A Parallel implementation on a Multi-Core Architecture of a Dynamic Programming Algorithm applied in Cognitive Radio Ad hoc Networks

Badr Benmammar[1], Youcef Benmouna[1], Asma Amraoui[1], Francine Krief[2]

[1]LTT Laboratory, University of Tlemcen, Algeria
[2]LaBRI Laboratory, Bordeaux INP, Talence, France

Abstract: Spectral resources allocation is a major problem in cognitive radio ad hoc networks and currently most of the research papers use meta-heuristics to solve it. On the other side, the term parallelism refers to techniques to make programs faster by performing several computations in parallel. Parallelism would be very interesting to increase the performance of real-time systems, especially for the cognitive radio ad hoc networks that interest us in this work. In this paper, we present a parallel implementation on a multi-core architecture of dynamic programming algorithm applied in cognitive radio ad hoc networks. Our simulations approve the desired results, showing significant gain in terms of execution time. The main objective is to allow a cognitive engine to use an exact method and to have better results compared to the use of meta-heuristics.

Keywords: Parallel computing, dynamic programming, cuckoo search, cognitive radio ad hoc networks.

1. Introduction

The rapid development of wireless communications increasingly leads to scarcity of spectrum due to the fixed policy of frequency allocation. To solve this lack of spectral resources, researchers thought to develop new wireless communication technologies such as cognitive radio (CR). CR is considered to be one of the most promising technologies for future wireless communication networks because it is based on the dynamic spectrum allocation and therefore makes it possible to use it more effectively. CR system uses its intelligence to optimize the user's quality of service by adapting its transmission parameters.
The idea of CR was officially presented in 1998 by Joseph Mitola III in a seminar at KTH, the Royal Institute of Technology, later published in an article by Mitola and Gerald Q. Maguire Jr in 1999 [1]. CR is a form of wireless communication in which a transmitter/receiver can intelligently detect which communication channels are in use and which are not, and can transfer to the unused ones. This allows optimum use of the available radio frequencies in the spectrum, whilst minimizing interference with other users [2].
In CR, there are two types of users: primary users (PU), those with assigned frequency bands and secondary users (SU) who do not have a license. These will be through CR allowed to use the free parts of the frequency bands of PUs without harmful interference.
CR system requires four major functions that enable it to opportunistically use the spectrum [3]. These functions consist in the CR terminal's main steps for spectrum management. They are: spectrum sensing, spectrum decision, spectrum sharing, and spectrum mobility.
Parallel computing is a type of computation in which the execution of processes is carried out simultaneously. The term parallelism refers to techniques to make programs faster by performing several computations in parallel. This requires hardware with multiple processing units. Parallel computing is closely related to concurrent computing, they are frequently used together, and often conflated, though the two are distinct: it is possible to have parallelism without concurrency (such as bit-level parallelism), and concurrency without parallelism (such as multitasking by time-sharing on a single-core CPU) [4].
Since the minimization of the task execution time in cognitive radio ad hoc networks (CRAHNs) is very important for increasing the performance of this type of network, we are interested in this work to the implementation of an exact method (dynamic programming) using multi core architecture in CRAHNs. We also implement a meta-heuristic (cuckoo search) and compare it to our dynamic programming implementation in order to make a wider validation of our contribution.
In this paper, we start by giving an overview on exact methods and meta-heuristics. We will focus, in particular on dynamic programming for the exact methods and cuckoo search for the meta-heuristics. We will then establish a state of the art on the use of dynamic programming and cuckoo search in communication networks. Then, we present our contribution in this paper which consists in implementing a parallel dynamic programming algorithm on a multi-core architecture applied in CRAHNs. We also implement cuckoo search and compare it to our dynamic programming implementation using threads. We conclude our work at the end of the paper.

2. Resolution methods

In the literature, there are 2 types of algorithms. Exact algorithms that find the optimal solution and can take an exponential number of iterations and approximate algorithms that produce a suboptimal solution and do not take an exponential number of iterations.

2.1. Exact methods

Exact methods such as branch and bound [5], cutting planes [6] or dynamic programming, guarantee finding an optimal solution of a problem for an instance of finite size in a limited time and of prove optimality.
In this article we will use dynamic programming and it is for this reason that we will present it in the following with a state of the art on its application in communication networks.

2.1.1. Dynamic programming

Dynamic programming was first coined by *Richard Bellman* in the 1950s [7], a time when computer programming was an esoteric activity practiced by so few people. Back then programming meant "planning" and "dynamic programming" was conceived to optimally plan multistage processes. Dynamic programming is a powerful technique that can be used to solve many problems in time $O(n^2)$ or $O(n^3)$ for which a naive approach would take exponential time. Dynamic programming is a design paradigm that can be seen as an improvement or adaptation of the divide-and-conquer method except that unlike divide-and-conquer, the sub problems with dynamic programming will typically overlap.

The effectiveness of this method is based on the principle of optimality stated by the mathematician *Richard Bellman*: "Every optimal policy is composed of optimal sub-policies".

The design of a dynamic programming algorithm consists of four steps [8]:

- Characterization of the structure of an optimal solution.
- Recursive definition of the value of the optimal solution.
- Ascending calculation of the value of the optimal solution.
- Construction of the optimal solution from the information obtained in the previous step.

For instance, we can say that finding the shortest path in a graph or the knapsack example are problems that can be solved using dynamic programming.

The first property of dynamic programming corresponds to being able to write down a recursive procedure for the problem we want to solve. The second property corresponds to making sure that this recursive procedure makes only a polynomial number of different recursive calls.

2.1.2. Application of dynamic programming in communication networks

In what follows, we will cite some works that have been interested to the application of dynamic programming in communication networks.

Fontes et al. have defined a dynamic programming approach to solve optimally the single-source incapacitated minimum cost network flow problem with general concave costs [9]. The new dynamic programming approach proposed by the authors does not based on the type of cost functions considered and on the number of nonlinear arc costs. Computational experiments were performed using randomly generated problems. The computational results reported for small and medium size problems indicate the effectiveness of the proposed approach.

Fonoberova et al. presents an approach for resolving some power systems problems by using optimal dynamic flow problems [10]. The classical optimal flow problems on networks are prolonged and generalized for the cases of nonlinear cost functions on arcs, multicommodity flows, and time-and flow-dependent transactions on arcs of the network. All network parameters are pretended to be dependent on time. The algorithms for solving such kind of problems are developed by using special dynamic programming techniques based on the time-expanded network method together with classical optimization methods.

Ilyas et al. have suggested the use of an operational and low-complexity dynamic programming algorithm (DPA), which is used in conjunction with four different route discovery algorithms [11]. The authors accomplish complexity analysis, statistical evaluation of changes in power consumption rates effected, and verify spatial redistribution of energy consumption of sensors in the network. The results on multihop networks of 100 randomly placed nodes show that, on average, the two best performing variants of DPA yield a reduction of up to 28% and 36% in power consumption rate variance at the cost of raising average power consumption by 15% and 21%, respectively. Computational complexities of DPA variants range from $O(N^3)$ to $O(N^4)$, which is significantly lower than linear search of the solution space of $O(N!^N)$. Analysis by diffusion plots shows that DPA decreases power consumption of sensors that experience the highest power consumption under the shortest path routes.

In the area of interest (CRAHNs), there are few research papers that have used dynamic programming. For example, *Liang et al.* have proposed a dynamic programming based power control algorithm with the consideration of primary user's QoS by a rate loss constraint criterion [12]. In the suggested algorithm the occupancy for each subcarrier by primary users is modeled as a discrete-time Markov chain and the concept of rate loss constraint is used to guarantee primary user's desired rate. Simulation results show that the proposed algorithm can obtain a maximum average data rate over a finite time horizon and effectively guarantee primary users' QoS.

Wang et al. have presented a novel power allocation algorithm to maximize the throughput under the bit error rate (BER) constraint and the total power constraint in cognitive radio [13]. Although water filling (WF) algorithm is the optimal power allocation in theory, it ignores the fact that the allocated power in use is discrete, and it doesn't take the waste power into consideration. In the improved algorithm, the total power is asymmetrically quantized to apply to the practice and reduce the computation complexity before adopting the dynamic programming which is commonly used to solve the knapsack problem, so this improved algorithm is called as asymmetrically quantized dynamic programming (AQDP). Moreover, AQDP reused the residual power to maximize the throughput further. The simulation results show that AQDP has improved the transmit throughput of all CR users compared with the classical power allocation algorithms referred as WF in this paper.

Gao et al. have discussed power and rate control schemes for a single cognitive radio channel in the presence of licensed primary radios (PRs) [14]. A dynamic programming based algorithm is proposed to maximize the long-term average rate for the CR link under constraints on the total energy budget and the CR-to-PR disturbance. In the suggested algorithm, the behavior of PRs is modeled as a two-state Markov chain. Based on such a model, the optimal power and rate control strategy for each time slot is derived, which is a function of the energy level at the beginning of current time slot and the previous behavior of PRs. Simulation results show that the proposed algorithm can lead to an important performance improvement in term of the long-term average rate while keeping the probability of CR-to-PR disturbance below a given level.

2.2. Approximate methods

Exact resolution methods ensure an optimal solution, but their use in practice is very rare because of their complexity. These methods can take a long time, especially when problems are large. In some cases, the user may prefer a reduced calculation time instead of an optimal solution. For this purpose, approximate resolution methods are applied in order to find an approximate solution to a problem and thus to speed up the resolution process. In the literature, we speak about heuristics and meta-heuristics. A heuristic comes from the Greek word "heureka" which means "to find". It is an approximate method that seeks good solutions that are close to optimality in polynomial time. Faced with the difficulties encountered by heuristics in obtaining a realizable solution of good quality for difficult optimization problems, meta-heuristics have appeared. They generally make it possible to obtain a solution of very good quality for problems whose methods are not known to be effective to treat them or when the resolution of the problem requires a high time or a large storage memory. Most meta-heuristics use random and iterative processes as a means of gathering information and exploring the search space. Meta-heuristics can be classified in many ways. One can distinguish between local search methods and global optimization methods. Others can distinguish single solution methods and population of solution methods.

Single solution methods use only one solution. Algorithms such as: Hill climbing [15], simulated annealing [16] and Tabu search [17] are part of this class of meta-heuristics.

Genetic Algorithms (GA) [18], Particle Swarm Optimization (PSO) [19], Ant Colony Optimization (ACO) [20] and Cuckoo Search (CS) algorithms are the best known examples in methods that manipulate a population of solution. In this article we will use cuckoo search and it is for this reason that we will present it in the following with a state of the art on its application in communication networks.

2.2.1. Cuckoo search

Cuckoo search (CS) is an optimization algorithm developed by *Xin-she Yang and Suash Deb* in 2009 [21]. CS is one of the latest meta-heuristic optimization algorithms; it is inspired from the brood parasitism of some cuckoo species. They lay eggs from other birds' nests and even remove host eggs to increase the probability of their eggs getting hatched. Three types of brood parasitism exist: Intra-specific, cooperative breeding and nest takeover. Some species of host birds simply throw out cuckoo's eggs or even leave their nest and put up a new one when alien eggs are discovered.

Certain cuckoo species are expert to imitate the color and texture of the eggs of chosen host species, as a result of which, the probability of their eggs to be abandoned reduces and intensifies their reproductively.

In general the parasitic eggs hatch slightly earlier than that of the host eggs. In addition, when the first cuckoo chick is hatched, the first instinct action it will take is to evict the host eggs randomly out of the nest for the sake of its own existence and food. Studies also show that cuckoo chicken imitates the sound of host chicken to gain access to more feeding opportunity.

The cuckoo searches for food in a random manner; the next move of this animal is based on the current location/state and the transition probability to the next location. It explores their landscape using a series of straight flight paths punctuated by a sudden 90 degree turn, according to [21]. The cuckoo utilizes a search pattern called Lévy flight.

In the whole process, we consider these three conditions:

- One egg will be laid at a time by each cuckoo in any nest chosen randomly.
- Nests that have better eggs are postponed to the next generation.
- The probability that the host species discovers the cuckoo's egg is within the probability range $pa \in [0, 1]$ and the total number of nests is fixed.

The cuckoo search algorithm starts with initial an initial population of solutions generated randomly. When the host species discovers the cuckoo's egg in its nest, it will abandon the nest or throw away that egg. The implementation of this last point is done by replacing *pa* of worse nests and builds new ones at new locations. Each egg corresponds to a feasible solution and its fitness value is calculated.

Levy flights are more efficient than Brownial random walks in exploring unknown, large-scale search space [22]. So, a new solution is formed using the concept of Lévy flight (1)

$$\boldsymbol{X_i(t+1)}=\boldsymbol{X_i(t)}+\boldsymbol{\alpha} \oplus \textbf{\textit{Lévy}}\ (\boldsymbol{\lambda}) \qquad (1)$$

Where α > 0 is the step size which should be related to the scales of the problem of interests. In most cases, we can use α = 1. The above equation is essentially the stochastic equation for random walk. In general, a random walk is a Markov chain whose next status/location only depends on the current location (the first term in the above equation) and the transition probability (the second term). The product ⊕ means entrywise multiplications. This entrywise product is similar to those used in particle swarm optimization, but here the random walk via Lévy flight is more efficient in exploring the search space as its step length is much longer in the long run.

The Lévy flight essentially provides a random walk while the random step length is drawn from a Lévy distribution (2):

$$\textbf{\textit{Lévy}} \sim \mathbf{u} = t^{-\lambda}; \qquad 1 < \lambda \leq 3 \qquad (2)$$

An efficient nonlinear relationship of variance of Lévy flight (3) is used to explore large unknown search spaces.

$$\boldsymbol{\sigma^2(t) \sim t^{2-\beta}; 1 \leq \beta \leq 2} \qquad (3)$$

The CS algorithm can be summarized as follow [21]:

```
Begin
 1: Define objective function f (x), x = (x_1, x_2, ..., x_d)
 2: Initial a population of n host nests x_i (i=1,2,...,n)
 3: While ((t < MaxGeneration) or (stop criterion)) do
 4: Get a cuckoo (say i) randomly and generate a new
    solution by Lévy flights
 5: Evaluate its quality/fitness F_i
 6: Choose a nest among n (say j) randomly
 7: if (F_i > F_j) then
 8:        Replace j by the new solution
 9: end
10: Abandon a fraction (pa) of worse nests and build new
    ones at new locations
11: Keep the best solutions (or nests with quality solutions)
12: Rank the solutions and find the current best
13: end while
14: Post process results and visualization
End
```

2.2.2. Application of cuckoo search in communication networks

Few studies are interested in the application of CS in wireless sensor networks and vehicular ad hoc network. However, we have not found credible work using CS in cognitive radio networks.

Dhivya et al. have used CS to aggregate data in wireless sensor network [23]. In the recommended technique, the least energy nodes are formed as clusters for sensing the data and high energy nodes as Cluster Head for communicating to the Base Station (BS). The modified CS is suggested to get enhanced network performance incorporating balanced energy dissipation and results in the formation of optimum number of clusters and minimal energy consumption. The feasibility of the scheme is manifested by the simulation results on comparison with the traditional cluster based routing methods. *Cheng et al.* have introduced an effective CS algorithm for node localization in wireless sensor network [24]. Based on the modification of step size, this approach enables the population to approach global optimal solution rapidly, and the fitness of each solution is employed to build mutation probability for avoiding local convergence. In addition, this approach restricts the population in the certain range so that it can prevent the energy consumption caused by important search. Extensive experiments were conducted to study the effects of parameters like anchor density, node density and communication range on the proposed algorithm with respect to average localization error and localization success ratio. Furthermore, a comparative study was conducted by the authors to realize the same localization task using the same network deployment. Experimental results show that the proposed CS algorithm can not only increase convergence rate but also reduce average localization error compared with standard CS algorithm and particle swarm optimization algorithm.

Ramakrishnan et al. have suggested a design of adaptive routing protocol based on CS algorithm in vehicular ad hoc network [25]. This new protocol integrates the features of both topology routing and geographic routing protocols which ensures the secured transmission of data with less delay and high packet delivery ratio. The proposed algorithm provides reliable and secure routes between source and destination node with optimal distance and low routing overheads. This algorithm uses a local stochastic broadcasting to find routes which reduces the network congestion thereby improving the packet delivery ratio.

3. Our implementation of parallel dynamic programming on a multi-core architecture in CRAHNs

In *Amraoui et al.* [26], we have already used dynamic programming in order to solve the spectrum allocation problem in CRAHNs. The algorithm was applied with the first-price sealed-bid auction (FPSBA). Our CRAHNs consisted of only 1 PU and n SU. The objective of the PU is to maximize its gain. So, there is only one PU who shares its spectrum and several SU who need to ensure free channels for assuring the quality of their application.

Figure 1 illustrates the scenario that was dealt in *Amraoui et al.*

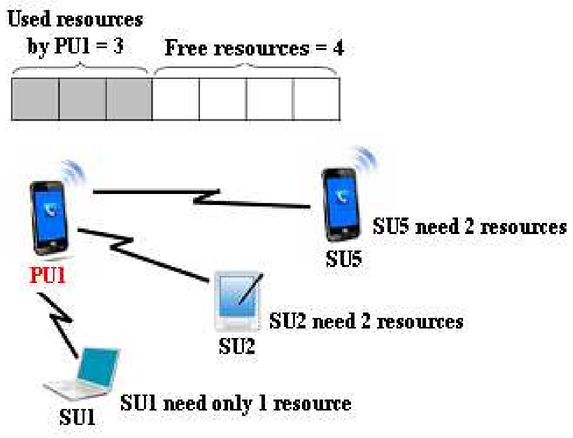

Figure 1. Proposed scenario in *Amraoui et al.*

For example, in figure 1, there are 3 SUs who want to access to the free resources at PU1. However there are only 4 free resources at PU1 which is not sufficient to serve the needs of all SUs (1+2+2 = 5 required resources). So which of SU1 or SU2 or SU5 will access to the spectrum? To solve this kind of problem, we have used dynamic programming.

To measure performance, we have used in *Amraoui et al.* a simple machine (laptop with 4GO of RAM). The number of SU was limited to 5 and the number of channels available on the PU side was limited to 4. The execution time was too short (3 milliseconds (ms)). In fact, we have treated a simple matrix of size 5*4. The execution time will multiply by 1000 (3 seconds) by increasing the number of SU to 12000 and the number of available channels on the PU side to 15000. This is the maximum capacity of our machine (processing a matrix of integers of size 12000*15000). In a real-time system like cognitive radio networks, 3 seconds is unacceptable.

However, and in order to improve this time and therefore, increase network performance, we propose in this paper an improvement of the algorithm proposed in *Amraoui et al.* using threads. We also aim to guarantee scalability. Indeed, having a fairly large number of users and communication channels is a scenario that will soon happen with the emergence of IoT (Internet of things) [27]. It should also be noted that CR is a promising enabler communication technology for IoT [28].

For our current work, we have used the multi-core architecture available at the polytechnic faculty of Mons, university of Mons Belgium [29]. We have used 32G of RAM with 32 cores (threads). In this case, we were able to process a matrix of size 30000*50000. 30000 is the number of SU, 50000 is the number of channels available on the PU side.

The code of dynamic programming implemented on the PU side in *Amraoui et al.* is as follows:

```
Function COUT (W, C, m)
Begin
 1:     n = C.length
 2:     for j = 0 to m do
 3:         T[0][j] = 0
 4:     end for
 5:     for i =1 to n do
 6:         for j=1 to m do
 7:             if (j>=W[i-1])
 8:                 T[i][j] = max(T[i-1][j],
                                        T[i-1][j-W[i-1]] + C[i-1])
 9:                            else
10:                                     T[i][j] = T[i-1][j]
11:                            end if
12:                    end for
13:          end for
14:          return T[n] [m]  // The total cost obtained by the PU
End
```

n is the number of SUs.
m is the number of free channels at PU.
W is an array of size *n*.
W[i] is the number of requested channels by SU_i.
C is an array of size *n*.
C [i] is the proposed price for *W [i]* by SU_i.
The increasing monotonic function to be optimized is: $Max \sum_{i=1}^{n} C[i] * x_i$.
The constraint is $\sum_{i=1}^{n} W[i] * x_i \leq m$.
x_i is a binary values. If it is equal to 1 it means that the SU_i is part of the solution (there is a spectrum that has been assigned to SU_i). x_i is equal to 0 otherwise.
T is a matrix of size *n*m*.
T [n] [m] is the maximum gain obtained by the PU.
As shown in figure 2, the matrix on which the previous program is working is of size n*m (see matrix T in the previous code).
The code runs row by row and the result of the next row will depend on that of the previous row.

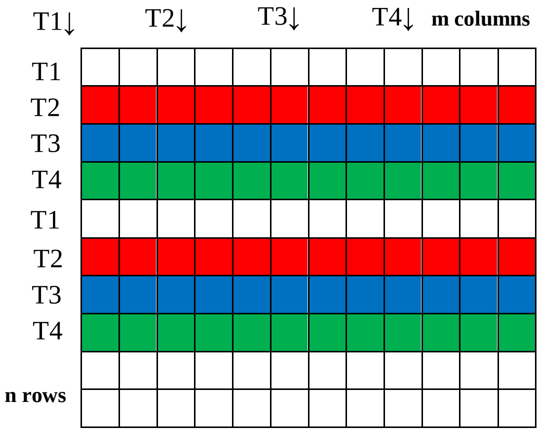

Figure 2. Matrix of size n*m.

As we have already explained, dynamic programming is an optimization approach that transforms a complex problem into a sequence of simpler problems; its essential characteristic is the multistage nature of the optimization procedure.
In the case of 4 threads for example (see figure 2), our idea is to run the first thread on the first row of the matrix, the second thread on the second row, the third thread on the third row, and so on. But as the processing of the second thread depends on the result of the first thread and the processing of the third thread depends on the result of the second thread and so on. The second thread will be launched on the second row when the first thread has processed m/4 cells of the first row of the matrix (4 here is the number of thread). The third thread will be launched on the third row when the first thread has processed 2*m/4 cells of the first row. The fourth thread will be launched on the fourth row when the first thread has processed 3*m/4 cells of the first row. Thereafter, the first thread will continue to work on the 5th row. The second thread will continue to work on the 6th row, the third thread on the 7th row and the 4th thread on the 8th row and so on until the whole matrix runs.
The delay between the launch of the threads (time required to process m/4 cells of the first row) is always the same because the four threads handle similar sub problems. The following function represents the behavior of the dynamic programming but on a single row *i* and between the columns of index *begin* and *end*.

```
Function dynamic (i, begin, end)
Begin
1: for j=begin to end do
2:    While (T[i-1][j]==-1) do sleep(0)
        // To synchronize two successive threads. In the case
        // of 4 threads for example, the 3rd thread must not
        // exceed the 4th thread
3:    End while
4: if (j>=W[i-1]) then
5:       T[i][j]= max(T[i-1][j],T[i-1][j-W[i-1]] + C[i-1])
6: else
7:       T[i][j]= T[i-1][j]
8: end if
9: end for
End
```

The thread implementation is represented by the following function.

```
 Function run ( )
 Begin
 1: n=C.length // The size of the matrix C represents the
    // number of SUs
 2: if (row==1) then  // Only for the first thread
 3: dynamic(1,1,m/nbthread) // Launching the first thread
    // on the first m/nbthread cells
 4: Dyn thread=new Dyn(2, m, C, W, nbthread)
   // Create a thread to work on row 2
   // Dyn is the constructor of the thread
 5: thread.start() // Run the second thread's run method
 6: for k=1 to nbthread-2 do
 7: dynamic(1,k*m/nbthread+1,(k+1)*m/nbthread)
// The first thread always works on a part of cells of size   //
m/nbthread
8: thread =new Dyn(k+2, m, C, W, nbthread)
// Create the (nbthread-2) threads to work on the
// corresponding rows
9: thread.start()  // Run the thread's run method
10: end for
11:dynamic(1,(nbthread-1)*m/nbthread+1,m)
// The first thread continues to work on the last part of the first
// row
12: row=row+nbthread
//The first thread processes the matrix with a nbthread step
13: while (row<=n) do
14: dynamic(row,1,m)
//The first thread processes the entire row
15: row=row+nbthread
16: end while
17: else // For all threads that have already been started by
// the first thread
18: dynamic(row,1,m)
// The thread processes the entire row
19: row=row+nbthread
// The thread processes the matrix with a nbthread step
20: while (row<=n) do
21: dynamic(row,1,m)
```

```
// The thread processes the entire row
22: row=row+nbthread
23: end while
24: end if
End
```

The following main program is used to run *nbthread* threads.

```
Function main ( )
Begin
 1: nbthread=2 // 4, 8, 16 or 32
 2: m=50000 // number of channels available on the PU
    // side
 3: n=30000 // number of SU
 4: row=1
 5: T : array of size (n+1)*(m+1)
 6: for j=0 to m T[0][j]=0 end for // There is only the first
    // row of T that is initialized to 0
 7: for i=1 to n do
 8: for j=0 to m do T[i][j]=-1 end for // Initialize the
     // remainder of T to synchronize two
     // successive threads
 9: end for
10: W : array of size n
11: for i=0 to n-1 do W[i]= i/1000+10 // number of channels
12: end for
13: C : array of size n
14: for i=0 to n-1 do C[i]= i/100+100 // price
15: end for
16: Dyn T1=new Dyn(row, m, C, W, nbthread) // The first
    // thread works on the first row
    // T1 will then launch the other threads
17: T1.start() // launching the run method
End
```

To compare correctly, we used the same matrices C and W whatever the number of threads. So, we avoided generating the two matrices in a random way. The following table shows the obtained results for 1, 2, 4, 8, 16 and 32 threads respectively.

Table 1. Obtained results.

Number of threads	Execution time (T_i)	Reduction rate (R_i)	Improvement ratio (I_i)
1	5656 ms	0	1
2	3106 ms	45,08%	1,82
4	1614 ms	71,46%	3,50
8	880 ms	84,44%	6,42
16	**722 ms**	**87,23%**	**7,83**
32	768 ms	86,42%	7,36

$\mathbf{R_i = (T_1-T_i) / T_1}$: the reduction rate obtained with the use of multiple threads related to the use of a single thread.
$\mathbf{I_i=T_1/T_i}$: the improvement ratio obtained with the use of multiple threads related to the use of a single thread.
For the different executions (whatever the number of threads), the gain obtained by the PU is equal to **525573.** It is always the same because it is an exact method.
We note that the execution time for 32 threads was not improved compared to the case of 16 threads. Using more threads does not necessarily decrease execution time. Indeed, with a large number of threads, we have more parallelism and therefore more communications that impact the execution time.
The following three figures represent the obtained results.

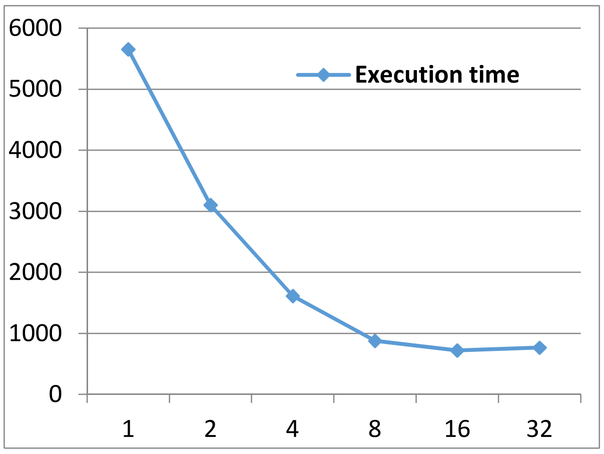

Figure3. Execution time related to the number of threads.

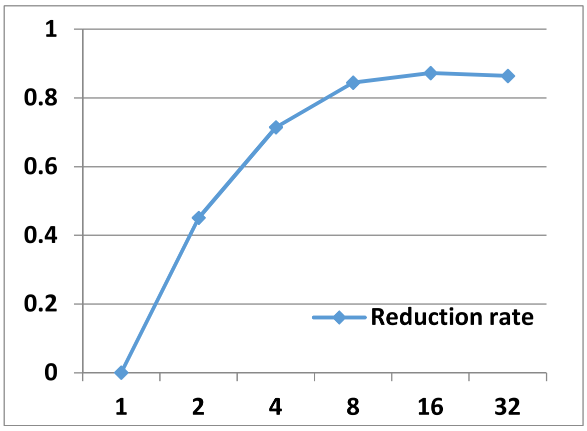

Figure 4. Reduction rate related to the number of threads.

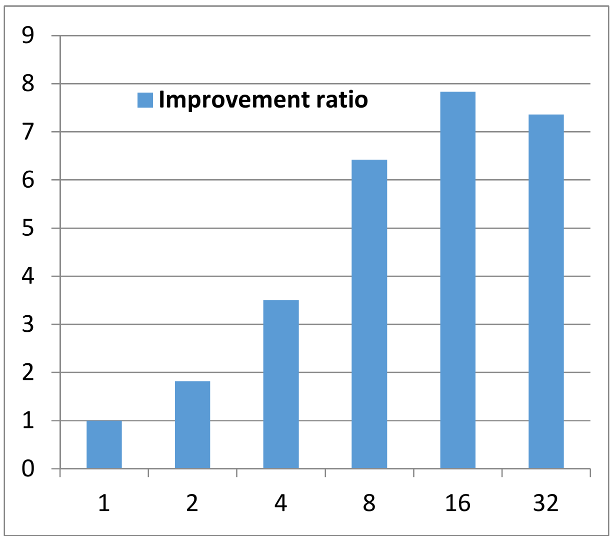

Figure 5. Improvement ratio related to the number of threads.

The best results are obtained with 16 threads. The execution time is reduced to more than 87% compared to the use of a single thread and the improvement ratio is greater than 7.
The improvement ratio which is equal to 7.83 (in the case of 16 threads) is very interesting because for a cognitive engine that does a processing of 1 minute using one thread; this processing can be done in less than 8 seconds using 16 threads.

To validate even more the obtained result, we have implemented the same problem of spectrum allocation with a meta-heuristic. For that we opted for the Cuckoo Search (CS). We made this choice because it is possible that we will encounter a scenario in which the cognitive user will be interested much more by the execution time than by the obtained gain. A meta-heuristic is faster than an exact method, but its result is less exact than that of the exact method.
The advantage of cuckoo search algorithm is that the number of tuning parameters is very less when compared to other algorithms like genetic algorithm or particle swarm optimization and hence can be easily applied to a wider range of optimization problem [30].

4. Our implementation of CS in CRAHNs

For our implementation with CS, each nest represents a solution for the spectrum allocation problem and a population of nest is used for finding the best solution of the problem. The formulation of the problem is the same that was used for dynamic programming. So, the increasing monotonic function to be optimized is: $Max \sum_{i=1}^{n} C[i] * x_i$ and each newly obtained solution should be satisfied according to the constraint: $\sum_{i=1}^{n} W[i] * x_i \leq m.$
Where : n is the number of SUs. m is the number of free channels at PU. W is an array of size n, $W[i]$ is the number of requested channels by SU_i. C is an array of size n, $C[i]$ is the proposed price for $W[i]$ by SU_i.

A population of N host nests is represented by $S = [S_1, S_2,\ldots,S_N]$ where each nest $S_i=[x_1,x_2,\ldots,x_n]$ represents a solution for the spectrum allocation problem.
x_i is a binary values. If it is equal to 1 it means that the SU_i is part of the solution (there is a spectrum that has been assigned to SU_i). x_i is equal to 0 otherwise.
The problem often encountered with meta-heuristics is the initialization of the parameters used for the simulation. So we started by choosing the size of the initial population as well as the number of iteration used to ensure the convergence of the meta-heuristic.
To choose the size of the initial population, we varied it from 25 to 200 with a step of 25. We chose the value of 0.1 for *pa* and 300 iterations for each population size. We also perform 10 simulations for each population size and we calculate the average of all the values found in order to make a good estimation of the obtained results. Table 2 shows the obtained results.

Table 2. Gain related to the population size.

Population size	Gain	Convergence interval
25	179550	70-80
50	180109	60-70
75	191166	90-100
100	**205472**	**115-125**
125	176898	75-85
150	188550	80-90
175	180459	70-80
200	196511	105-115

Note that regardless of the population size, the CS converges in all cases between 60 iterations (minimum value) and 125 iterations (maximum value).

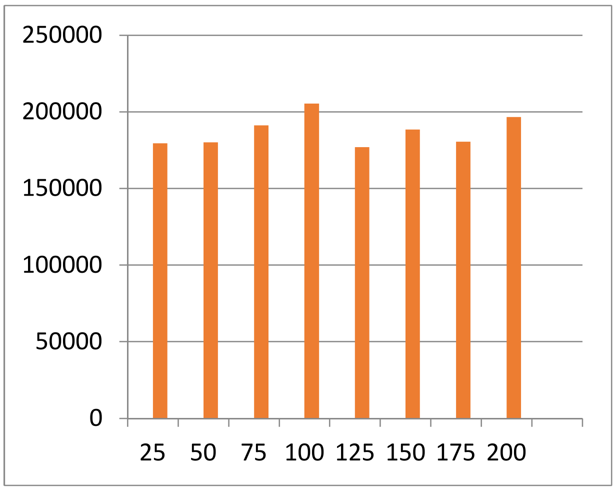

Figure 6. Gain related to the population size.

In the previous figure, we note that the maximum gain obtained by the PU corresponds to an initial population size equal to 100. In this case, the convergence is between 115 and 125 iterations. The gain obtained by the PU is equal to **205472.** It is calculated at iteration number 125 (at the end of the convergence).
In the following, and in order to solve the spectrum allocation problem with CS, we set the initial population size to 100 and we use 125 iterations (to ensure the convergence of the meta-heuristic). The following table shows the obtained result step by step in terms of obtained gain by the PU and execution time.
Figure 7 shows the convergence interval between the two iterations 115 and 125. In this case, the PU obtained a gain of 205472 after a run time of 2219 ms.

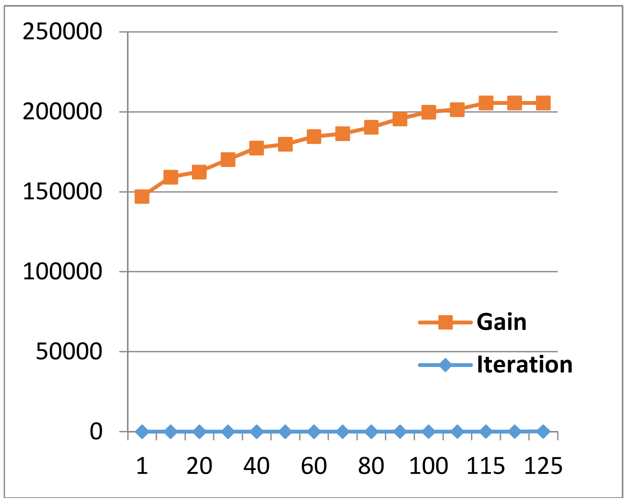

Figure 7. Obtained gain by the PU related to the number of iteration.

Table. 3 Obtained result in terms of obtained gain by the PU and execution time.

Iteration number	Gain	Time (ms)
1	147101	154
10	159259	335
20	162461	511
30	170148	682
40	177400	848
50	179768	1009
60	184546	1171
70	186318	1332
80	190262	1492
90	195556	1655
100	199739	1816
110	201362	1978
115	**205472**	**2059**
120	**205472**	**2140**
125	**205472**	**2219**

5. Comparison between parallel dynamic programming and cuckoo search

In the following, we will make a comparison between parallel dynamic programming and cuckoo search. The cuckoo search meta-heuristic allowed the PU to gain 205472 after a run time of 2219 ms. This gain represents 39% of that obtained with dynamic programming (525573).

Figure 8 shows the obtained results by dynamic programming and cuckoo search in terms of gain. This is not a surprise for us; the exact method provided a gain more interesting than meta-heuristic.

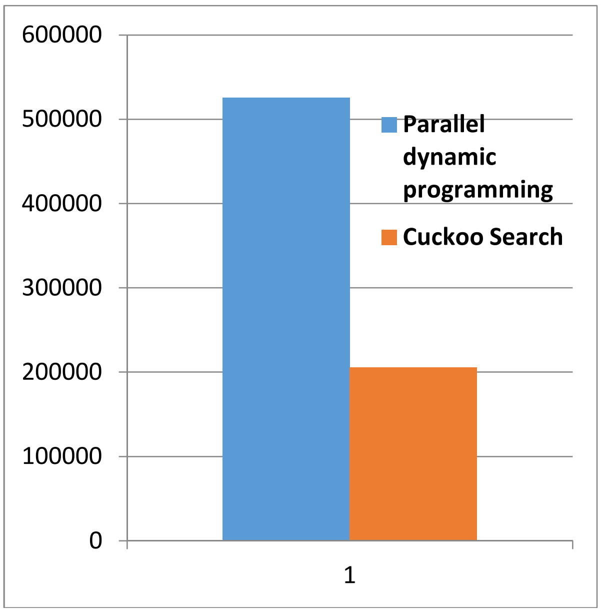

Figure 8. Comparison in terms of obtained gain by the PU.

Concerning the execution time, the dynamic programming provides the maximum gain with a single thread after more than 5 seconds. The cuckoo search provided 39% of this gain after more than 2 seconds.

On the other hand, we note, according to figure 9, that the execution time of the cuckoo search exceeds that of the parallel dynamic programming using more than 4 threads (cores).

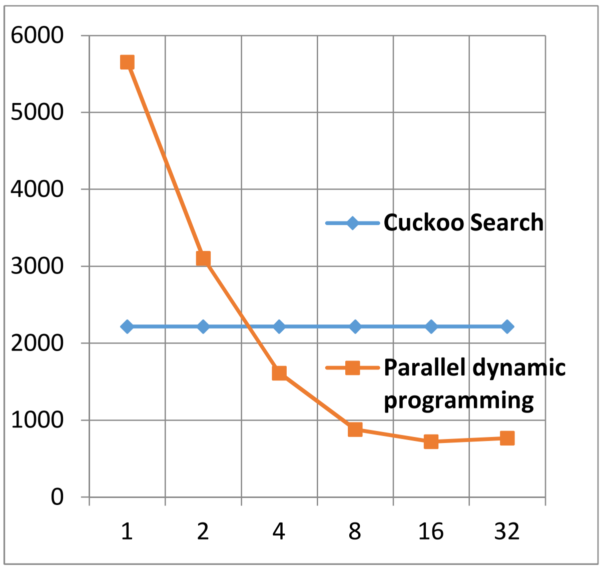

Figure 9. Comparison in terms of execution time

The best execution time for parallel dynamic programming is provided with 16 threads (722 ms).

With parallelism, dynamic programming (exact method) has exceeded performance in terms of execution time of a meta-heuristic which is the cuckoo search. This result is very interesting because the dynamic programming always provides the exact result (maximal gain in our case), a thing that cannot be guaranteed by a meta-heuristic.

So we can affirm the following result: cognitive engines typically using meta-heuristics can also use exact methods with threads to guarantee both a better execution time but also the results accuracy.

6. Conclusion

In this paper, we have presented a parallel version of dynamic programming applied in cognitive radio ad hoc networks. To measure the performance of our contribution, we have used the multi-core architecture available at the polytechnic faculty of Mons, university of Mons Belgium. Our simulations approve the desired results, showing significant gain in terms of execution time. The main objective is to allow a cognitive engine to use an exact method and to have better results compared to the use of meta-heuristics. Moreover, we compared the parallel dynamic programming with the cuckoo search meta-heuristic and we proved that with our parallel dynamic programming, in addition to having an exact result, we also have a very important gain in terms of execution time. As perspectives, we aim to study parallel dynamic programming but with more criteria (multi-objective fitness functions). In this paper, we have treated only channel prices as a single selling criterion used by the PU. Other criteria may be envisaged such as the allocation time or the transmission quality. We will also try to parallelize other exact methods such as branch and bound, cutting plane and branch and cut.

References

[1] J. Mitola, G. Q. Maguire. "Cognitive radio: making software radios more personal." IEEE personal communications, Vol. 6, No. 4, pp. 13-18, 1999.
[2] B. Benmammar, A. Amraoui, F. Krief. "A survey on dynamic spectrum access techniques in cognitive radio networks." International Journal of Communication Networks and Information Security, Vol. 5, No 2, pp. 68, 2013.
[3] Ian F. Akyildiz et al. "NeXt generation/dynamic spectrum access/cognitive radio wireless networks: A survey." Computer networks, Vol. 50, No 13, pp. 2127-2159, 2006.
[4] G. S. Almasi, A. Gottlieb. "Highly parallel computing". 1988.
[5] A. H Land, A. G. Doig. "An automatic method of solving discrete programming problems." Econometrica: Journal of the Econometric Society, pp. 497-520, 1960.
[6] R. Gomory. An algorithm for the mixed integer problem. No. RAND-P-1885. RAND CORP SANTA MONICA CA, 1960.
[7] R. Bellman. "Dynamic programming and Lagrange multipliers." Proceedings of the National Academy of Sciences. Vol. 42. No 10, pp. 767-769, 1956.
[8] H. M. Pande. Design analysis and algorithm. Firewall Media, 2008.
[9] D. BMM Fontes, E. Hadjiconstantinou, N. Christofides. "A dynamic programming approach for solving single-source uncapacitated concave minimum cost network flow problems." European Journal of Operational Research Vol. 174. No. 2. pp. 1205-1219, 2006.
[10] M. Fonoberova. "Algorithms for finding optimal flows in dynamic networks." Handbook of Power Systems II. Springer Berlin Heidelberg. pp. 31-54. 2010.
[11] M. U. Ilyas, H. Radha. "A dynamic programming approach to maximizing a statistical measure of the lifetime of sensor networks." ACM Transactions on Sensor Networks (TOSN). Vol. 8. No. 2, pp. 18, 2012.
[12] H. Liang, Hui, X. H. Zhao. "Dynamic programming based power control algorithm with primary user QoS guarantee for cognitive radio networks." Chinese Journal of Electronics. Vol. 22. No. 2, pp. 353-358. 2013.
[13] Q. Wang et al. "An Improved Dynamic Programming for Power Allocation in Cognitive Radio." Information Technology and Intelligent Transportation Systems. Springer International Publishing, pp. 43-51, 2017.
[14] L. Gao, W. Peng, S. Cui. "Power and rate control with dynamic programming for cognitive radios." IEEE GLOBECOM 2007-IEEE Global Telecommunications Conference. IEEE, 2007.
[15] S. J. Russell et al. Artificial intelligence: a modern approach. Vol. 2. Upper Saddle River: Prentice hall, 2003.
[16] S. Kirkpatrick, C. D. Gelatt, M. P. Vecchi. "Optimization by simmulated annealing." science Vol. 220, No.4598, pp. 671-680, 1983.
[17] F. Glover. "Tabu search-part I." ORSA Journal on computing. Vol. 1. No. 3. pp. 190-206. 1989.
[18] J. H. Holland, "Genetic algorithms." Scientific American. Vol. 267.No. 1. pp. 66-72. 1992.
[19] R. C. Eberhart, J. Kennedy. "A new optimizer using particle swarm theory." Proceedings of the sixth international symposium on micro machine and human science. Vol. 1. 1995
[20] M. Dorigo, V. Maniezzo, A. Colorni. "Ant system: optimization by a colony of cooperating agents." IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics). Vol. 26. No. 1. pp. 29-41. 1996.
[21] X. S. Yang, S. Deb. "Cuckoo search via Lévy flights." Nature & Biologically Inspired Computing, 2009. NaBIC 2009. World Congress on. IEEE, 2009.
[22] X. S. Yang, Nature-inspired metaheuristic algorithms. Luniver press, 2010.
[23] M. Dhivya, M. Sundarambal. "Cuckoo search for data gathering in wireless sensor networks." International Journal of Mobile Communications. Vol. 9. No. 6, pp. 642-656. 2011.
[24] J. Cheng, L. Xia. "An Effective Cuckoo Search Algorithm for Node Localization in Wireless Sensor Network." Sensors. Vol. 16. No. 9. pp. 1390. 2016.
[25] B. Ramakrishnan, S. R. Sreedivya, M. Selvi. "Adaptive routing protocol based on cuckoo search algorithm (ARP-CS) for secured vehicular ad hoc network (VANET)." International Journal of computer networks and applications (IJCNA). Vol. 2. No. 4. pp. 173-178. 2015.
[26] A. Amraoui, B. Benmammar, F. Krief, F. T. Bendimerad. Auction-based Agent Negotiation in Cognitive Radio Ad Hoc Networks, Fourth International ICST Conference, ADHOCNETS 2012, Paris, France, October 16-17, 2012, Revised Selected Papers Series: Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering, Vol. 111. pp. 119-134, Springer Edition, 2013.
[27] D. Lavrova, A. Pechenkin. "Applying Correlation and Regression Analysis Methods for Security Incidents Detection in the Internet of Things." International journal of communication networks and information security (IJCNIS). Vol. 7. No. 3. pp. 131-137. 2015.
[28] P. Rawat, K. D. Singh, and J. M. Bonnin. "Cognitive radio for M2M and Internet of Things: A survey." Computer Communications. Vol. 94, pp. 1-29. 2016.
[29] http://www.ig.fpms.ac.be/fr/content/cluster-de-calcul-ig
[30] B. M. Tharik. "A comparative study of firefly algorithm and cuckoo search algorithm in optimizing turning operation with constrained parameters." International Journal of Engineering Research and Technology. Vol. 2. No. 4. ESRSA Publications, 2013.