatively small, the explanation being not that computers are not fast enough, but that the exponential function $f(n) = a^n$ with $a > 1$ increases very quickly. For example, for $a = 3$ and $n = 50$, the work time needed covers centuries.

According to the above mentioned elements, it is highly recommended that for a given problem we should elaborate algorithms that are not exponential. The algorithms that are considered to be efficient are those for which the number of operations is polynomial (i.e. it is expressed in the form of a polynomial in n where n is the number of input data). It is not always possible to avoid the exponential algorithms; one example is the problem of generating all the sub-sets of a set having n elements when the number of results is 2^n and therefore the number of operations is bound to be exponential.

The backtracking algorithm aims at avoiding the generation of all possible solutions, thereby cutting short the calculation time.

The components of vector x are assigned values in the ascending order of the indices. This means that x_k is assigned a value only after $x_1, x_2, ..., x_{k-1}$ have been assigned values that do not infringe the constraints. Moreover, the value of x_k has to be chosen such as $x_1, x_2, ..., x_{k-1}$ also meet certain constraints.

Not meeting the constraints expresses the fact that any way we might choose $x_{k+1}, ..., x_n$, we will not obtain a solution (so the constraints ought to be met for a solution to be obtained). As a result, the next step will be to assign a value to x_k only when the constraints have been met. If they haven't, a new value is chosen for x_k or, in case the finite set of values V_k has been used out, a new choice is attempted for the preceding component x_{k-1} of the vector, decreasing k by one unit, etc. This backtracking gives the name of the method, as it expresses the fact that when we cannot advance, we backtrack the current sequence of the solution.

One has to notice that in certain cases, the fact that $x_1, x_2, ..., x_{k-1}$ meets the constraints is not sufficient to guarantee that we will obtain a solution for which the first $k - 1$ solutions coincide with these values.

The choice of constraints is of utmost importance, a good choice leading to a substantial reduction in the number of calculations. In the ideal case, the constraints should be not only necessary, but also sufficient in obtaining a solution. But, usually,

these are the constraints applied to the first k components of the vector. Obviously, the constraints in the case $k = n$ are the very constraints imposed by the statement of the problem.

Through the backtracking method, any solution vector is built progressively, starting with the first component and up to the last one, eventually backtracking some values previously assigned. We remind that $x_1, x_2, ..., x_n$ are assigned values in sets $V_1, V_2, ..., V_n$. By assigning and failed assigning attempts because of not meeting the constraints, certain values are used out. For a certain component x_k we will mark C_k the set of values used out at the current moment. Obviously, $C_k \subset V_k$.

The process of obtaining the solution vectors by means of the backtracking method is easy to implement, due to the fact that any modification in configuration only affects few elements, namely the k index of the component, component x_k and the set C_k of used out values.

The corresponding algorithm is the following:

```
//the sets V_1, V_2, ..., V_n are initialized
//the initial configuration is built
k = 1;C_i = φ ∀i = 1, 2, ..., n
while (k > 0)
{
  //the configuration is not final
  if (k = n + 1) {
                    //the configuration is of a solution type
                    solution();//keep solution x = (x_1, x_2, ..., x_n)
                    k = k − 1; //backtracking after the building of a solution
                  }
  else if C_k ≠ V_k { //there are still values unused
                      choose a value v_k from V_k − C_k
                      C_k = C_k ⋃ {v_k}
                      if (v_1, v_2, ..., v_k meet the constraints) { //assign, advance
                                                                     x_k = v_k;
                                                                     k = k + 1;
                                                                     }
                    }
       else {//backtracking
             C_k = φ;
             k = k − 1;
            }
}
```

The implementation of the algorithm can be simplified if one takes into consideration that the sets of values $V_1, V_2, ..., V_n$ are of the form $V_1 = \{1, 2, ..., s_1\}$, ..., $V_n = \{1, 2, ..., s_n\}$, so each set V_k is made of the first s_k natural numbers, starting with 1. Therefore, set V_k can be represented simply by its number of elements s_k. It is also supposed that the values for each component x_k shall be chosen in an ascending order.

In this situation, each set of used out values C_k has the form $1, 2, ..., v_k$ and as a consequence, it can be represented only by value v_k. As v_k is the number of elements belonging to set C_k, this set is empty if and only if $v_k = 0$.

The elements shown above allow us to replace the sets in the algorithm by numbers; moreover, the initial configuration can be reduced to the values of components $x_1, x_2, ..., x_n$.

The description of the method requires that, at the moment we try to assign a value to component x_k, the following elements should be mentioned:

- the values chosen until that moment for the components $x_1, x_2, ..., x_{k-1}$ are the values $v_1, v_2, ..., v_{k-1}$;
- the sets of values $C_1, C_2, ..., C_{k-1}$ used out for each of the $x_1, x_2, ..., x_{k-1}$ components.

In short, the description can be written

$$\left(\begin{array}{cccc||cccc} v_1 & ... & v_{k-1} & & x_k & x_{k+1} & ... & x_n \\ C_1 & ... & C_{k-1} & & C_k & \phi & ... & \phi \end{array} \right).$$

The significance of the solution for such a configuration is the following:

- components $x_1, x_2, ..., x_{k-1}$ were assigned values $v_1, v_2, ..., v_{k-1}$ and these values meet the constraints;
- an attempt will be made to assign a value to component x_k, the assigned value belonging to set $V_k - C_k$;
- the values used out for $x_1, x_2, ..., x_{k-1}$ are to be found in sets $C_1, C_2, ..., C_{k-1}$, sets that also contain values $v_1, v_2, ..., v_{k-1}$ considered used out;
- for components $x_{k+1}, ..., x_n$ no value has been used out and therefore $C_{k+1} = ... = C_n = \phi$.

In the beginning, no value is used out and the first step will be to attempt at establishing a value for the first component. In this case, the initial configuration is

$$\left(\begin{array}{||cccc} x_1 & x_2 & ... & x_n \\ \phi & \phi & ... & \phi \end{array} \right),$$

in which all C_i, $\forall i = 1, 2, ..., n$ sets are empty. One solution is a configuration of form

$$\left(\begin{array}{cccc||} v_1 & v_2 & ... & v_n \\ C_1 & C_2 & ... & C_n \end{array} \right)$$

with the significance that vector $v_1, v_2, ..., v_n$ is a solution to the problem.

One of the following four possible modifications can be applied to a given configuration:

1. ***Assign a value and advance***. This modification appears when there are still values that have not been used for an element x_k and we choose a value from $V_k - C_k$ so that $v_1, v_2, ..., v_k$ should meet the constraints. In this case the chosen value v_k is added to the set C_k and is considered used, then we advance to the next component x_{k+1}. The operation can be presented as follows:

$$\left(\begin{array}{cccc||cccc} v_1 & ... & v_{k-1} & & x_k & x_{k+1} & ... & x_n \\ C_1 & ... & C_{k-1} & & C_k & \phi & ... & \phi \end{array} \right) \longrightarrow$$

$$\left(\begin{array}{cccc||ccc} v_1 & ... & v_{k-1} & v_k & x_{k+1} & ... & x_n \\ C_1 & ... & C_{k-1} & C_k \bigcup \{v_k\} & \phi & ... & \phi \end{array} \right).$$

2. ***Attempt failed***. This modification appears at the moment when there are still values unused for x_k but we choose a value that makes $v_1, v_2, ..., v_k$ not to meet the constraints. The operation can be presented as follows:

$$\left(\begin{array}{cccc||cccc} v_1 & ... & v_{k-1} & & x_k & x_{k+1} & ... & x_n \\ C_1 & ... & C_{k-1} & & C_k & \phi & ... & \phi \end{array} \right) \longrightarrow$$

$$\left(\begin{array}{ccc||cccc} v_1 & ... & v_{k-1} & x_k & x_{k+1} & ... & x_n \\ C_1 & ... & C_{k-1} & C_k \bigcup \{v_k\} & \phi & ... & \phi \end{array} \right).$$

3. ***Backtracking after failed attempt***. This type of modification appears when all the values for x_k have been used, that is $C_k = V_k$. In this case, we come back to component x_{k-1} and try to assign it a new value. The value chosen will be from the values left unused for x_{k-1}. On backtracking, the set of used out values for x_k will be considered empty. The operation can be presented as follows:

$$\left(\begin{array}{cccc||cccc} v_1 & ... & v_{k-1} & & x_k & x_{k+1} & ... & x_n \\ C_1 & ... & C_{k-1} & & C_k & \phi & ... & \phi \end{array} \right) \longrightarrow$$

$$\left(\begin{array}{ccc||cccc} v_1 & ... & v_{k-2} & x_{k-1} & x_k & ... & x_n \\ C_1 & ... & C_{k-2} & C_{k-1} & \phi & ... & \phi \end{array} \right).$$

4. ***Backtracking after having built a solution***. This modification takes place when all the components of the vector were assigned values that meet the constraints in the problem statement. In this case, a solution was built and, after having kept it, we returned to the attempt at finding another possible unused value for the last x_n component. The modification can be represented as follows:

$$\left(\begin{array}{cccc||} v_1 & v_2 & ... & v_n \\ C_1 & C_2 & ... & C_n \end{array} \right) \longrightarrow \left(\begin{array}{ccc||c} v_1 & ... & v_{n-1} & x_n \\ C_1 & ... & C_{n-1} & C_n \end{array} \right).$$

From this configuration also results that checking if one solution was found can come down to $k = n + 1$.

An important aspect is that of bringing the process of searching for all solutions to an end. Through modifications repeated by one of the four types introduced

previously, one cannot reach twice the same configuration and therefore, at a given moment one reaches configuration

$$\left(\begin{array}{c||cccc} & x_1 & x_2 & ... & x_n \\ & V_1 & \phi & ... & \phi \end{array}\right).$$

In this case a backtracking should take place as all the possible values for x_1 have been used out. In this case we pass on to a configuration where $k = 0$. This checking is used in practice in order to determine the process of solution building.

3. The multicore backtracking algorithm

We propose to consider a threshold t for the index k of the components for which the solutions search will continue by a separate thread. The main thread follows the bellow mentioned steps:

```
//the sets V_1, V_2, ..., V_t are initialized
//the initial configuration is built
k = 1; C_i = φ ∀i = 1, 2, ..., t
while (k > 0)
{
  //the configuration is not final
  if (k = t + 1) {
                    //the configuration has the first t elements established
                    it generated a separate thread with the initial
                    configuration x = (x_1, x_2, ..., x_t)
                    k = k − 1; //backtracking after having built
                    // a separate thread
                  }
  else if C_k ≠ V_k { //there are still values unused
                      choose a value v_k from V_k − C_k
                      C_k = C_k ⋃ {v_k}
                      if (v_1, v_2, ..., v_k meet the constraints) { //assign, advance
                                                                     x_k = v_k;
                                                                     k = k + 1;
                                                                     }
                    }
       else {//backtracking
             C_k = φ;
             k = k − 1;
            }
} //solutions are collected from the secondary threads
```

Unlike the standard algorithm for the backtracking method, in a multicore system, positions $x_1, x_2, ..., x_t$ of vector x are determined by the main thread and the remaining positions are determined by the secondary threads. The main thread is

different from the standard variant. When the elements $x_1, x_2, ..., x_t$ were determined the continuation is achieved by a secondary thread and the main thread operates a backtracking after having found one solution. The operations can be presented as follows:

$$\left(\begin{array}{cccc||cccc} v_1 & ... & v_{t-1} & & x_t & x_{t+1} & ... & x_n \\ C_1 & ... & C_{t-1} & & C_t & \phi & ... & \phi \end{array} \right) \longrightarrow$$

$$\left(\begin{array}{cccc||ccc} v_1 & ... & v_{t-1} & v_t & x_{t+1} & ... & x_n \\ C_1 & ... & C_{t-1} & C_t \bigcup \{v_t\} & \phi & ... & \phi \end{array} \right)$$

and then

$$\left(\begin{array}{cccc||ccc} v_1 & ... & v_{t-1} & v_t & x_{t+1} & ... & x_n \\ C_1 & ... & C_{t-1} & C_t \bigcup \{v_t\} & \phi & ... & \phi \end{array} \right) \longrightarrow$$

$$\left(\begin{array}{ccc||cccc} v_1 & ... & v_{t-1} & x_t & x_{t+1} & ... & x_n \\ C_1 & ... & C_{t-1} & C_t \bigcup \{v_t\} & \phi & ... & \phi \end{array} \right).$$

The secondary threads continue the search for solutions from position $t+1$ of the vector, considering the first t elements determined by the main thread. The secondary thread is based on the following algorithm:

//the sets $V_{t+1}, ..., V_n$ are initialized
//the initial configuration is built
$k = t + 1; C_i = \phi \ \forall i = t + 1, ..., n$
//Elements $x_1, x_2, ..., x_t$ are received from the main thread
while $(k > t)$
{
  //the configuration is not final
  if $(k = n + 1)$ {
                    //the configuration is of a solution type
                    solution();//keep solution $x = (x_1, x_2, ..., x_n)$
                    $k = k - 1$; //backtracking after the building of a solution
                   }
  else if $C_k \neq V_k$ { //there are still values unused
                    choose a value v_k from $V_k - C_k$
                    $C_k = C_k \bigcup \{v_k\}$
                    if ($v_1, v_2, ..., v_k$ meet the constraints) { //assign, advance
                                                                     $x_k = v_k$;
                                                                     $k = k + 1$;
                                                                     }
                   }
       else {//backtracking
             $C_k = \phi$;
             $k = k - 1$;
            }
} //send solutions to the main thread

Unlike the standard method, through repeated alterations made by one of the four types previously presented, the same configuration can never be arrived at twice, so, at a certain moment configuration

$$\left(\begin{array}{cccc||ccc} v_1 & ... & v_{t-1} & v_t & x_{t+1} & ... & x_n \\ C_1 & ... & C_{t-1} & C_t & V_{t+1} & ... & \phi \end{array} \right)$$

is reached. In this case a backtracking should occur, because all the possible values for x_{t+1} were used out. Under the circumstances, we pass to a configuration where $k = t$. This type of verification is used in practice in order to check the conclusion of the process of solution building. At the moment the secondary thread reaches a configuration where $k = t$, the solutions that have been found have to be transferred over to the main thread.

For $t = 1$ and $t = 2$ the main process as well as the secondary ones are organized in a manner described by Figure 1.

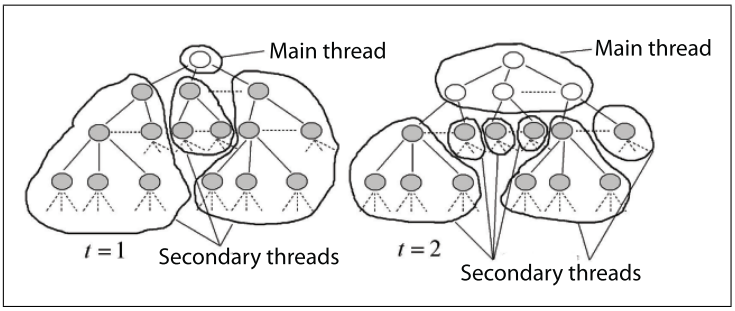


Fig. 1. The main thread and the secondary threads.

The number of secondary processes varies according to the constraints imposed by the problem statement and by the value of t.

Example 1. We consider a chess board with the dimensions $n \times n$ on which n queens have to be places so that they do not attack one another.

Figure 2 shows the number of secondary processes in the case of the problem given in example 3, built for various values of n and t.

	t = 1	t = 2	t = 3	t = 4
n = 14	14	156	1364	9632
n = 15	15	182	1764	13980
n = 16	16	210	2236	19688
n = 17	17	240	2786	27020
n = 18	18	272	3420	36264

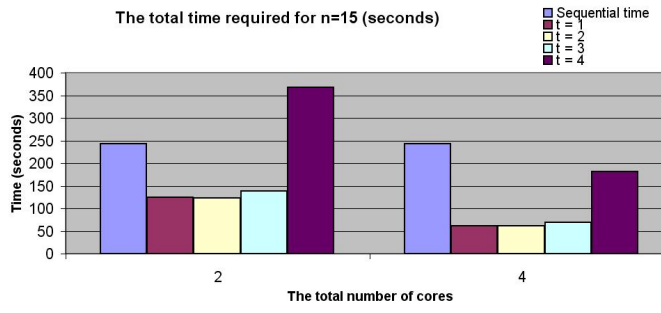
Fig. 2. The number of secondary processes for the given n and t.

The performance level of the algorithm given above is analyzed using two systems. The first system is a dual AMD Opteron 280. The second system is a AMD

Opteron 1218. Further details about AMD Opteron architecture can be found in [17] The multicore backtracking algorithm is implemented using C and Pthreads[18][19]. POSIX Threads (Pthreads), is a POSIX standard for threads. The standard defines an API for creating and manipulating threads. Pthreads are most commonly used on Unix-like POSIX systems but Microsoft Windows implementations also exist. Pthreads defines a set of C programming language types, functions and constants. It is implemented with a pthread.h header and a thread library.

For $n = 15$, in the case of the queen's problem, one notices the evolution depending on t and on the number of cores (marked c) given in Figure 3 and for $n = 16$ in Figure 4. The time is measured in seconds. The efficiency is computed as follows, expressed in percentage:

$$\text{time for one core}/(c * \text{time for c cores}).$$



Sequential version time	243

Multicore version	t=1		t=2		t=3		t=4	
Number of cores	2	4	2	4	2	4	2	4
Time(seconds)	125	62	123	61	139	69	368	182
Efficiency	97.20%	97.98%	98.78%	99.59%	87.41%	88.04%	33.01%	33.37%

Fig. 3. The time spans measured for $n = 15$.

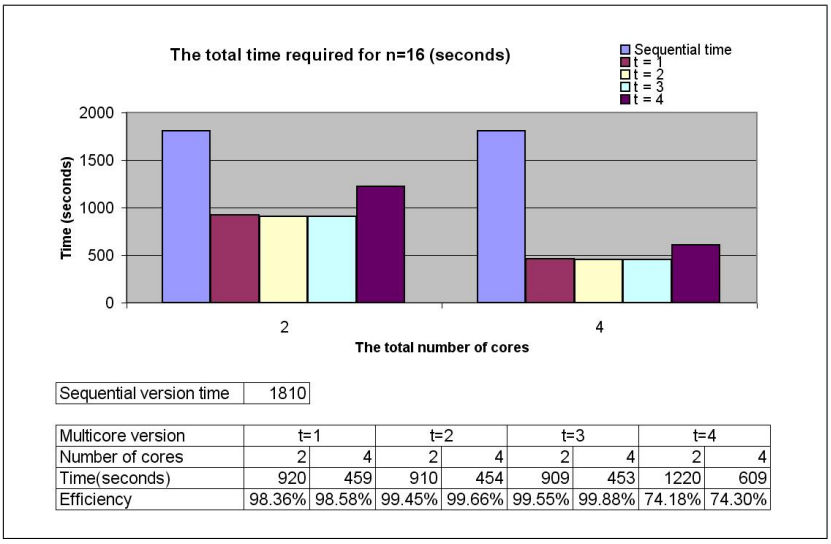
For $t > 2$ and $n \leq 16$ the total time span is higher because the amount of thread communication required and the necessary time needed for the creation of the secondary threads. For $t > 2$ and $n > 16$ the efficiency is increased because it creates more uniform threads in terms of execution time.

For $n = 17$, in the case of the queen's problem, one notices the evolution depending on t and on the number of cores (marked c) given in Figure 5 and for $n = 18$ in Figure 6.

One can notice that for a double number of cores, the time span needed to find the solutions is approximately cut down to half. For $n = 18$, the time span needed to find solutions is reduced from over 32 hours to about 8 hours, using a system with 4 cores(dual AMD Opteron 280).

As compared to the approaches using agents, the presented algorithm, allows a much better control over the number of transmitted messages. In the approaches

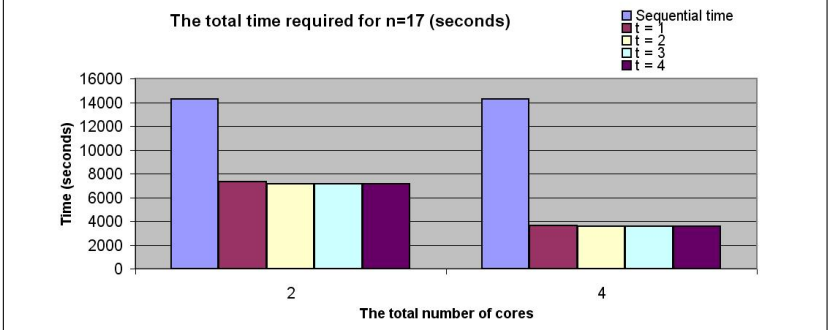
using agents, the number of messages transmitted by the participant processes is very large and difficult to control. For n-queen's problem, if every agent keeps the position of a queen, the number of messages for $n = 8$ is of figures of thousands in order to reach a single solution. Due to the high number of messages used to obtain all the solutions, the approach using agents may be difficult to use, especially when it's necessary to obtain all the solutions.



Sequential version time	1810

Multicore version	t=1		t=2		t=3		t=4	
Number of cores	2	4	2	4	2	4	2	4
Time(seconds)	920	459	910	454	909	453	1220	609
Efficiency	98.36%	98.58%	99.45%	99.66%	99.55%	99.88%	74.18%	74.30%

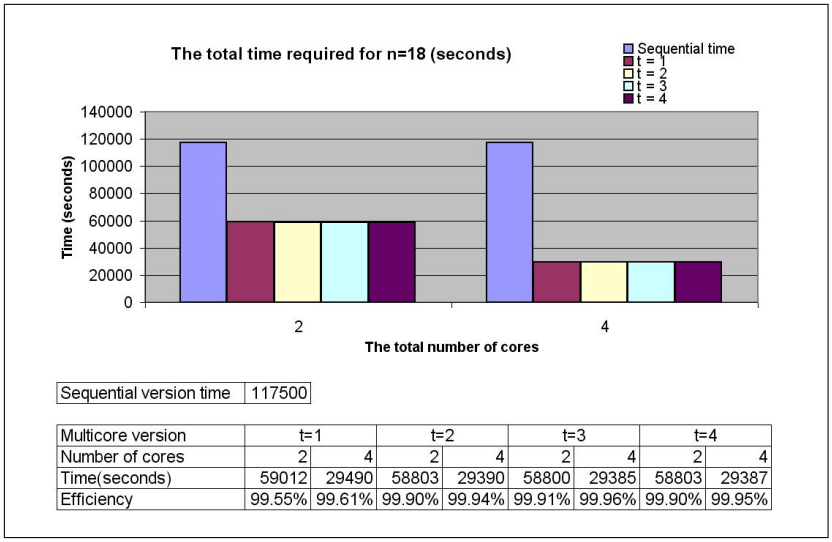
Fig. 4. The time spans measured for $n = 16$.



Sequential version time	14250

Multicore version	t=1		t=2		t=3		t=4	
Number of cores	2	4	2	4	2	4	2	4
Time(seconds)	7328	3614	7143	3567	7142	3566	7145	3570
Efficiency	97.22%	98.57%	99.74%	99.87%	99.76%	99.90%	99.72%	99.78%

Fig. 5. The time spans measured for $n = 17$.



Sequential version time	117500

Multicore version	t=1		t=2		t=3		t=4	
Number of cores	2	4	2	4	2	4	2	4
Time(seconds)	59012	29490	58803	29390	58800	29385	58803	29387
Efficiency	99.55%	99.61%	99.90%	99.94%	99.91%	99.96%	99.90%	99.95%

Fig. 6. The time spans measured for $n = 18$.

In the approach previously presented, through value t which determines the number of secondary threads, one can control the number of messages transmitted between the main process and the secondary threads. Controlling the number of messages and reducing their number, allows the improvement of the ratio between the time needed for transmitting the messages and the time allocated for searching the solutions. If the number of threads and transmitted messages is very big and the amount of calculations for each secondary thread is somehow reduced, it can cause an excessive fragmentation of the process of finding the solutions. This leads to an increase of the total time for identifying the solutions. Excessive fragmentation may be avoided because the previous algorithm allows indirectly the controlling of the number of secondary threads generated through value t.

The presented algorithm allows a possibility of return in case of errors. With algorithms based on agents, in case of malfunction of a computer implied in the searching of a solution, the resumption of the searching of a solution, the resumption of the searching process is quite complex. With the previously presented algorithm, only the searching of the solutions in the subspace of solutions allocated to that core is affected. In case of malfunction of the core which hosts the thread occurs, the recover supposes only the generation of another secondary thread meant to perform the search in the same subspace of possible solutions.

4. Conclusions

The algorithm presented could be used to solve backtracking problems on multicore systems. Experiments show that the algorithm scales well with large jobs. If the job is small, the efficiency is low because the calculation volume is not so important

with respect to the amount of communication required and the necessary time needed for the creation of secondary threads.

The algorithm we presented can be implemented on different programming environments in order to turn into account the specific performances obtained by different hardware platforms.

Modern processors have multicore architectures that offer true hardware parallelism at low cost and the number of cores will continue to grow. This development will have important impacts on software engineering theory and practice, as every programmer will be confronted with parallel systems. The community needs to prepare for that situation.

References

[1] Advanced Micro Devices – Press Resources, `http://www.amd.com/us-en/Corporate/VirtualPressRoom/0,,51_104_543_15944~131372,00.html`

[2] Intel Corporation, `http://www.intel.com/p/en_US/products/server/processor/xeon7000/specifications`

[3] Brito I., Meseguer P., *Synchronous, Asynchronous and Hybrid Algorithms for DisCSP*, *Workshop Proceedings of The Fifth International Workshop on Distributed Constraint Reasoning*, Toronto, Canada, September 27, 2004.

[4] Zivan R., Meisels A., *Synchronous and Asynchronous Search on DisCSPs*, in *Proceedings of the First European Workshop on Multi-Agent Systems (EUMA)*, Oxford, UK, 2003.

[5] Yokoo M., Durfee E.H., Ishida T., Kubawara K., *Distributed constraint satisfaction for formalizing distributed problem solving*, in *Proceedings DCS*, 1992, pp. 614–621.

[6] Yokoo M., *Asynchronous weak-commitment search for solving distributed constraint satisfaction problems*, in *Proceedings CP95*, Cassis, France, 1995, pp. 88–102.

[7] Yokoo M., Durfee E.H., Ishida T., Kubawara K., *The distributed constraint satisfaction problem: formalization and algoritms*, IEEE Transactions on Knowledge and Data Engineering, 1998, **10**:673–685.

[8] Armstrong A., Durfee E., *Dynamic prioritization of complex agents in distributed constraint satisfaction problems*, in *Proceedings IJCAI97*, Nagoya, Japan, 1997.

[9] Hamadi Y., Bessire C., Quinqueton J., *Backtracking in distributed constraint networks*, in *Proceedings ECAI98*, Brighton, UK, 1998, pp. 219–223.

[10] Hamadi Y., *Interleaved backtracking in distributed constraint networks*, International Journal on Artificial Intelligence Tools, 2002, vol. **11**, no. 2, pp. 167–188.

[11] Silaghi M.C., Sam-Haroud D., Faltings B., *Asynchronous search with aggregations*, in *Proceedings AAAI00*, Austin, Texas, 2000, pp. 917–922.

[12] Bessiere C., Maestre A., Meseguer P., *Distributed dynamic backtracking*, in M.C. Silaghi, editor, *Proceedings of the IJCAI'01 workshop on Distributed Constraint Reasoning*, Seattle WA, 2001, pp. 9–16.

[13] Casas A., Carro M., Hermenegildo M.V., *A High-Level Implementation of Non-deterministic, Unrestricted, Independent And-Parallelism*, *Proceedings of the 24th International Conference on Logic Programming*, Udine, Italy, 2008, pp. 651–666.

[14] Chu M., Ravindran R., Mahlke S., *Data Access Partitioning for Fine-grain Parallelism on Multicore Architectures*, *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, 2007, pp. 369–380.

[15] Pankratius V., Schaefer C., Jannesari A., Tichy W.F., *Software engineering for multicore systems: an experience report*, *Proceedings of the 1st international workshop on Multicore software engineering*, Leipzig, Germany, 2008, pP. 53–60.

[16] Giurgiu C.-F., *An implementation of the backtracking algorithm for distributed systems*, Analele Universitatii de Vest din Timisoara, Seria Matematica-Informatica, Volume **XLVI**, Issue 1, 2008, pp. 61–74.

[17] Advanced Micro Devices, `http://www.amd.com/us-en/Processors/ProductInformation/0,,30_118_8796,00.html`

[18] The Open Group, `http://www.opengroup.org/onlinepubs/007908799/xsh/pthread.h.html`

[19] Blaise Barney, Lawrence Livermore National Laboratory POSIX Threads Programming, `https://computing.llnl.gov/tutorials/pthreads/`