2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum

A parallel simulated annealing approach for the mapping of large process networks

François GALEA
Embedded real time systems laboratory
CEA, LIST
Gif-sur-Yvette, France
francois.galea@cea.fr

Renaud SIRDEY
Embedded real time systems laboratory
CEA, LIST
Gif-sur-Yvette, France
renaud.sirdey@cea.fr

***Abstract*—We propose a parallel simulated annealing approach to solve a dataflow process network mapping problem, where a network of communicating tasks is mapped into a set of processors with limited resource capacities, while minimizing the overall communication bandwidth between processors. The speedups obtained using this approach enables us to solve problems with more than one thousand tasks, on up to 48 processors, in reasonable time. Results have been obtained by taking profit of the specific architecture of a Non-Uniform Memory Access (NUMA) computer.**

***Keywords*-simulated annealing; parallelism; process network mapping.**

I. Introduction

With the end of the frequency version of Moore's law, a new generation of massively multi-core microprocessors is emerging. This has triggered a regain of interest for the so-called dataflow programming models in which one expresses computation-intensive applications as networks of concurrent processes (also called agents or actors) interacting through (and only through) unidirectional FIFO channels. See e.g. [2] for a recent instantiation of this model.

On top of more traditional compilation aspects, compiling a dataflow program in order to achieve a high level of dependability and performance on such complex processor architectures involves solving a number of difficult, large-size discrete optimization problems amongst which graph partitioning, quadratic assignment and (constrained) multi-flow problems are worth mentioning [5].

In this paper, we focus on the problem of mapping a dataflow process network (DPN) on a clusterized parallel microprocessor architecture composed of a number of nodes, each of these node being a small SMP, interconnected by an asynchronous packet network. With that respect, we present a parallel simulated annealing algorithms able to tackle relatively large instances of that problem in a reasonable amount of time.

The rest of this paper is organized as follows. Sect. II formally state the DPN mapping problem as well as justifies the practical relevance of choosing to design a parallel simulated annealing for that problem from both an optimization and a software engineering viewpoint. Sect. III describes the structure of our algorithm and Sect. IV provides some experimental results. Lastly, Sect. V conclude the paper with some perspectives.

II. The DPN mapping problem

A. Problem statement

Let T denote the set of tasks in the DPN and N the set of nodes. Let R denote the set of resources offered by the nodes (e.g., memory capacity, processing capability). Also, let w_{tr} denotes the consumption of tasks t in resource r, $q_{tt'}$ denote the bandwidth between tasks $t \neq t'$ and $d_{nn'}$ denote the routing distance between nodes $n \neq n'$. Also, for simplicity sake and with a slight loss of generality, we assume that all nodes are identical and we denote by C_r the capacity of any of the nodes for resource r.

Given the variables

$$x_{tn} = \begin{cases} 1 & \text{iff task } t \text{ is assigned to node } n, \\ 0 & \text{otherwise,} \end{cases}$$

our DPN mapping problem can then be expressed as the following mathematical program:

$$\begin{cases} \text{Minimize } \displaystyle\sum_{t\in T}\sum_{t'\neq t}\sum_{n\in N}\sum_{n\neq n'} x_{tn}x_{t'n'}q_{tt'}d_{nn'}, \\ \text{s. t.} \\ \displaystyle\sum_{n\in N} x_{tn} = 1 \ \forall t \in T, & (1) \\ \displaystyle\sum_{t\in T} w_{tr}x_{tn} \leq C_r \ \forall n \in N, r \in R, & (2) \\ x_{tn} \in \{0,1\} \ \forall t \in T, n \in N. \end{cases}$$

Constraints of type (1) simply express that each task must be assigned to one and only one node and constraints of type (2) requires that the node capacity is not exceeded.

This generalized quadratic assignment problem is straightforwardly NP-hard in the strong sense notably by restriction to the Node Capacitated Graph Partitioning Problem [1] (arbitrary network topology and bandwidths as well as equidistant nodes), to the Quadratic Assignment Problem (in the case where the capacity constraints allow to assign one and only one task per node and where the internode distance is arbitrary) as well as to the bin-packing problem. This

978-0-7695-4676-6/12 $26.00 © 2012 IEEE
DOI 10.1109/IPDPSW.2012.221

list emphasizes the numerous sources of difficulties when tackling this problem.

In terms of instance size, in our application context, we have to be able to map networks with a few thousands of tasks on architectures having a few tens of nodes. Such an order of magnitude rules out exact resolutions methods: the best known methods for the node capacitated graph partitioning problems are limited to graphs with a few hundreds of vertices, and the best known algorithms for the QAP are limited to instances of size around 30 [4]. Due to the complex structure of the problem, induced by its numerous facets, designing specialized heuristics is both difficult and risky. On the contrary, it can be expected that general purpose metaheuristic design paradigms, simulated annealing for instance, will lead to more robust (although more computation intensive) algorithms in particular with respect to the stability of solution quality.

Furthermore, completely solving the DPN mapping problem also requires the calculation of the effective routing paths across the packet network. Taken independently, this problem is a minimum cost multi-flow with additional constraints which render it NP-hard in the strong sense (by restriction to 3-partition as well as to the directed edge-disjoint path problem [3]). Still, for practical processor architectures, the instances are of moderate size and can be solved to optimality using off-the-shelf MILP solvers. In the present paper, we ignore this part of the problem. It should however be emphasized that building a tractable mathematical model including both mapping and routing path calculation is difficult and that our intent is to view the latter problem as a slave problem of the former. This will incur a non negligible increase in the computational cost of the whole procedure but this will also coarsen the grain of the per solution calculations which is a good thing with respect to parallelism efficiency.

B. Algorithmic approaches

The DPN mapping problem crops up in the context of building a dataflow compilation chain for parallel microprocessor architectures targeting the embedded market. In this context, as already stated in the introduction, a number of hard discrete optimization problems must be solved. However, the operations researcher cannot ignore the peculiarities of this context when designing appropriate resolution algorithms. Indeed, the development cycle of an embedded application requires a short programmer/target feedback loop at the beginning of the development cycle where the programmer's intent is to obtain a first working version of his application and to get its coarse-grained structure right. On the contrary, towards the end of the development cycle, it is usual to invest both human and computing time (compilation time for say up to one night can often be afforded at that point) for both fine tuning and fine-grained optimizations. Thus, whereas the beginning of the cycle requires fast heuristics and can do with solutions of moderate quality, the use of more computationally intensive algorithms as well as more powerful computer systems (for compilation) is desirable at the end of the cycle.

Therefore, the DPN mapping problem needed to be initially tackled using fast algorithms. In order to do so, the overall problem was decomposed in two problems which were solved in sequence: a first GRASP-like algorithm was used to partition the process network graph and those partitions were subsequently assigned to the nodes by solving a (relatively) small size QAP by means of simulated annealing [5], [6]. Of course, an approach by decomposition disrupts the problem structure in the sense that it unfairly favours the first objective function which consists in minimizing the network (i.e., internode) bandwidth irrespective of the routing distance. Thus, although this initial approach is suited for the beginning of the development cycle of an embedded application, global resolution methods are complementarily needed for better optimization towards the end of this development cycle. For these reasons, as well as for those discussed in the preceding section, we have designed a parallel simulated annealing(-like) procedure for the DPN mapping problem which is described in the next section.

C. Parallel Simulated Annealing

The literature regarding parallel metaheuristics is particularly large. However, only a very small number of works specifically deal with multi-threaded or parallel simulated annealing. Ram *et al.* [9] propose two distributed methods in which n initial solutions are generated in a first collaborative phase, then n sequential simulated annealing searches are performed in parallel. Those methods are compared by solving instances of the job shop scheduling and traveling salesman problems. Lazarova [10] implemented a hybrid distributed and dual-threaded simulated annealing method in which the processors manage a part of the current solution. The method is applied to instances of the room assignment problem. Finally, Safaei *et al.* [11] presents a multi-threaded simulated annealing method for a real bi-objective maintenance scheduling problem with conflicting objectives. This method consists of two threads, each optimizing one of the objectives.

III. A PARALLEL SIMULATED ANNEALING ALGORITHM

A. Architecture-awareness

The target system is a Non-Uniform Memory Access (NUMA) Dell server with four 2.1 GHz AMD Opteron 6172 processors, 64 gigabytes of RAM, running SUSE Linux Enterprise server 11.1. Each of the processors consists of two NUMA nodes of six cores each, giving us a total of 48 available cores on the whole system. The communication channel between the processor sockets is a ring network of HyperTransport links. Each of the sockets is connected to the two NUMA nodes of the local processor using a pair of

HyperTransport links. The six cores in each NUMA node have direct access to a shared memory bank of 8 gigabytes per node.

As it is a NUMA system, the whole memory is addressable by any of the cores of the system. A memory access (read or write) of one core to a memory area located on its own NUMA node (a *local access*) directly passes through the local memory controller of the node and therefore is considered optimal. A memory access from one core to a memory area on a different node (a *remote access*) requires the data to be transferred through the HyperTransport network, resulting in poorer performance. Therefore, the performance of memory accesses is optimized if each core pererably accesses memory areas which are located on its NUMA node. Randomly accessing the whole memory by all cores without taking the system memory topology into account would lead to heavy use of the HyperTransport interconnection, hence substantially low overall performance.

We developed our algorithm using the POSIX threads application programming interface, which is suitable for shared memory architectures. We made use of the Portable Hardware Locality (hwloc) software package, which allows to obtain precise information about the communication topology between cores and memory. It also allows to have accurate control on how threads are assigned to cores, as well as how memory areas are assigned to NUMA nodes.

In our implementation, each thread is assigned to a dedicated CPU core, and is not allowed to migrate from one core to another. Each thread allocates memory on the NUMA node of the core it runs on. We also make use of local copies of the constant instance data on each NUMA node, in order to minimize unnecessary communication on the HyperTransport interconnection. More generally, we make use of resource locality whenever possible.

We also tried to reduce the number of locks and mutual exclusion, as they cause very important bottlenecks. Modern processors allow atomic operations on variables in memory, which are accessible through compiler *intrinsics*, which are function-like calls, but actually are compiled into a specific CPU instruction. They can be used to avoid using locking or mutual exclusion when manipulating simple shared variables like counters and pointers. Among the available atomic operations, we make use of the *add-and-fetch* and *compare-and-swap* operations. Add-and-fetch atomically adds a value given as argument to an integer variable in memory (*ie.*, its address is contained in a pointer), and returns the new value. It is useful for maintaining shared counters. Compare-and-swap atomically compares the value of a variable in memory with an argument value, and if the two values are equal, the variable contents is assigned a new value given as second argument. It returns a boolean value which is true if the comparison was successful, and therefore the new value was assigned. We use this to maintain a shared pointer to the current solution. The technique consists in first reading the current shared value, then use compare-and-swap to attempt an update of the old value with the new one. If compare-and-swap was successful, then we are done. Otherwise, we have to decide if the new shared pointer still must be updated or not with the local value.

It is important to note that even though the atomic intrinsics allow lockless management of shared data structures, a performance bottleneck still may occur if data locality is not taken into account. Atomic operations make heavy use of the memory bus, so it is never a good idea to let all threads constantly manipulate a single shared global variable.

Finally, we also had to solve a race condition when deallocating obsolete solution data structures. Indeed, as we use no mutual exclusion when reading and updating shared pointers to the current solution, a race condition could occur if a thread reads the pointer to the shared solution and starts working on the data it points to, and if at the same time, another thread updates the solution with a new one, and decides to deallocate the memory area assigned to the old solution data. This is solved by using reference counting, and using a read-copy-update mechanism for managing the deallocations. Reference counting implies managing a counter for each instance of a shared data structure. The reference counter for a shared data structure is initialized to 1 when it is designated as the new current solution. If the pointer is updated, the reference counter of the previous data structure is decremented. If a thread wants to use an instance of the solution data structure, it first must increment the reference counter for this instance; this is done atomically using add-and-fetch. Then, when the thread has finished working on a solution structure, it atomically decrements the reference counter using add-and-fetch with value -1, and if the returned value is zero, then no other thread is supposed to be using that solution structure, so the data may be deallocated. The problem is that reading the shared solution pointer then incrementing the reference counter must be done atomically, otherwise the structure may be deallocated before the reference counter is incremented. For this, we use a very simple implementation of the read-copy-update (RCU) mechanism to manage the deallocations. When a thread wants to access the shared solution, it first atomically increments a shared counter associated to the shared pointer to the solution data. Then, it can read the shared pointer value and increment the reference counter for the data structure it points to. It can now atomically decrement the shared counter associated to the shared pointer. When a thread wants to write a new value to the shared pointer, it first gets the previous pointer value and sets the new one, using the compare-and-swap mechanism described above. Then it busy waits for the value of the shared counter to reach the value 0. From this point we are sure no thread is currently attempting to access the solution data we want to release. Whenever a thread reads the shared solution data pointer, it will access another structure, since the pointer has

been updated; thus we can safely decrement the reference counter for the previous solution data structure, potentially deallocating it if the reference count reaches zero.

Please note that the RCU implementation we use is very simplified compared to more classic implementations. We replaced the grace period mechanism to check for the possibility of deallocations with a simpler mechanism using a shared counter. We also replaced the RCU callback mechanism, which is supposed to manage the deallocation after a grace period, with a simple busy-waiting loop for the counter to reach zero. This has shown to be working very well when solution structures are shared between up to 8 threads, with less than 0.1% of the overall CPU time spent in busy-waiting.

B. Structure of the algorithm

Simulated annealing (SA) is an iterative process, which step-by-step explores solutions by randomly generating a new solution from a current solution, using a neigbourhood function. Each generated solution is sumbitted to a probabilistic test based on the solution value (or energy) variation Δ_E and a current temperature value T. The acceptance probability is 1 when $\Delta_E < 0$, meaning the solution value is decreasing, and it is $e^{-\frac{\Delta_E}{T}}$ otherwise. If the test is successful, the new solution becomes the current solution from which new solutions are generated and tested.

To take profit from parallelism, we had to break that process into parallel generation of neighbour solutions. Typically, all threads in a NUMA node create new solutions from a shared solution in the local memory of that node. Once an accepted new solution has been found, it has to be decided if this new solution becomes the new shared current solution. The method we employed is based on a timestamp value we manage in solutions, in the following way: the initial solution has a timestamp value of 0, then each newly created solution is assigned a timestamp value equal to the timestamp value of its predecessor plus one. Thus, one can consider a timestamp difference between two solutions as a distance measure between those solutions. As we want to mimic as much as possible the behaviour of a sequential SA, we favour new solutions with a greater timestamp value, *ie.*, with further distance from the initial solution. We therefore allow to change the shared solution only if the new solution has a greater timestamp than the previous shared one.

The management of the global shared solution is as follows: one thread by NUMA node is designated the *leader* thread for that node. After the leader thread has updated the shared solution on its NUMA node, it also tries to update the global solution using the same timestamp criterion as above. Symmetrically, when the leader thread attempts to get the latest shared solution, it first fetches the global solution and eventually updates the node-shared solution, still based on this timestamp criterion.

The algorithm maintains global values for the current temperature, the best solution value, and the worst solution value. These values are used by the cooling scheme and the stop criterion.

The parameters for our algorithm which determine the initial temperature, the cooling scheme and the stop criterion are inspired from [7], see also [8].

As initial temperature, we chose the value for a solution we obtained using a first-fit greedy algorithm for the bin-packing problem.

The number of steps in each temperature level is the number of tasks to be placed.

When the number of steps is achieved by the threads, one thread resets the step counter to zero, and updates the temperature.

From temperature level k with temperature value T_k, we compute the temperature for level $k+1$ with the fomula:

$$T_{k+1} = \frac{T_k}{1 + \frac{\log(1+\delta)}{e_P+1} T_k}$$

where δ is a small positive number (we set it to 0.05), and e_P is the value for the worst solution of the problem. Typically, the value for the worst known solution may be used.

The algorithm is stopped when the temperature reaches

$$T_f = \frac{\beta(e_P - e)}{|T||N| \log 2 - 3}$$

where β is an expected difference ratio between the solution obtained by the algorithm and the optimal solution, $|T|$ is the number of tasks, and $|N|$ is the number of nodes. Experience showed that small values for β such as 0.05 tend to make the end of the algorithm execution spend a lot of time without finding any better solution; we use the value 0.2 which tends to reduce execution time while still obtaining the best solution values we find when $\beta = 0.05$.

For more information about SA parameter tuning, please refer to [7].

C. Positioning versus sequential execution

This parallel algorithm clearly does break some basic aspects of the traditional sequential simulated annealing. The SA process is often considered equivalent to a random walk into the solution space, which is dependent on the current temperature level. At fixed temperature, the SA algorithm executes a random walk which simulates an homogeneous Markov chain. Once the stationary distribution of that chain is assumed to be achieved, the temperature is updated (decreased) using the cooling schedule and a new random walk begins.

In the opposite, our algorithm performs parallel steps from the same starting point, so this is a different process. However, we have to keep in mind that the practical performances of the SA algorithm is not too sensitive with respect to the order in which the solution are explored

Table I
INSTANCES OF THE PROBLEM

Instance	#tasks	#nodes	node capacity
12×12	144	4	40
18×18	324	9	40
23×23	529	16	40
31×31	961	25	40
37×37	1369	36	40

despite of the fact that it disrupts some known sufficient conditions for convergence of the algorithm. Thus, we can assume that if the number of solutions explored is high enough, the parallel and sequential exploration methods can be considered equivalent.

Note that if our algorithm is run on a single thread, the corresponding random walk is exactly that of a sequential SA algorithm. When the number of parallel threads is large, the process is drastically different. However, at early stages of the algorithm, the temperature decreases quickly to achieve temperature levels for which the probability for a new solution to be rejected is very close to one. Most of the solution tests will be rejected, no matter the level of parallelism.

IV. EXPERIMENTAL RESULTS

We tested our parallel SA algorithm on the 48-core Dell NUMA server we described above. Proper knowledge of the architecture allowed us to obtain satisfactory results when using all 48 cores in a single parallel search procedure.

A. Instances

The instances we used in this article are square grids of tasks we map on a square torus network of nodes. Only one resource is taken into account. Each task takes one resource unit, meaning the resource constraints limit the number of tasks assigned to a node.

Table I lists the different instances we worked with. The name of each instance corresponds to the grid size of the task layout. The node layout is a square torus, hence the number of nodes in all instances is a square value.

For each pair of tasks (t, t'), the bandwidth $q_{tt'}$ is set to 1 if tasks t and t' are adjacent in the task grid, and 0 otherwise. For each pair of nodes (n, n'), the distance $d_{nn'}$ is the Manhattan distance between nodes n and n'.

B. Computational results

We run our algorithm on the instances using different numbers of cores. Several runs were performed for each instance and number of cores, in order to extract statistical information about average solution time, average solution value and best solution value among the different runs. The statistics for instances 12×12 to 23×23 were obtained using 10 runs. Instance 31×31 was run 8 times for numbers of cores of 6 and above. Instance 37×37 was run 4 times using between 12 and 48 cores.

Table II
AVERAGE SOLUTION TIMES (SECONDS)

	Cores used						
Instance	**1**	**2**	**4**	**6**	**12**	**24**	**48**
12×12	4.26	2.77	1.66	1.43	0.83	0.50	0.37
18×18	92.5	52.7	27.5	19.3	10.5	5.81	3.65
23×23	589.6	324.4	164.6	113.0	58.3	31.8	18.9
31×31	-	-	-	1090	561	291	156
37×37	-	-	-	-	2428	1230	654

Table III
AVERAGE SPEEDUP VS. SEQUENTIAL TIME

	Cores used					
Instance	**2**	**4**	**6**	**12**	**24**	**48**
12×12	1.54	2.57	2.98	5.13	8.50	11.5
18×18	1.76	3.36	4.80	8.78	15.9	25.4
23×23	1.82	3.58	5.21	10.11	18.5	31.2

Table II lists the average execution times of our solver on all instances, when using different numbers of cores. Runs using 1 to 6 cores always use the cores inside a single NUMA node. Twelve-core runs use the cores in the two NUMA nodes inside the same processor, and 24-core runs are executed on the 24 cores of two 12-core processors. Execution times for the biggest instances (31×31 and 37×37) are only shown using parallel execution. Table III lists the corresponding speedups compared to single-threaded execution, on instances for which sequential execution could be done in affordable time.

One can observe significant speedups when comparing the sequential and parallel execution on the three smallest instances. A trend appears, tending to show that larger instances benefit more from a larger number of cores (31.2 speedup on 24 cores for the 23×23 instance). This shows that efficient results can be achieved when taking profit from the knowledge of the memory and processor topology of such NUMA architectures.

We also compared the results in terms of quality. Average solution values are listed in Table IV. Minimum solution values are shown in Table V. Table IV clearly shows that when the number of used cores increases, the average quality of the obtained solutions tends to get worse and worse, even though the deviation is lower than 10% between sequential execution and 48-core execution.

We can observe from Table V that even though the minimum solution value between consecutive runs is significantly lower than the average value, this difference is below 10%, and is not higher than what could be expected from any type of metaheuristics, even sequential.

V. CONCLUSION

We have presented a parallel simulated annealing approach to solve instances of the DPN mapping problem. We shown that efficient speedups can be achieved using all 48 cores of the NUMA server we used.

Table IV
AVERAGE SOLUTION VALUES

Instance	Cores used						
	1	**2**	**4**	**6**	**12**	**24**	**48**
12×12	24	24	24	24	25	25.1	25.9
18×18	81	81.4	81.9	83.1	84	84.3	88.3
23×23	160	160.5	156.6	158.4	161.3	169.3	173.3
31×31	-	-	-	326.1	329.7	344.7	346.3
37×37	-	-	-	-	508.5	509.5	536.6

Table V
MINIMUM SOLUTION VALUES

Instance	Cores used						
	1	**2**	**4**	**6**	**12**	**24**	**48**
12×12	24	24	24	24	24	24	24
18×18	78	78	79	80	80	81	80
23×23	154	154	148	152	146	153	163
31×31	-	-	-	298	298	317	314
37×37	-	-	-	-	485	494	509

Even though the quality of the solutions tends to decrease when the number of cores is large, our results still show that one can take profit from parallelism to much faster obtain good solutions.

The deviation of the solution quality is certainly due to the fact that our algorithm clearly breaks the behaviour of the sequential SA, especially when the solution is updated often, which is the case at early stage of the execution, when the temperature is high. However, the temperature drops relatively fast, to reach a point where almost every newly generated solution is rejected; in that case, sequential or parallel execution is equivalent. One solution to this solution quality problem could then be to increase the number of solution generation steps between temperature decreases at high temperatures, at the cost of slightly longer execution time, hence lower speedup.

Another perspective, as already stated in Sect. II-A, consists in taking the routing path calculation problem into account within our algorithmic framework. Since that problem will be viewed as a slave problem i.e., it will be solved either approximately or exactly for each admissible mapping and the resulting economic function will be taken into account in the global objective function, this will be done without much destructuring of the algorithm. However, although this inclusion can be expected to significantly increase the overall calculation time, it can also be expected to increase the efficiency of our parallelization scheme. Also, another perspective, consists in generalizing our algorithmic framework to cope for the stochastic nature of the tasks weights (which depends in part on computing kernels execution times which are themselves random variables) along the line of the generalization of the GRASP-like heuristic for the DPN mapping problem presented in [6]. Again, this can be expected to induce a non negligible supplementary computation cost, little destructuring and an increase in parallelism efficiency. Whether such a complete algorithm for the global stochastic DPN mapping problem is practical remains of course to be experimentally assessed.

REFERENCES

[1] C. E. Ferreira, A. Martin, C. C. de Souza, R. Weismantel and L. A. Wolsey (1998): "The node capacitated graph partitioning problem: a computational study", Mathematical Programming 81, pp. 229-256.

[2] T. Goubier, R. Sirdey, S. Louise and V. David (2011): "ΣC: a programming model and langage for embedded many-cores", LNCS 7016, pp. 385-394.

[3] B. Korte and J. Vygen (2006): "Combinatorial optimization", Springer.

[4] E. M. Loiola, N. M. N. de Abreu, P. O. Boaventura-Netto, P. Hahn and T. Querido (2007): "A survey for the quadratic assignment problem", European Journal of Operationnal Research 176, pp. 657-690.

[5] R. Sirdey (2011): "Contributions à l'optimisation combinatoire pour l'embarqué : des autocommutateurs cellulaires aux microprocesseurs massivement parallèles", HDR Thesis, Université de Technologie de Compiègne.

[6] O. Stan, R. Sirdey, J. Carlier and D. Nace (2012): "A heuristic algorithm for stochastic partitioning of large process networks", submitted for publication.

[7] R. Sirdey, J. Carlier and D. Nace (2009): "Approximate solution of a resource-constrained scheduling problem". J. Heuristics 15, pp. 1-17.

[8] P. J. M. van Laarhoven and E. H. L. Aarts (1987): "Simulated annealing: theory and applications". Kluwer Academic Publisher.

[9] D.J. Ram, T.H. Sreenivas and K.G. Subramanian (1996): "Parallel simulated annealing algorithms". Journal of Parallel and Distributed Computing 37 (2), pp. 207-212.

[10] M. Lazarova (2008): "Parallel simulated annealing for solving the room assignment problem on shared and distributed memory platforms", ACM International Conference Proceeding Series Vol. 374(2), Proceedings of the 9th International Conference on Computer Systems and Technologies and Workshop for PhD Students in Computing, Gabrovo, Bulgaria, pp. 1-6.

[11] N. Safaei, D. Banjevic and A. K.S. Jardine (2011): "Multithreaded simulated annealing for a bi-objective maintenance scheduling problem". International Journal of Production Research, DOI:10.1080/00207543.2011.571444, pp. 1-19.