GPU-based two level parallel B&B for the Blocking job shop scheduling problem.

Adel Dabah[1,2], Ahcen Bendjoudi[1], Didier el-baz[3], Abdelhakim AitZai[2]
[1] *CERIST Research Center 3 rue des freres Aissou, 16030 Ben-Aknoun Algiers, Algeria*
Email: {adabah,abendjoudi}@cerist.dz
[2] *University of Sciences and Technology Houari Boumedienne (USTHB) Algiers, Algeria*
Email: {h.aitzai,adel.dabah}@usthb.dz
[3] *CNRS, LAAS, 7 avenue du colonel Roche, F-31400 Toulouse, France*
Universit de Toulouse, F-31400 Toulouse, France
Email: elbaz@laas.fr

***Abstract*—Branch and bound algorithms (B&B) are well known techniques for solving optimally combinatorial optimization problems. Nevertheless, these algorithms remain inefficient when dealing with large instances. This paper deals with the blocking job shop scheduling (BJSS), which is a version of classical job shop scheduling with no intermediate buffer between machines. This problem is an NP-hard problem and its exact resolution using the sequential approach is impractical. We propose in this paper a GPU-based parallelization in which a two level scheme is used. The first level is a node-based parallelization in which the bounding process is faster because it is calculated in parallel using several threads organized in one GPU block. To fully occupy the GPU, we propose a second level of parallelization which is a generalization of the first level. Therefore, at each iteration, several search tree nodes are evaluated on the GPU using several thread-blocks. The obtained results, using the well-known Taillard instances, confirm the efficiency of the proposed approach. Also, the results show that our approach is 65 times faster than an optimized sequential B&B version.**
***Keywords*-Job shop; blocking with swap; GPGPU; parallel computing; Branch-and-Bound.**

I. INTRODUCTION

The job shop scheduling problem (JSSP) consists to schedule a set of jobs on a set of machines. Each job has its own sequence of crossing on machines. The execution of a job on a machine is called operation and each one uses the machine for uninterrupted processing time. The classical JSSP assumes an infinite storage space between machines which is not realistic. The BJSS is a version of the classical JSSP with no storage space, where a job has to wait on a current machine until the next one becomes available. Our goal is to minimize the *Makespan* (*Cmax*). The classical JSSP is known to be NP-hard in the strong sense [2], and the blocking extension of this problem BJSS appears to be even more difficult to solve [1].

This problem has several applications areas such as: manufacturing systems with no storage space, train scheduling, hospital resource scheduling, *etc*. Despite the large number of application areas and the economic impact of reducing the storage space, the BJSS has been treated by few authors using metaheuristics [3, 8, 10, 5, 9, 6] and exact methods [1, 10].

The Branch and Bound algorithms (B&B) are well known techniques for solving optimally the combinatorial optimization problems. Nevertheless, this method takes huge time to solve small instances and remains inefficient when dealing with large instances. Therefore the parallelization of this method is indispensable. A parallel implementation of this algorithm is interesting due to the nature of the B&B techniques which are suitable for parallelization.

In the literature, several GPU-based parallelization of the B&B algorithm have been proposed. Most of these works deal the flow shop problem [13,14,15], knapsack problems [18] and Traveling Salesman Problem [12]. Also, most authors take the classical approach: a single GPU thread supports the evaluation of a single node of the search tree. This approach uses a lot of GPU resources, therefore, a limited number of threads are launched in parallel.

To the best of our knowledge, our work is the first heterogeneous (CPU-GPU) implementation of B&B algorithm dedicated to JSSP and its blocking extension. We propose in this paper a two level parallelization scheme. The first level (Parallel Evaluation of one Bound) exploits the fact that the evaluation of each node can be calculated in parallel. Therefore, at each iteration one node will be sent for parallel evaluation on GPU by using one thread-block. Experiments using the Taillard instances show that this version is 7 times faster than the optimized sequential B&B version. The disadvantage of this first level is the underused of the GPU. For this reason, we propose a second level of parallelization to fully occupy the GPU. This level represents a generalization of the first level named Parallel Evaluation of Several Bounds. Therefore, at each iteration several search tree nodes will be sent for evaluation using several GPU blocks. The obtained results, using the Taillard instances, confirm the efficiency of the proposed approach and the positive impact of using parallel architectures to solve this problem. Also the results of our two level scheme show a good speed up of the execution time with 65 times faster with NVIDIA K40 GPU accelerator compared to an

optimized sequential B&B version.

The remainder of this paper is organized as follows: Section 2 describes the blocking job shop scheduling problem, the alternative graph model and related work. Section 3 contains a brief description of the sequential B&B algorithm and its components. Section 4 presents the proposed parallelization and implementation of the B&B algorithm. Section 5 discusses computational results. Finally conclusions and perspectives are presented in Section 6.

II. Blocking job shop scheduling problem

A. Problem Formulation

The classical job shop scheduling problem can be defined by a set *J* of *n* jobs $(J1, ..., Jn)$ to be processed on a set *M* of *m* machines $(M1, ..., Mm)$. Each machine can process at most one job at given time. The execution of a job on a machine is called operation. We note by O the set of operations $(o_1, ..., o_{n*m})$. Each operation o_i needs the use of a machine $M(i)$ for an uninterrupted duration called processing time p_i. Each job has its own sequence of crossing on machines which creates precedence constraints between consecutive operations of the same job. A solution (schedule) for this problem consists to assign a starting and finishing times t_i, c_i for each operation o_i $(i = 1, ..., n*m)$; while satisfying all constraints. Our goal is to minimize the *Makespan* (*Cmax*). The JSSP assumes that there is an unlimited intermediate buffer capacity between consecutive operations of a job which is impossible in real manufacturing.

The BJSS is a version of the classical JSSP with no intermediate buffers, where a job has to wait on a current machine until the next machine becomes available for processing. It can be modelled as an alternative graph model introduced by Mascis *et al.* [1] which is a generalization of the disjunctive graph of Roy and Sussman [4]. This model can be defined as $G = (N, F, A)$. N represents a set of operations with two additional dummy operations (start and finish) modelling the start and the finishing of the schedule. F represents a set of fixed arcs imposed by precedence constraints between consecutive operations of the same job and f_{qp} is the length of arc $(q, p) \in F$. Finally, A is a set of alternative pairs $((i, j), (h, k))$ which represent the processing order for concurrent operations on the same machine and a_{ij} is the length of alternative arc (i, j). Each arc represents the fact that one operation must be completed before starting the processing of other operation.

We call the last operation of each job (example o_r) an ideal operation because the machine becomes immediately available after the end of it prossessing time p_r.

If o_i is a blocking operation, we note by $\sigma(i)$ the operation immediately following o_i in the same job.

Let us consider two blocking operations o_i, o_j and one ideal operation o_r, where $M(i) = M(j) = M(r)$. Since the three operations cannot be executed at the same time, we associate with them a pair of alternative arcs.

Case1: alternative pair between o_i and o_j (Fig. 1).

If o_i is processed before o_j, and since o_i is blocking, $M(i)$ can start processing o_j only after the starting time of $\sigma(i)$(when o_i leaves $M(i)$), we represent this situation with the alternative arc$(\sigma(i), j)$ having length 0. The same for the other alternative arc$(\sigma(j), i)$ since o_j is a blocking operation.

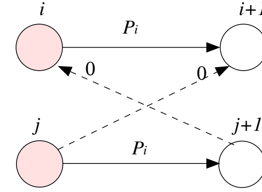

Figure 1. Alternative pair between blocking operations.

Case2: alternative pairs between o_i, o_r and o_j, o_r (Fig. 2).

The same as the first case for the alternative arcs $(\sigma(i), r)$ (a) and $(\sigma(j), r)$ (b) because both o_i and o_j are a blocking operations. The other alternative arcs depend on the fact that o_r is an ideal operation, then we add the alternative arcs $(r, i)(a)$ and $(r, j)(b)$, with length p_r.

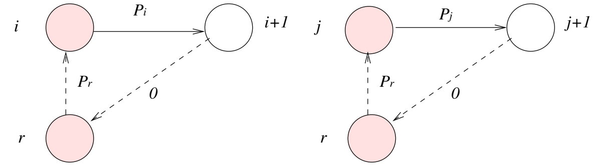

Figure 2. Alternative pairs between blocking and ideal operations.

A selection SL is a set of arcs obtained from A by choosing at most one arc from each pair, and $G(SL) = (N, F \cup SL)$ represents the obtained graph. We say that a selection SL is feasible if there is no positive length cycle in $G(SL)$ and the evaluation (*Makespan*) of SL is the longest path in $G(SL)$.

As we said that SL is a complete selection if exactly one arc is chosen from each pair, therefore $|A| = |SL|$. We define a schedule (solution of the problem) as a complete feasible selection. Finally, given a feasible selection SL, let $l(i, j)$ be the length of a longest path from operation i to j in $G(SL)$.

Table 1 represents BJSS instance with two products (jobs) and three machines. The first product $(J1)$ has 5 min processing time on machine $M1$, 3 min on $M2$ and 8 min on machine $M3$. The second product $(J2)$ has 8 min processing time on machine $M2$, 2 min on $M1$ and 7 min on machine $M3$.

Figure 3 represents an alternative graph of the BJSS instance in Table 1. This graph has three alternative pairs,

Table 1
BJSS INSTANCE WITH TWO JOBS AND THREE MACHINES.

job	sequence	processing times
$J1$	$M1, M2, M3$	5, 3, 8
$J2$	$M2, M1, M3$	8, 2, 7

two between blocking operations and one between ideal operations. Both operations 2 and 4 need the same machine $M2$ and since $M2$ can not process both operations at the same time, we associate with them an alternative pair. Since operations 2 and 4 are a blocking operations the first alternative arc (*3*, *4*) represents the choice that operation 2 must be finished before the beginning of operation 4. His mate, arc (*2*, *5*) represents the choice that operation 4 must be finished before the beginning of operation 2. We use the same process to generate the alternative pair ((*2*,*5*),(*6*,*1*)) between operations 1 and 5. The alternative pair between

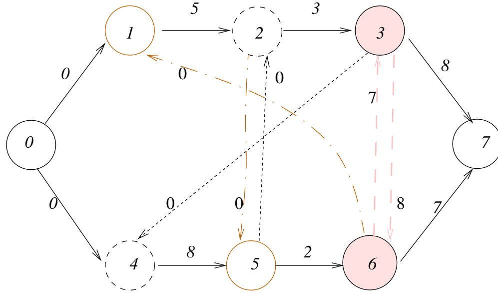

Figure 3. Alternative graph for BJSSP instance of table 1.

operations 3 and 6 is ((*3*, *6*)(*6*, *3*)) because both operations 3 and 6 are ideal.

Figure 4 represents a feasible schedule (solution) for the BJSS instance in Table 1, obtained by choosing one arc from each pair in the alternative graph in Figure 3. The *Makespan* (*Cmax* = 26) of this schedule is the longest path in the obtained graph.

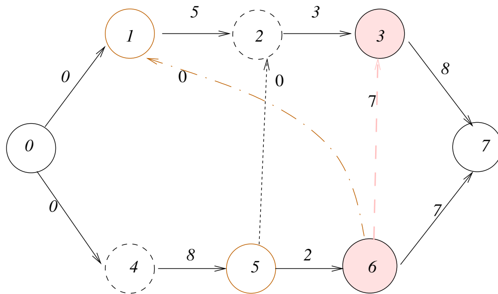

Figure 4. Schedule for BJSP in table 1 whit *Cmax*=26.

The Gantt chart in Figure 5 represents both the processing and blocking times of the solution in Figure 4. For example, after the end of its processing time $J1$ blocks $M1$ until $M2$ becomes available for processing $J1$.

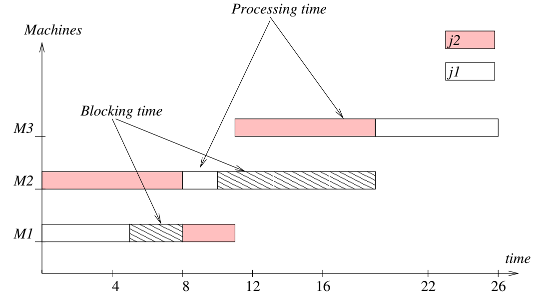

Figure 5. Gantt chart of the schedule in figure 4.

B. Related works

Most of B&B methods, for the job shop problem, are based on the resolution of single machine problems proposed by Carlier. For solving optimally the BJSS we find the B&B method proposed by Mascis *et al.* [1]. the authors formulate the problem by means of an alternative graph model which is a generalization of the disjunctive graph of Roy and Sussman [4]. Based on this model, They solve optimally the 10×10 benchmark instances of this problem.

Ait Zai *et al.* [10], proposed an original B&B method based on graph theory to solve the BJSS. The idea of his branching scheme relies on the implicit enumeration of all possible combinations on a given machine. The authors gave solutions for local instances only.

The B&B algorithms are not sufficient when dealing with large problem instances, therefore computing architectures such GPUs are required. Several authors have proposed to accelerate the B&B method using GPUs. Most of these works focus on solving permutation Flow Shop Problem (FSP), knapsack problem and Traveling Salesman problem.

In [13] and [14]. Chakroun *et al.* take the classical approach of sending nodes to be evaluated on GPU to solve the FSP problem since this step takes more than 98% of the global execution time. Therefore, each GPU thread supports the evaluation of a single node of the search tree. In [15] A.bendjoudi *et al.* extend the approach below and propose a multi-core/GPU scheme to exploit both multi-core CPU processors and GPU accelerator to solve the same problem.

In [18], Esseghir *et al.* proposed a CPU-GPU based B&B applied to the knapsack problem. In the proposed parallelization scheme the branching and bounding can be donne either on the CPU or the GPU according to the size of the search tree. This approach uses less comunication CPU-GPU and better managment of data-structures in GPU memory.

In [12], Carneiro *et al.* apply the B&B to the traveling salesman problem where a pool of nodes is sent to the GPU for evaluation. Each GPU-thread applies the branching and

bounding operators to a single node and builds its own local tree. The resulting nodes are moved back to the CPU where the promising nodes are inserted into the tree.

To the best of our knowledge, our work is the first heterogeneous (CPU-GPU) implementation of B&B algorithm dedicated to JSSP and its blocking extension.

III. The Branch and Bound algorithm for BJSS

The B&B algorithms make an intelligent enumeration of all feasible solutions. They are characterized by two operators: branching and bounding. The branching is a recursive process, which consists to replace the search space of a given problem by a set of smaller sub-problems. The lower bounding operator is used to compute the lower bound for the evaluation of all feasible solutions in the considered sub-problem. The elimination operator uses the bounds to eliminate the sub-problems that cannot improve the current best solution found for the problem.

Table 2
The description of the symbols used in our B&B algorithm.

Symbol	Description
UB	Upper Bound.
$LIST$	A set of nodes (sub-problems).
$s*$	The optimal solution.
R_i	The ieme successor of node R.
LB	Lower Bound.
$LB(R_i)$	Lower bound of node R_i.

Algorithm 1 and Table 2 describe the general structure and symbols used in the proposed Branch-and-Bound algorithm. The most effective B&B algorithms, for the JSSP, are based on the disjunctive graph model [11]. Our B&B is based on the adaptation of this approach to the blocking case (alternative graph) [1]. Our method consists to fix an order (precedence) between every two concurrent operations, which leads to fix the correspondent alternative pair (from A), and a set of fixed arcs represents a selection.

A. Branching

The B&B algorithm can be represented by a search tree. The tree is rooted by the original problem; no alternative pairs are fixed ($|S_0|$=0).

A search tree node R is characterized by (S_R, A_R) and represented by the graph G(S_R)=($N, F \cup S_R$). S_R denotes the set of fixed alternative arcs and A_R represents a set of unselected alternative pairs in this node.

The branching creates two immediate successors ($R1$,$R2$) of R by fixing an alternative pair $((i,j),(h,k)) \in A_R$ that has a direct impact on the longest path in the graph. The node $R1$ (resp. $R2$) is characterized by $S_{R1} = S_R \cup (i,j)$ (resp. $S_{R2} = S_R \cup (h,k)$) and $A_{Ri} = A_R - \{((i,j),(h,k))\}$. The corresponding successors represent the sub-search space related to the fixed alternative arc. After this, each successor is handled recursively in the same way until we find a

Algorithm 1 Pseudo-code of the sequential B&B algorithm

OUTPUT: s^*.
BEGIN
1. $LIST = \{original\ problem\}$;
REPEAT
2. Choose a Node R and remove it from $LIST$;
3. Generate successors R_i from R | ($i = 1, ..., n$);
4. FOUR EACH successour R_i;
BEGIN
5. IF $LB(R_i) < UB$ THEN
BEGIN
6. IF R_i represents one solution THEN
BEGIN
7. $UB = LB(R_i)$;
8. s^* = solution in R_i;
END
9. ELSE $LIST = LIST \cup R_i$;
END
END
UNTIL $LIST$ = $\emptyset$;
RETURN s^*;
END.

complete selections or eliminate sub-problems and prune the tree if the lower bound value of the current sub-problem is bigger than the upper bound. Finally our exploration strategy consists to choose after a branching process the node which has the bigger Makespan (worst first strategy). The advantage of this strategy is the huge number of feasible solutions explored in short time which leads to improve the UB and eliminates a large number of branches.

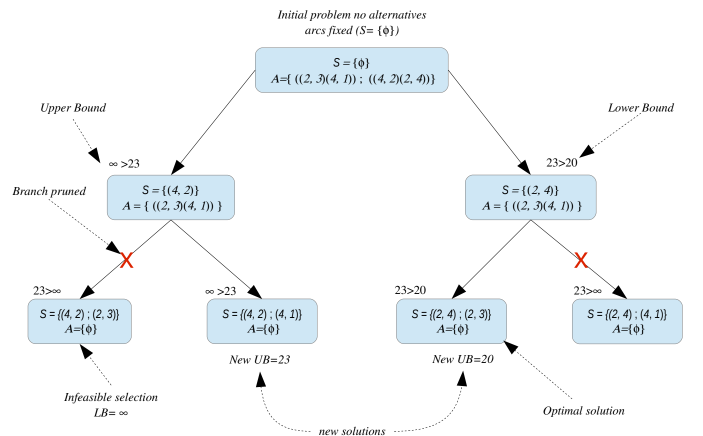

Figure 6. Search tree for the BJSS instance.

Figure 6 represents a search tree for a BJSS instance with 2 jobs and 2 machines; the Figure shows also the existence of two alternative pairs. The first one ((2,3),(4,1)) is between

operations 1 and 3. The second pair ((2,4),(4,2)) is between operations 2 and 4.

B. Evaluation (Bounding)

Any solution of the problem can be considered as initial value for an upper bound (in our case UB=∞) which is updated as soon as a new better solution is found. The lower bound (Evaluation) used in our case is the one used by Carlier *et al.* [7] to solve optimally the JSSP. It is based on the *one machine scheduling problem*. To do a link with alternative graph model , each search tree node represents an alternative graph. The lower bound used is similar to the Makespan of the sub-problem obtained by adjusting the head and tail structures ($H_i = l(0,i), T_i = l(i, n*m)$) for each operation o_i (i = 1, ..., $n*m$); in the graph representation. This process is very expensive and consumes 70% of global execution time of the method. This process is done sequentially for all operations affected by the change made and can be repeated several times for the same operation if there are multiple paths that lead to this operation.

The complexity of the evaluation process depends on the number of operations ($n \times m$) in the treated instance, therefore, the evaluation time increases by increasing the size of the instances. The implementation of the evaluation process, as illustrated below, requires six data structures. The matrix MP $(n*m) \times (n*m)$ represents the length of all alternative arcs, MP[i][j]=a_{ij} if the arc exists and -1 if not. The matrix Succ $((n*m) \times n)$ contains the successors of each operation therefore, row i represents the successors of operation o_i. Similarly, the matrix Pred $((n*m) \times n)$ contains the predecessors of each operation. Vector S contains all selected and unselected pairs. Vector H (resp. T) contains the Head (resp. Tail) of each operation. The element H[i]=$l(0,i)$ represents the longest path from o_0 to o_i. The same T[i]=$l(i, n*m)$ the longest path from o_i to the last operation in the graph o_{n*m}.

Pseudo code of Update Head and Tail.

```
for each operation o affected by the change
    for(int i = 0; i < Successor_o; i++){
        int s=Succ[o][i];
        int a_os =MP[o][s];
        if(H[o]+a_os > H[s]){
            if(s == op){cycle = 1;}
            else{ H[s]=H[o]+a_os; list = list ∪ s; }
    }}
    for(int i = 0; i < Predecessor_o; i++){
        int p=Pred[o][i];
        int a_po =MP[p][o];
        if(T[o]+a_po > T[p]){
            if(p == op){cycle = 1;}
            else{ T[p]=T[o]+a_po; list = list ∪ p; }
    }}
```

C. Immediate selection

The immediate selection represents several techniques which allows to accelerate the B&B algorithm by reducing the number of branching necessary to obtain the optimal solution. This process is done sequentially and costs 18% of the global processing time since there is a large number of alternative pairs (99000) for big instances. This process uses the head and tail values computed in the bounding process.

Given a sub-problem R with feasible selection S_R and a set of unselected pairs A_R. For each unselected pair $((i,j),(h,k)) \in A_R$:
if $l(0,h)+a_{hk}+l(k,n) \geq UB$ then S_R=$S_R \cup (i,j)$. This rule expresses the fact that adding the arc (k,h) (resp. (i,j)) to S_R will produce a sub-problem with a lower bound greater than the upper bound, consequently the arc (i,j) (resp. (h,k)) is added to S_R.

Pseudo code of the Immediate selection

```
int Immedia-selection ()
{   int cycle=0;
    for(int l = 0; l < nbpair; l++){
      int p=S[l];
      if(p > 2){
        Let ((i,j),(h,k)) be the corespondent pair
        int a_ij =MP[i][j];
        int a_hk =MP[h][k];
        int u_2=H[h]+ a_hk+T[k];
        int u_1=H[i]+ a_ij+T[j];
          if(u_1 > UB && u_2 > UB) { cycle = 1; l = ∞;}
          else if(u_1 > UB){S[l]=2;}
          else if (u_2 > UB){S[l]=1;}
      }   }
  return cycle; }
```

IV. The Proposed GPU-based B&B algorithm

The parallelization of B&B algorithm is amplified by the fact that each node of the B&B search-tree can be explored independently. To the best of our knowledge, there is no GPU-based parallel B&B dedicated to JSSP and its blocking extension. We have seen that the evaluation process and the immediate selection consume more than 85% of the global execution time. Therefore, it is crucial to parallelize this phase in order to accelerate the B&B execution time.

The GPU architectures are based on SIMT (Single Instruction, Multiple Threads) paradigm. According to this paradigm, the same program called *kernel* is executed simultaneously by a set of parallel threads whith different data. The threads are organized according to a grid of thread-blocks hierarchy specified in the kernel call. The Grid represents a set of thread-blocks. Threads of the same block can cooperate by using a private shared memory and barrier of synchonisation. Threads can access multiple

memory spaces: *constant memory* and *texture memory* are read-only cached memory accessible by all threads. The *global memory* is read-write memory, also accessible by all threads. Unlike the global memory the *shared memory* is cached memory accessible only by threads in the same block;

In the following, we present a new parallelization scheme for B&B algorithm exploiting GPU-based architecture. The proposed scheme has two levels of parallelization. The first level exploits the fact that the evaluation step can be done in parallel for each node and the second level of parallelization is developed to fully occupy the GPU. We also present the mapping of different data structures needed in the evaluation step.

A. Parallel Evaluation of one Bound (PEB)

This first level scheme is a node based parallelization [19]. It does not change the design of the B&B algorithm because it is similar to the sequential version except that the evaluation is done in parallel on GPU for each node as shown in Figure 8.

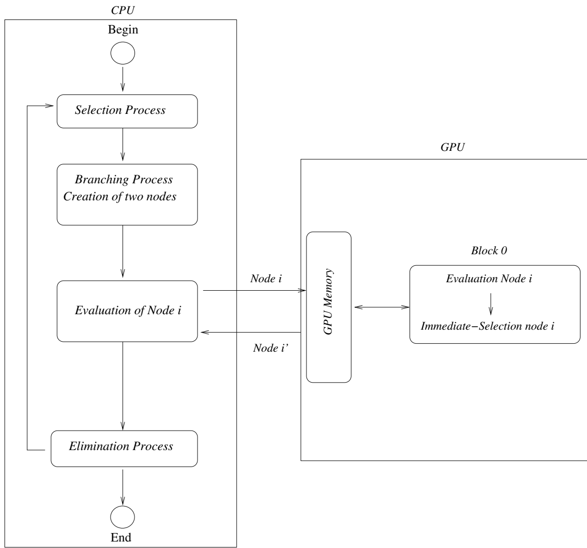

Figure 8. First level PEB scheme.

As already presented, each node of the search tree represents a graph of $n \times m$ operations. The evaluation process consists to update the head and tail values for each operation in the graph. At the PEB level, we propose a parallel evaluation scheme based on the idea that each GPU-thread supports updating head and tail values for a single operation in the graph. Exploiting the fact that the updating can be done independently for each operation. Therefore, the GPU block size equals $n \times m$ the number of operations in the graph.

As shown in Figure 7, at each iteration, only one node is sent to the GPU for evaluation and immediate selection using one thread-block. Each thread updates the head and tail values for one operation. The new values are sent back to the CPU to be used in the branching and elimination process. Furthermore, the following Kernel illustrates the pseudo-code of evaluation and immediate selection used by GPU threads.

Evaluation and Immediate selection Kernel

```
__global__ Evaluation (int * H1, int * T1, int * S)
{    __shared int cycle = 0, MAJ = 0;
  __shared__ int H[n * m];
  __shared__ int T[n * m];
  int index = blockIdx.x * blockDim.x + theadIdx.x;
     /*Initialization of MP, Pred, Succ using vector S*/
if(index < n * m){
     /*copy the vectors H and T in shared memory*/
  H[index] =H1[index];
  T[index] =T1[index];
Do{ MAJ = 0;
  for(int i = 0; i < nbjob; i++){
    int p=Pred[index][i];
    int a_po =MP[p][index];
    if(t[index]+a_p index > t[p])
      {t[p] =t[index]+a_p index;     MAJ = 100; }
    }
  for(int i = 0; i < nbjob; i++){
    int s=Succ[index][i];
    int a_index s =MP[index][s];
    if(H[index]+a_index s > h[s])
    {H[s]=H[index]+a_index s;     MAJ = 100;}
   }
    __syncthreads();
 }while (MAJ! = 0 && cycle == 0)
 }

  /*Immediate-selection:Each thread processes set of pairs*/

int nt = (nbpair/threadDim.x) + 1;
  for(int l = 0; l < nt; l++){
 int id = (l * threadDim.x) + threadIdx.x;
 if(id < nbpair){int p=S[id];
    if(p > 2){ Let ((i, j), (h, k)) be the corresponding pair
     int a_ij =MP[i][j]; int a_hk =MP[h][k];
     int u_2=H[h]+ a_hk+T[k];
     int u_1=H[i]+ a_ij+T[j];
        if(u_1 > UB && u_2 > UB) { cycle = 1; id = ∞;}
          else if(u_1 > UB){S[id]=2;}
          else if (u_2 > UB){S[id]=1;}
          else {S[id]=Max(u_2, u_1 );}
        }
    }
  }
}
```

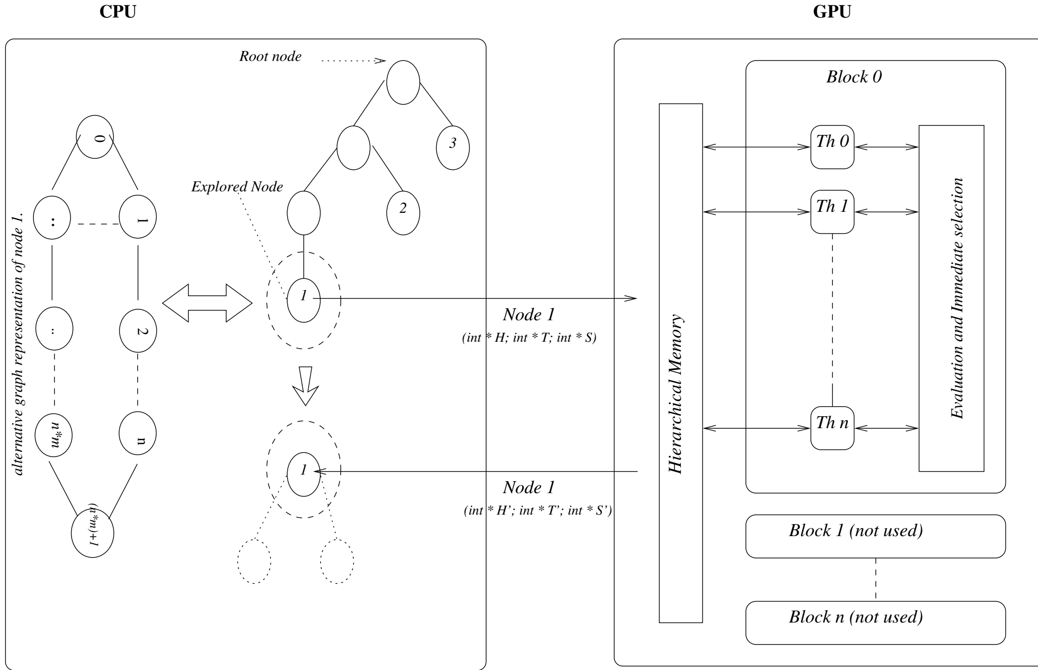

Figure 7. GPU evaluation of single node.

As we have seen in Figure 7, a single block is used on the GPU to evaluate one node. Therefore, only one block is involved in the evaluation process and the others are idle. The weakness of this solution resides in the underused of the GPU capacity and then a waste of a significant computing power. To overcome this drawback, we propose a second level of parallelization.

B. Parallel Evaluation of Several Bounds (PESB)

We propose in this section a second level of parallelization which allows to fully occupy the GPU. This level represents a generalization of the first level (PEB) and it is called Parallel Evaluation of Several Bounds (PESB). This level generalizes the idea of the first level (Bounding is faster) to exploit the power of GPUs. At each iteration a pool of nodes is sent to the GPU for evaluation and immediate-selection. *i.e.* each thread-block supports the evaluation of a single node. Then the new results for each node are sent back to the CPU to be used by the selection, branching and elimination process as shown in Figure 9.

As we have already seen, six data structures are used. The vectors Head (H [$m*n$]), Tail (T [$n*m$]) and alternative pairs (S [$nbpair$]) are sent from the CPU to the GPU. Therefore, they are stored in the global memory of the GPU. The matrices MP, Succ and Pred are also stored in the GPU global memory. These matrices are calculated on the GPU using the vector S. To accelerate this initialization phase the calculation is divided across all the threads in the block.

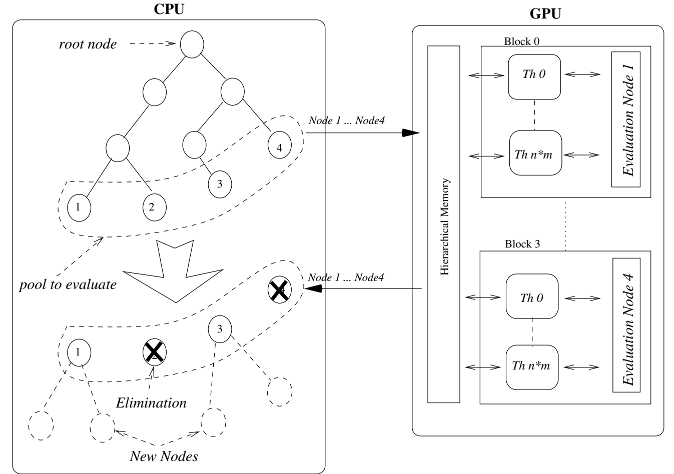

Figure 9. GPU evaluation of several nodes.

The access to the global memory is much longer than shared memory, but the latter contains only 48 kbit by Streaming Multiprocessor (SM). Therefore, the number of blocks that can run in parallel on one SM depends on the amount of shared memory used by each block. In order to have the right number of blocks running in parallel, we use the shared memory only for the Head and Tail vectors since there is a hiegh number of accesses to these vectors.

V. EXPERIMENTATIONS

In this section computational results are given using benchmarks obtained from the well known classical job shop instances by dropping the infinite buffer capacity constraint, and replacing it by a zero buffer capacity.
We tested our algorithms using 80 benchmarks proposed by Taillard's [16]. The different instances are designated by $n \times m$, where n and m represent respectively the number of jobs and the number of machines.
The experiments have been carried out using Intel Xeon E5640 CPU with 2.67GHz clock speed and Nvidia Tesla K40 with 2280 cuda cores and 12 GB GDDR5 of global memory. The approach has been implemented using C-CUDA 7.0.

In the folowing, we repport the execution times of the sequential and the proposed GPU B&B method. For both the tables 3 and 4, the first column (*Size*) reports the size of the treated problem. Each row ($n \times m$) represents the average execution time in second of 10 benchmarks instances where each one explores 700,000 nodes. Column *Seq.* $B\&B$ reports the average execution time in seconds of an optimized sequential B&B developed in another project. finally, column (*speedup*) reports The ratio between the sequential and parallel execution time.

Table 3
AVERAGE EXECUTION TIME OF THE FIRST LEVEL PEB TO EXPLORE 700000 NODES.

Size	*Seq.* $B\&B$	$B\&B_{PEB}$	*speedup*
15×15	188	612	0.31
20×15	384	672	0.57
20×20	393	760	0.51
30×15	1076	832	1.30
30×20	1127	1000	1.13
50×15	4246	1244	3.41
50×20	10546	1726	6.51

Table 3 reports the results of the first level of parallelization (PEB). Column $B\&B_{PEB}$ reports the average execution time obtained by sending one node for evaluation on GPU at each iteration.

The first result from table 3 is the positive impact of using GPU architectures to reduce the execution time needed to solve this problem. We notice a loss in performance for small instances because the communication time between the CPU and the GPU is more important than the computing time on the GPU. With the increase of instance size, we notice a significant improvement in execution time compared to the sequential case.

The results also confirm the efficiency of our first level which is 7 times faster compared to a sequential B&B version. Notice that this level does not depend on GPU capacity since we use one thread-block. For the 100×20 benchmark instances there are more than 2000 operations. Therefore, we can not use this approach since the GPU hardware limit is 1024 threads par block.

Table 4
AVERAGE EXECUTION TIME OF THE SECOND LEVEL PESB TO EXPLORE 700000 NODES.

Size	$B\&B_{Seq.}$	$B\&B_{M_{core}}$	$B\&B_{P_{ESB}}$	*Blocks*$_{used}$	*speedup*
15×15	188	52	23	240	8.2
20×15	384	113	29	240	13.2
20×20	393	120	41	240	9.6
30×15	1076	375	48	240	22.4
30×20	1127	447	69	240	16.3
50×15	4246	1454	114	100	37.3
50×20	10546	3728	161	100	65.5

Table 4 reports the results of the second level of parallelization ($PESB$). Column $B\&B_{M_{core}}$ gives the execution time, in second, obtained by exploiting only the CPU cores of our workstation. 4 instances of parallel B&B with message exchange have been launched since our workstation CPU contains four cores. Column $B\&B_{PESB}$ reports the average execution of our second level $PESB$ obtained by sending several nodes for evaluation on GPU. Finally, column *blocks-used* shows, at each iteration, the number of nodes sent to the GPU for evaluation by $B\&B_{PESB}$ version.

Unlike the first level, this level scheme ($PESB$) provides good acceleration even for small instances. This level depends on the GPU-hardware since we send several nodes to be evaluated on the GPU. The number of nodes sent to the GPU depends on GPU hardware limit and the resources consumed by our kernel (shared memory and registers). To facilitate this task, Nvidia has developed a tool called *Cuda Occupancy Calcular* [18]. that is used in our case to know the right number of blocks to run on the GPU.

This step results boost the speed up to reach more than 65 times faster compared to an optimized B&B method. Also the results confirm the efficiency of our two level scheme GPU-B&B.

Comparing the results of our GPU B&B with those of multi-core B&B, shows the advantage of using GPUs against CPUs that usually contain a limited number of cores. We also notice that our GPU B&B scheme reaches around 23 times faster than multi-core B&B version.

VI. CONCLUSION

In this paper we proposed an original way to accelerate the B&B method. The proposed two level scheme represents a first work to accelerate the B&B method dedicated to job shop problem and it blocking extension. This problem is NP-hard and represents a version of the JSSP with no intermediate buffer between machines. The first level scheme does not change the design of the B&B algorithm except that the bounding operator is faster. A second level is proposed in order to fully occupy the GPU. Therefore, at each iteration

several search tree nodes will be sent to be evaluated using several GPU blocks.

The obtained results confirm the efficiency of the proposed approach and the positive impact of using parallel architecture to solve this problem. Also the results show a good speedup since we have obtained an acceleration around 65 for large instances compared to an optimized sequential B&B version.

As a perspective, we plan to explore heterogeneous architectures such as multi-core CPUs, coupled with GPUs and Intel Xeon Phi architecture.

Acknowledgment

Dr. Didier El Baz gratefully acknowledges the support of NVIDIA Corporation with the donation of the Tesla K40 GPU used for this research work.

References

[1] A. Mascis and D. Pacciarelli, Job-shop scheduling with blocking and no-wait constraints, European Journal of Operational Research, 143, 498-517 (2002).

[2] G. Hall and S. Sriskandarajah, A survey of machine scheduling problems with blocking and no-wait in process, Operations Research, 44(3), 510-525 (1996).

[3] Y. Mati, N. Rezg, and X.L. Xie, A taboo search approach for deadlock-free scheduling of automated manufacturing systems, Journal of Intelligent Manufacturing, 12(5-6), 535552 (2001).

[4] B. Roy and B. Sussman, Les problems dordonnancement avec contraintes disjonctives, Note DS 9 bis, SEMA, Paris, (1964).

[5] C. Meloni, D. Pacciarelli, and M. Pranzo, A rollout metaheuristic for job shop scheduling problems, Annals of Operations Research, 131, 215-235 (2004).

[6] D. Pacciarelli and M. Pranzo, An Iterated Greedy metaheuristic for the Job Shop Scheduling Problem with Blocking, Proceedings of the 10th Metaheuristic International Conference, Singapore, (2013).

[7] J. Carlier, E. Pinson /European Journal of Operational Research 78 (1994) 146-161.

[8] H. Groflin and A. Klinkert, A new neighborhood and tabu search for the blocking job shop, Discrete Applied Mathematics, 157(17), 3643-3655 (2009).

[9] A. Oddi, R. Rasconi, A. Cesta, and S.F. Smith, Iterative improvement algorithms for the blocking job shop, In Twenty-Second International Conference on Automated Planning and Scheduling, 2012.

[10] A. AitZai, B. Benmedjdoub, and M. Boudhar, Branch and bound and parallel genetic algorithm for the job shop scheduling problem with blocking, International Journal of Operational Research, 14(3), 343-365, (2012).

[11] P. Brucker, Scheduling Algorithms (5ed., Springer), 178-221 (2007).

[12] T. Carneiro, A. Einstein Muritiba, M. Negreiros, G. Augusto Lima de Campos A New Parallel Schema for Branch-and-Bound Algorithms Using GPGPU. 2011 23rd International Symposium on Computer Architecture and High Performance Computing.

[13] I. Chakroun, M. Mezmaz, N. Melab and A. Bendjoudi. Reducing thread divergence in a GPU-accelerated branch&and&bound algorithm. Concurrency and Computation: Practice and Experience, 2013.

[14] Melab, N., Chakroun, I., Mezmaz, M., & Tuyttens, D. (2012, September). A GPU-accelerated branch-and-bound algorithm for the flow-shop scheduling problem. In Cluster Computing (CLUSTER), 2012 IEEE International Conference on (pp. 10-17). IEEE.

[15] A. Bendjoudi, M. Chekini , M. Gharbi , M. Mehdi , K. Benatchba, F. Sitayeb-Benbouzid, and N. Melab Parallel B&B Algorithm for Hybrid Multi-core/GPU: HPCC 2013,

[16] E. Taillard. Taillards FSP benchmarks. http://mistic.heigvd.ch/taillard/problemes.dir/ordonnancement.dir /ordonnancement.html.

[17] M. Esseghir Lalami, D. El-Baz. GPU implementation of the Branch and Bound method for knapsack problems. 2012 IEEE 26th International ParallelParallel and Distributed Processing Symposium Workshops &PhD Forum.

[18] CUDA GPU Occupancy Calculator http://developer.nvidia.com/cuda.

[19] Gendron and T.G. Crainic. Parallel Branch-and-Bound Algorithms : Survey and Synthesis. Operations Research, 42(06):10421066, 1994.