Parallel CPU and GPU computations to solve the job shop scheduling problem with blocking

Abdelhakim AitZai
Faculty of Electronics and Computer Science
University of Sciences and Technology HOUARI BOUMEDIENNE
Algiers, Algeria
h.aitzai@usthb.dz

Adel Dabah
Faculty of Electronics and Computer Science
University of Sciences and Technology HOUARI BOUMEDIENNE
Algiers, Algeria
dabah.adel@gmail.com

Mourad Boudhar
Faculty of Mathematics
University of Sciences and Technology HOUARI BOUMEDIENNE
Algiers, Algeria
mboudhar@usthb.dz

***Abstract*—In this paper, we studied the parallelization of an exact method to solve the job shop scheduling problem with blocking JSB. We used a modeling based on graph theory exploiting the alternative graphs.**
We have proposed an original parallelization technique for performing a parallel computation in the various branches of the search tree. This technique is implemented on computers network, where the number of computers is not limited. Its advantage is that it uses a new concept that is the logical ring combined with the notion of token. We also proposed two different paradigms of parallelization with genetic algorithms. The first uses a network of computers and the second uses GPU with CUDA technology. The results are very interesting. In addition, we see a very significant reduction of computation time compared to the sequential method.

***Keywords*—Job shop; blocking; parallelization; branch-and-bound; network**

I. INTRODUCTION

The scheduling problem with blocking can be defined as follows: let n be the number of jobs each job k of which is composed of k_i (i=1,…,n) operations which have to be processed on m different machines with no storage space and according to a given order. Each operation has to be processed on a specified machine without going back to the same machine and each machine has to process one operation at a time. The processing time of each operation on each machine is given and the interruption of the processing of any operation is not permissible. Taking into account all these constraints we try to find a scheduling of all operations such that the end date of the last operation is minimized. In other words, the job shop problem with blocking (JSPB) is a version of the job shop scheduling problem with no intermediate buffers, i.e. where a job has to wait on a machine until it gets processed on the next machine.

Several papers have tackled the resolution of this problem but its NP-Hardness, described in detail by Hall and Sriskandarajah [11], is the real obstacle to find an exact solution. Moreover, researchers have not succeeded in finding an exact method to solve the JSPB with more than 10jobs×10machines. For this reason, they have turned their attention to the investigation of meta-heuristics which, while they do not guarantee to find the optimum, can in general reach good solutions in real time.

The majority of work found in the literature considers the flow shop problem with blocking (FSPB), while the number of papers that have tackled the JSPB is minimal (Gröflin and Klinkert [9]; Mascis and Pacciarelli [14]; Pham and Klinkert [17]. Since the publication of the first work dealing with the JSP, a great effort has been made for the design of branch-and-bound (B&B) algorithms. Among others, we can cite Carlier and Pinson [4] algorithm which solves the problem with complexity 10×10 proposed by Fisher and Thompson [7]. Then, Carlier and Pinson [5] introduced the possibility of fixing the alternative arcs without branching, which made their algorithm very efficient. The majority of subsequent works were based on their results. As for the parallel approach, we find the parallel implementation of the B&B method proposed by Perregaard and Clausen [16]. This algorithm is based on the results of Carlier and Pinson [6] and Brucker [4]. Ben Mabrouk [3] has proposed a parallelization of a hybrid genetic–tabu search-based algorithm for solving the graph coloring problem.

This paper tries to contribute to these efforts by investigating the performance of a parallel B&B procedure in the resolution of the problem, along with a parallel Genetic Algorithm GA using CUDA GPU. The idea of moving towards the parallelization of optimization methods is not likely to solve the problem of very long execution times (for big sizes); nevertheless it remains

978-1-4799-1365-7/13/$31.00 ©2013 IEEE

interesting to explore it with new techniques. These methods can be divided in the literature in two well-known major classes: parallelization with multiple communications and master-slave parallelization. There are almost no parallel methods in the literature for solving the JSPB; as mentioned above, even the resolution of the problem, sequentially, is very limited in comparison with the resolution of the FSPB.

In this paper, we first model the JSPB using the alternative arcs already known in the literature Mascis and Pacciarelli [14]. We propose a parallel B&B method based on implicit enumeration (the sequential version is detailed in AitZai [1], which uses the hybrid master-slave principle. Then, we develop two parallelization methods based on the master-slave approach applied to Genetic Algorithm GA, the first uses a computer network and the second uses graphics processors GPU under CUDA.

Work on the parallelization of GA is scarce. Indeed, several parallelization techniques are proposed in the literature, especially for genetic algorithms, such as fine-grained methods (Gorges-Schleuter [8]; Manderick and Spiessens [13]; Mühlenbein [15]) and the parallel hierarchical methods (Gruau [10], but papers on the parallel GA using GPU are limited.

The rest of this paper is organized as follows: Section 2 introduces the JSPB modeling using alternative graphs. Section 3 presents the parallel B&B method we propose. In sections 4 and 5 we give the details of our parallelization methods based on CPU and GPU models. Comparison and analysis of the results produced by the proposed methods is given in Section 6. Finally, Section 7 provides a general conclusion and gives some prospects for future work.

II. PROBLEM FORMULATION

This section is dedicated to the modeling of the JSPB using alternative graphs as detailed in Mascis and Pacciarelli [14].

The simple job shop scheduling problem is usually modeled in two different ways: either by linear programming or conjunctive graphs. In the latter, the orientation of the conjunctive arcs defines the order in which the various operations are performed on any given machine. With the advent of the blocking constraint, the latter could not be modeled by conjunctive graphs; this is why another modeling approach is used, still with graph theory. This new modeling is called alternative graphs. In this type of graphs, a new concept, alternative arcs, is introduced. Two operations connected by an alternative arc, whose weight is zero, means that they can start at the same time, contrary to disjunctive arcs that define an order relationship between the operations. In this modeling approach, the vertices of the alternative graph represent the operations and the defined arcs are of two types: the conjunctive arcs (the arcs of priorities between the operations of the same job) and the alternative arcs.

III. PARALLEL B&B METHOD

In this section, we reused the parallel B&B method proposed by AitZai and Boudhar [2]. The implementation of the B&B method on the job shop model has shown its inadequacy to process, with only one processor, problems of size higher than (10×10). This is in addition to the limits in the memory space and response time. These reasons make the design of a parallelized solution a necessity. Through this parallelization, we aim to design an algorithm that can replace the work of a powerful processor, thus ensuring a very fast information transfer between network stations.

In our work, we have used a *control* station which is in charge of the following tasks:

- Collect problem data.
- Assign tasks to other stations which will carry out the search for the optimal solution for the given problem.
- Be attentive to any new improvement and spread it to all other stations.
- Ensure the algorithm terminates.

As for the computing stations, they look for a better solution than the current one and inform the *control* station of each improvement until they find the optimal solution. The work request issued by any computing station is carried through a transfer of an offer to help to its right neighbor (but not to the controller). If not satisfied by the right neighbor, then this offer is passed on to the right neighbor of the latter until satisfaction or algorithm termination.

After a full analysis of what had been done thus far, we have proposed our parallel B&B algorithm which is independent of the network physical topology and ensures the fastest possible communication so as to improve the response time. The independence of our algorithm is ensured by the proposed logical ring topology where a logical ring may be created on any physical topology. Fast communication then requires a point-to-point contact between the controller and each of the computing stations; this is ensured by a switch. Multi-cast sending is necessary between the control station and the other stations. This condition is doable thanks to a multi-cast technology which is available in the Java language.

Fig. 1 shows our logical ring topology with all the calculating stations (computers). The creation of this logical ring will be done by the controller which sends to each station the following structure:

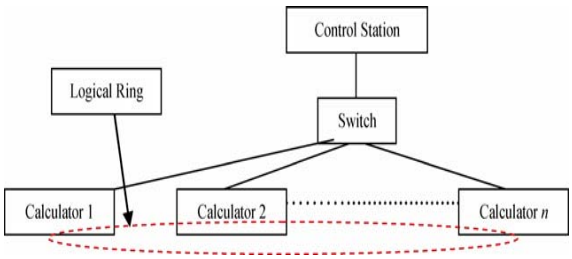
- The IP address of the controller.
- The IP addresses of the left and right neighbors of this calculator
- The information if this station is a launcher or not.

The controller randomly selects (or by means of a selection technique) a calculating station to launch the computation; this station is called the *launcher*. The controller sends the initial problem data to the launcher and it launches five parallel processes that take care of the following:

- Simulated annealing heuristic.
- Being attentive to the requests of the initial problem.
- Being attentive to new improvements
- Being attentive to the token blocking
- Being attentive to the termination of the algorithm.

As the structure is defined (i.e. the neighbors), the launcher creates a token and sends it on the network, thus signifying the beginning of the work. After receiving the token, the other calculators ask for the initial problem from the controller not from the launcher. This way of including the stations one by one aims to implement the 'use based on calculators needs' policy; hence a problem which can be computed using *k* stations will not require more stations.

Fig. 1. The logical ring topology



After receiving the initial problem, the calculator sends a work request to the right neighbor, while keeping the token to ensure that only one station requests it at a time. Two cases may arise:

- If the right neighbor has some extra work (i.e. the sub problems queue has more than one sub problem), then it directly sends half of this line up to the requestor by using its IP address which appears in the request. Once the work is received by the requestor, it lets the token free to its left neighbor thus giving it an opportunity to request the work.
- Otherwise, it passes the request on to the right neighbor and the request turns on in the logical ring until obtaining a work or the algorithm terminates.

In the classical technology of the token ring, the token has been used as a privileged access to the ring. The station receiving the token has the right to send the data if any; otherwise, it gives back the token to the ring by sending it to the left or right neighbor according to the protocol in use. We have inverted this by deciding that the station receiving the token has the right to ask for a work (and not send it) if it is at rest. The token will be created only once by the launcher, thus the possibility to have many tokens circulating in the net is eliminated. Also, when a calculating station wants to request work, it must keep the token, which makes the circulation of many requests impossible. Thus, we avoid any inconsistency concerning the token management or the job request.

IV. PARALLEL GA WITH CPU

In this section, we propose two different approaches of priority rules-based encoding GA parallelization. The first one is based on CPU networking and the second is based on GPU.

Even meta-heuristics offer efficient strategies to look for solutions in combinatory optimisation problems. The computation times associated with the exploration of the search space may be very large. One way of speeding up this exploration is the use of parallelism. A greater contribution of parallel computation to meta-heuristics is observed. By applying convenient parameters, parallel meta-heuristics may be more robust than their sequential counterparts in terms of obtained solutions' quality. Some parallel mechanisms allow reaching better solutions without requiring a larger number of operations.

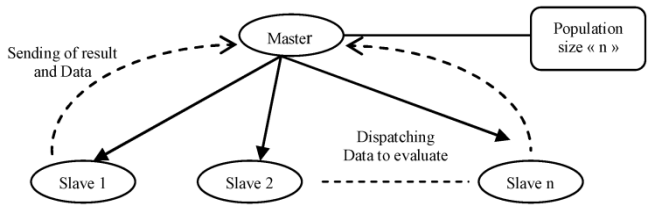
Thanks to their structure, GAs adapt very well to parallel execution. So, with the growing popularity of parallel computers, parallel versions of GAs have been introduced and are subject to much research.

Parallelisation of GAs can be implemented in two ways, first through populations with multiple communications and the second through the master–slave method with a single population. The two types of parallel GAs have been extensively used to reduce the execution time of many applications. The choice between the two parallel methods is determined by several factors, such as ease of use or execution and their potential to reduce the runtime. Generally, parallel master–slave GAs are the easiest to implement, because it is easier to configure their control parameters and their implementation is easier. In contrast, GAs with multiple communication populations enable more parallelism but are more difficult to implement because, in this type of parallelism, it is necessary to set the values of several additional parameters, besides the usual parameters of the GA itself, which affects the efficiency and quality of solutions found by these classes of methods.

The easiest way to parallelise a GA is to distribute the evaluation of individuals on multiple slave processors or GPU processors, while a master performs the basic operations of the GA (selection, crossover and mutation); see Fig.2. The master–slave parallel GA is effective for several reasons:

- It explores the search space in exactly the same way as a sequential GA, and therefore, the base structures used in the design of the GA sequence remain valid.
- Its implementation is very simple.
- In many application cases, it provides significant improvements to the end results.

Fig. 2. Model of computation parallelisation



We have used for our CPU parallelisation [2] a star network of inter-connected computers. This allows us to ensure communication between devices either in device-to-device (direct communication between two stations) or in multi-cast which allows to broadcast information on the entire network. Our parallelization technique is mainly based on a master–slave model. So, we have chosen a master computer to generate populations of individuals using selection, crossover and mutation. Then the individuals are sent to the slave stations to carry out any calculation of fitness. This way, all the hard work of calculating the fitness is shared amongst all the network computers.

This parallelisation method keeps all the characteristics of the sequential GA, without any changes to the already set parameters. The only parameter that changes in the CPU parallel algorithm is the size of the population which directly depends on the number of computers on our star-shaped network, since the master computer divides the population during processing into a number, i.e. proportional to the number of computers that are available on the network, then it sends the sub-populations to the various slave computers where the calculation of the fitness is carried out. In the next section we will present another parallelization paradigm which is based on the exploitation of the graphics processor (GPU).

V. PARALLEL GA WITH GPU

The growth of the video game industry and multimedia applications has prompted manufacturers to produce more efficient graphics processors in terms of computing power. This development follows a different way than the development of CPU processors. This discrepancy can be explained by the difference that exists between the categories of problems to deal with. Indeed a CPU aims to complete a task as quickly as possible against a GPU, which performs a similar treatment on multiple data during the same time and as fast as possible.

The use of this computing power outside the multimedia field is called GPGPU (General Purpose Graphics Computing Unit). To take advantage of this computing power, manufacturers have introduced tools to exploit these resources. Thus NVIDIA proposed an environment of a parallel computing architecture, which uses the parallel computing capability of NVIDIA GPUs, called CUDA (Compute Unified Device Architecture). It includes a compiler and a set of development tools, which can be used with the C language to write code designed to run on the GPU. To understand the idea of such an environment we must first introduce the hardware and software models. Unlike CPUs, graphics processors are optimized to perform the same treatment on a large amount of data. This difference exists both in the memory system and in the execution of the instructions.

GPUs are built in a scalar manner i.e. they have several computation units, which can carry a limited number of instructions in the same time. A GPU includes several independent multiprocessors; each multiprocessor contains 8 Scalar Processors SP, two specialized processors and a shared memory. To manage multiple threads running several programs, the multiprocessors use a new architecture called SIMT (Single Instruction Multiple Thread) unlike SIMD (Single Instruction Multiple Data) used in early versions of CUDA. These multiprocessors connect each thread to a scalar processor SP. Then, each thread can be run independently with its own instruction address and its own registers. SIMT Multiprocessors create, manage and launch instructions on groups of 32 elements, each element being called a thread (not to be confused with a CPU thread) and each group of 32 elements is called a warp. Each multiprocessor is composed of:

- A set of 32-bit registers.
- A shared memory cache.
- A constant memory cache that is shared by all SPs (accessible in read-only).
- A cache read-only access memory called texture that is also shared by all SPs; each access to this memory area is via specific textures units.

Hereafter, we try to identify the parts of the parallel GA algorithm that can be delegated to the GPU. First, we have used the same GA parallelization idea (Master/Slave) described in the beginning of this section, i.e. which delegates the decoding of solutions and the computing of chromosomes fitness to the GPU.

To do this, we must define the main needed memory space:

- A Storage space for the operations processing time vector. The size of the latter is n which represents the number of operations. Hence, the i^{th} vector cell represents the processing time of the i^{th} operation.
- A Storage space for the machines vector. The size of the latter is n. Hence, the i^{th} vector cell contains the machine number which will execute the i^{th} operation.
- A Storage space for the chromosomes vector (matrix chromosomes), containing the chromosomes to be treated on the GPU for decoding and evaluation.

Another storage space is needed for decoding and evaluating solutions:

- An $m \times nbc$ storage space containing, for each chromosome, the list of the ready operations that are to be performed,

where, *m* is the number of machines and *nbc* the number of chromosomes.

- An *m*×*nbc* storage space containing, for each chromosome, the status (available or not) of each corresponding machine.
- An *m*×*nbc* storage space containing, for each chromosome, the list of the blocking operations on each machine.
- An *m*×*nbc* storage space containing, for each chromosome, the available operations to perform.
- An *m*×*nbc* storage space needed for each chromosome computing evaluation.
- An *n*×*nbc* storage space containing, for each chromosome, the scheduled operations sequence.
- An *nbc* storage space containing, chromosomes evaluation.

As we can see, the algorithm requires much memory resources. In addition, memory accesses during the solution construction phase are very important; this can lead to a long execution time. To solve this problem and reduce memory access while optimizing bandwidth, we have followed the various optimization strategies described in the CUDA programming guide. So we have made the following choices:

The use of textures to optimize bandwidth to access the chromosomes matrix, the operation processing time vector and the machines vector.

Stocking of the remaining data in global memory.

Each thread execution is a construction of a solution. In addition, this thread can access only the corresponding storage space.

The thread data are aligned side by side. This alignment gives more opportunities for different threads to coalesced memory access so as to reduce the latency of global memory accesses.

Access to the chromosomes data (operation processing time, appropriate machine, etc.) is done through the texture units. This allows us to harness the caching technique offered by the graphics controller, and also increase virtual bandwidth.

VI. RESULTS AND DISCUSSION

We present in Table 1 the results of the parallel B&B method. The first column represents the size of the problem, where nb_jobs is the number of jobs whereas nb_machines is the number of machines. The second column gives the value of C_{max} found by simulated annealing, which is used as a starting least upper bound. The third column is the C_{max} found by our parallel B&B method. The run time of the sequential and the parallel B&B can be found in the fourth and fifth columns. In the sixth column, we show the number of calculators used.

We can clearly see the gain in response time for more and more complex examples. This is seen even better, e.g. 5×10: only 22.92 (s) for 3 calculators and 6.04 (s) for 4 calculators, whereas it was 77.25(s) in the sequential case. The largest gain in execution time is evident in the examples 6×5, 9×5 and 6×10 that have times 168.8 (s), 13,932.00 (s) and 34,920.67 (s), respectively, in the sequential case, while these times are reduced to: 22.53 (s) with 3 calculators; 1,501.25 (s) with 3 calculators and 4,325.67 (min) with 4 calculators, respectively.

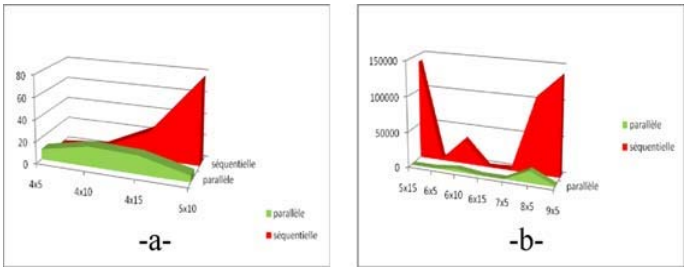
One can easily notice that our parallel algorithm accelerates the solving process with the increase in the number of calculators, unlike other parallel architectures (e.g. master- slave which get heavier).

The most interesting point is that the calculation gives sequential times smaller than in the parallel case (Fig. 3.(a)) for small examples (4×5, 4×10) which can be explained by the time taken for message exchanges between stations. This exchange time is negligible when processing large examples (Fig. 3. (b)). So for small examples, it is preferable to use a sequential calculation. We can also see that increasing the number of positions likely improves the response time of our B&B parallel algorithm.

TABLE I PARALLEL B&B RESULTS

Nb jobs × Nb machines	Cmax	Sequential running time (seconds)	Parallel running time (seconds)	Number of PC's in the network
4×5	395	188×10-3	9,547	3
4×10	633	4,875	18,360	3
4×15	955	25,578	16,656	3
5×10	681	77,250	22,922	3
5×10	681	77,250	6,047	4
5×15	806	-	87 413,531	3
6×5	427	168,8	22,531	3
6×15	1 229	26289,5	46,046	3
6×10	758	34920	4 325,672	4
7×5	638	2181	260,453	3
8×5	659	1778,4	17 139,406	4
9×5	690	13932	1 501,250	3

Fig. 3. Response time.



These results show the improvement made by the introduction of parallelism in solving this problem known in the literature to be NP-Hard.

We present in this part the gain obtained by harnessing the computing power of NVIDIA graphic cards.

The tests were performed on a station characterized by the following characteristics:

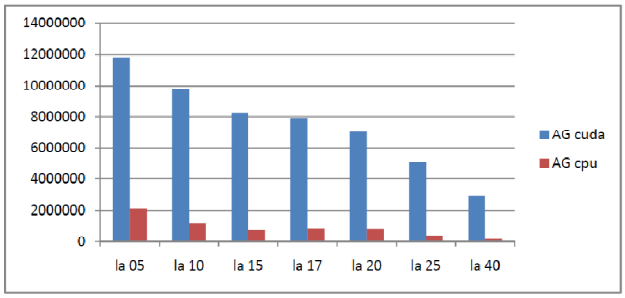
- CPU × 2: XEON E5620 2.40 Ghz
- RAM: 6GB (DDR2)
- OS: Windows Seven 64bit
- GPU: NVIDIA Quadro 2000 01 Go.

The results of our CUDA algorithm are shown in TABLE II. The first column contains the instance name, the second column the number of solutions explored and Cmax value obtained by the Genetic Algorithm in CUDA. These results are obtained using the configuration of 66 blocks with 16 threads per block (over a period of 300s) giving a total of 1056 threads. The third column represents the number of solutions explored and Cmax value obtained by the sequential Genetic Algorithm with a population size equal to 1056 and total execution time limited to 300s. We can easily see, the performance of CUDA computing in the number of explored solutions.

TABLE II. THE NUMBER OF SOLUTIONS EXPLORED BY GPU

Instances	GA using CUDA		GA Using CPU	
	Number of explored solutions	*Cmax*	Number of explored solutions	*Cmax*
La_05	11827200	735	2129920	740
La_10	9793536	1320	1167360	1272
La_15	8220672	1737	757760	1763
La_17	7913664	1087	839680	1093
La_20	7028736	1239	819200	1205
La_25	5067562	1733	389120	1730
La_40	2881536	2180	184320	2184

Fig. 4. Graphical representation of table II



CONCLUSION

In this paper, we have presented a parallel B&B method and parallel GA using CPU and GPU to solve the job shop scheduling problem with blocking. We have developed a new parallelization technique to our B&B method. This technique is based on the logical ring topology where we used a token so that the station receiving the token has the right to ask for work (and not send it). This technique ensures the independence of our algorithm from the network physical topology; it permits the fastest possible communication so as to improve the response time. We also presented two (slave/master) parallelization techniques applied on genetic algorithm, the first uses a network of computers and the second uses the GPU graphics processors under CUDA. The results obtained are very interesting.

A future improvement of our work would be to develop better lower and upper bounds for the B&B method. This will allow solving larger instances of the problem. It would also be interesting to see the effect of parallelization of the B&B method proposed by Carlier and Pinson (1994) on the considered problem.

REFERENCES

[1] A. AitZai, B. Benmedjdoub, and M. Boudhar, "A branch and bound and parallel genetic algorithm for the job shop scheduling problem with blocking", Int. J. Operational Research, Vol.14, No.3, pp.343 – 365, 2012.

[2] A. AitZai and M. Boudhar, "Parallel branch-and-bound and parallel PSO for the job shop scheduling with blocking", Int. J. Operational Research, vol. 16, No. 1, 2013.

[3] B. Ben Mabrouk, H. Hasni, and Z. Mahjoub, "On a parallel genetic–tabu search based algorithm for solving the graph colouring problem", European Journal of Operational Research, Vol. 197, pp.1192–1201, 2009.

[4] P. Brucker, B. Jurisch, and B. Sievers, "A branch and bound algorithm for the job-shop scheduling problem", Discrete Applied Mathematics – DAM, Vol. 49, Nos. 1–3, pp.107–127, 1994.

[5] J. Carlier, and E. Pinson, "An algorithm for solving the job-shop problem", Management Science, Vol. 35, No. 2, pp.164–176, 1989.

[6] J. Carlier, and E. Pinson, "Adjustment of heads and tails for the job-shop problem", European Journal of Operational Research, Vol. 78, pp.238–251, 1994.

[7] H. Fisher, and G.L. Thompson, "Probabilistic learning combinations of local job-shop scheduling rules', in J.F. Muth and G.L. Thompson (Eds.), Industrial Scheduling. Englewood Cliffs: Prentice-Hall, pp.225–251, 1963.

[8] M. Gorges-Schleuter, "ASPARAGOS: an asynchronous parallel genetic optimization strategy', Proceedings of the Third International Conference on Genetic Algorithms. San Mateo, CA: Morgan Kaufmann Publishers, pp.422–427, 1989.

[9] H. Gröflin, and A. Klinkert, "A new neighbourhood and tabu search for the blocking job shop", Discrete Applied Mathematics, Vol. 157, No. 17, pp.3643–3655, 2009.

[10] F. Gruau, "Neural network synthesis using cellular encoding and the genetic algorithm', PhD Thesis, Ecole Normale Supérieure de Lyon, 1994.

[11] N.G. Hall, and C. Sriskandarajah, "A survey of machine scheduling problems with blocking and no-wait process", Operations Research, Vol. 44, No. 3, pp.510–525, 1996.

[12] S. Lawrence, " Resource constrained project scheduling: an experimental investigation of heuristic scheduling techniques" (Supplement), Graduate School of Industrial Administration, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1984.

[13] B. Manderick, and P. Spiessens, "Fine-grained parallel genetic algorithms", in J. David Schaffer (Ed.), Proceedings of the Third International Conference on Genetic Algorithms, San Mateo, CA: Morgan Kaufmann Publishers, pp.428–433, 1989.

[14] A. Mascis, and D. Pacciarelli, "Job shop scheduling with blocking and no-wait constraints", European Journal of Operational Research, Vol. 143, pp.498–517, 2002.

[15] H. Mühlenbein, "Parallel genetic algorithms, population genetics and combinatorial optimization", Proceedings of the Third International Conference on Genetic Algorithms, San Mateo, CA: Morgan Kaufmann Publishers, pp.416–421, 1989.

[16] M. Perregaard, and J. Clausen, "Parallel branch-and-bound methods for the job-shop scheduling problem", Annals of Operations Research, Vol. 83, pp.137–160, 1998.

[17] D. Pham, and A. Klinkert, "Surgical case scheduling as a generalized job shop scheduling problem", European Journal of Operational Research, Vol. 185, pp.1011–1025, 2008.