IVM-based Work Stealing for Parallel Branch-and-Bound on GPU

J. Gmys[1], M. Mezmaz[1], N. Melab[2] and D. Tuyttens[1]

[1] Mathematics and Operational Research Department (MARO), University of Mons, Belgium
[2] INRIA Lille Nord Europe, Université Lille 1, CNRS/CRIStAL, Cité scientifique - 59655, Villeneuve d'Ascq cedex, France

Abstract. The irregularity of Branch-and-Bound (B&B) algorithms makes their design and implementation on the GPU challenging. In this paper we present a B&B algorithm entirely based on GPU and propose four work stealing strategies to balance the workload inside the GPU. Our B&B is based on an Integer-Vector-Matrix (IVM) data structure instead of a pool of permutations, and work units exchanged are intervals of factoradics instead of sets of nodes. To the best of our knowledge, the proposed approach is the pioneering to perform the entire exploration process on GPU. The four work stealing strategies have been experimented and compared to a multi-core IVM-based approach using standard flow shop scheduling problem instances. The reported results show, on the one hand, that the GPU-accelerated approach is more than 5 times faster than its multi-core counterpart. On the other hand, the best of the four strategies provides a near-optimal load balance while consuming only 2% of the total execution time of the algorithm.

Keywords: GPU computing; Branch-and-Bound; Combinatorial optimization; Work Stealing

1 Introduction

Many permutation-based combinatorial optimization problems, like flowshop, can be solved to optimality using Branch-and-Bound (B&B) algorithms. These algorithms perform an implicit enumeration of all possible solutions by dynamically constructing and exploring a tree. This is done using four operators: branching, bounding, selection and pruning. Execution times of B&B algorithms significantly increase with the size of the problem instance, and often only small or moderately-sized instances can be practically solved. Because of their massive data processing capability and their remarkable cost efficiency, graphics processing units (GPU) are an attractive choice for providing the computing power needed to solve such instances. However, B&B is a highly irregular application, in terms of control flow, memory access patterns and work load distribution. The acceleration of B&B algorithms using GPUs is therefore a challenging task which is addressed by only a few works in the literature [2, 3, 10]. Most approaches use pools of subproblems, implemented as linked-lists, stacks or queues to store and manage the B&B tree. The GPU memory imposes limitations on the use of such dynamic data

Gmys J., Mezmaz M., Melab N., Tuyttens D. (2016) IVM-Based Work Stealing for Parallel Branch-and-Bound on GPU. In: Wyrzykowski R., Deelman E., Dongarra J., Karczewski K., Kitowski J., Wiatr K. (eds) Parallel Processing and Applied Mathematics. PPAM 2015. Lecture Notes in Computer Science, vol 9573. Springer, Cham, pp. 548–557, 2016.

structures, and pool-based approaches perform at least one of the B&B operators on the CPU, requiring costly data transfers between CPU and GPU. The Integer-Vector-Matrix (IVM) data structure [16], used in our GPU-based algorithm, allows to overcome this issue. To the best of our knowledge, the proposed approach is the pioneering to perform the entire exploration process on GPU. Performing all B&B operators on the device raises the issue of balancing the workload inside the GPU. In this paper, we present the GPU-based B&B algorithm using the IVM structure and propose four work stealing (WS) strategies to address the issue of work load imbalance.

2 Parallel B&B and flow shop

Serial B&B. B&B algorithms explore the space of potential solutions (search space) by dynamically building a tree whose root node represents the initial problem to be solved. The leaf nodes are possible solutions and the internal nodes represent subspaces of the total search space. Subproblems (internal nodes) are stored in a data structure which initially contains only the root node. The best solution found so far is initialized at ∞ and can be improved from one iteration to another. At each iteration of the algorithm, the branching operator partitions a subproblem into several smaller, pairwise disjoint subproblems. The bounding operator is used to compute a lower bound (LB) value of the optimal solution of each generated subproblem. Based on this LB the pruning operator decides whether to eliminate a node or to continue its exploration. According to a predefined exploration strategy the selection operator chooses one subproblem among all pending subproblems stored in the data structure. For instance, the selection of a node can be based on its depth in the B&B tree which leads to a depth-first strategy, which is used in this paper.

Parallel B&B. The pruning mechanism efficiently reduces the size of the search space and thus the computing power needed for its exploration. However, especially for larger instances, the exploration time remains significant and parallel processing is required to speed up the exploration process. A taxonomy of models to parallelize B&B algorithms is presented in [13]. Among the identified models are (1) the parallel tree exploration model, and (2) the parallel evaluation of bounds model. Our GPU-based B&B algorithm uses a combination of models (1) and (2).

Model (1) consists in simultaneously exploring different search subspaces of the initial problem. This means that the selection, branching, bounding and pruning operators are executed in parallel, synchronously or asynchronously, by different B&B processes which explore these subspaces independently. In synchronous mode, like in our GPU-based algorithm, the B&B algorithm has different phases between which the B&B processes are synchronized and may exchange information. The degree of parallelism in this model may be important, especially when solving large instances. Indeed, the number of parallel exploration processes is only limited by the capacity to supply them continuously with subproblems to explore. As the shape of the B&B tree is highly irregular, this work supply strongly depends on the distribution and sharing of the work load. Model (1) can be combined with other parallel B&B models.

Indeed, each independent B&B process may in turn be parallelized, adding a second level of parallelism. For instance, each independent B&B process may use model

(2), which parallelizes only the bounding operator. This model is data-parallel, synchronous and fine-grained (the cost of the evaluation of a bound) and is thus suitable for GPU computing. The GPU-accelerated B&B proposed in [14] is based on model (2) and offloads pools of subproblems for bounding to the GPU. Although a considerable acceleration of the algorithm is achieved, the data transfers between CPU and GPU constitute a bottleneck for the offload-approach. Our GPU-based B&B uses a 2-level combination of models (1) and (2). It aims at using enough parallel exploration processes on the GPU such that the number of generated subproblems per iteration saturates the device during the parallel bounding phase.

Flow-shop. The permutation flow shop problem is a well known NP-hard combinatorial optimization problem. It belongs to the category of scheduling problems and is defined by a set of N jobs $J_1, J_2, \ldots, J_N$ to be scheduled in the same order on M machines. The scheduling obeys the chain production principle, i.e. a job can not be processed on a machine M_j before it has finished processing on all machines M_i located upstream ($i < j$). An operation cannot be interrupted, and a machine processes not more than one job at a time. A duration is associated with each operation. The goal is to find a permutation schedule that minimizes the total processing time called makespan. In [4], it is shown that the minimization of the makespan is NP-hard from 3 machines upwards. The effectiveness of B&B strongly depends on the relevance of the used LB for the makespan. The LB proposed by Lageweg *et al.* [8] is used in our bounding operator. This bound is known for its good results and has complexity of $O(M^2 N log(N))$, where N is the number of jobs and M the number of machines. It is mainly based on Johnson's theorem [6] which provides a procedure for finding an optimal solution for the flow shop problem with $M = 2$.

The benchmark instances used in our experiments are the flowshop instances defined by Taillard [17]. We use only the 10 instances where $M = N = 20$. For most instances where $M = 5$ or 10, the bounding operator gives such good LBs that it is possible to solve them in few seconds using a sequential B&B. Instances with $M = 20$ and $N = 50, 100, 200$ or 500 are very hard to solve. For example, the resolution of the instance *Ta056* ($N = 50$, $M = 20$), performed in [15], lasted 25 days with an average of 328 processors and a cumulative computation time of about 22 years.

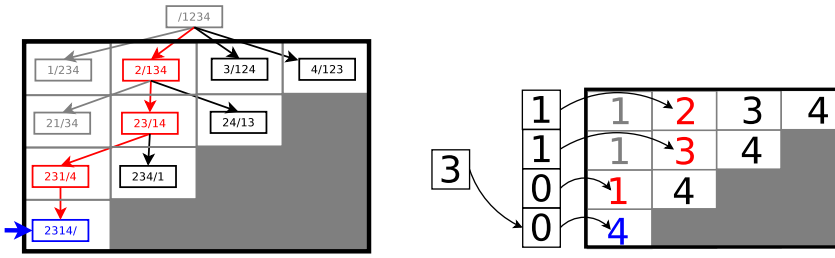
3 GPU IVM-based B&B

3.1 Serial IVM-based B&B

The pool of Fig. 1a is represented as a tree in order to visualize the problem-subproblem relationship between nodes. In Fig. 1b the corresponding Integer-Vector-Matrix (IVM) is represented. The notation $23/14$ denotes the subproblem where jobs 2 and 3 are scheduled, while 1 and 4 are unscheduled. In the initial problem all jobs are unscheduled. The root node $/1234$ is decomposed into four nodes, namely $1/234$, $2/134$, $3/124$ and $4/123$. In the IVM-approach this decomposition corresponds to the first row of the matrix M which contains all job numbers $1, 2, 3, 4$. The example assumes that the algorithm selects and branches the second node $2/134$. For the IVM-structure this translates

to setting $V(0) = 1$. The so-called *position-vector* V always points to the currently active node at depth I. The decomposition of $2/134$ gives three nodes, $21/34$, $23/14$ and $24/13$. In IVM-terms this decomposition consists in copying the elements of row $I = 0$, except the scheduled job $M(0,V(0)) = 2$, to the next row. Also, the integer I is incremented by one when a subproblem is decomposed. The example also assumes that the node $21/34$ is processed or pruned. The IVM-based selection operator should therefore ignore the corresponding cell of M and move rightward to the next cell by incrementing $V(1)$. Cells that correspond to pruned nodes are flagged, multiplying them by -1. Therefore, the algorithm decomposes the second node $23/14$, obtaining two new nodes which are $231/4$ and $234/1$. Again, the next-row process performs this decomposition in the IVM structure.

Each of the pool management operators can be expressed in terms of actions on the IVM. The depth-first selection strategy is naturally encoded in this data structure. In order to enable the bounding operator to compute the lower bounds of a subproblem encoded by the IVM structure, a decode operation is required. For example, the solution encoded in Fig. 1b can be directly read by looking (from row $I = 0$ to row $I = 3$) at the values that are pointed by the position vector. With the same vector and matrix, if the integer is $I = 1$, then the represented subproblem is $23/14$.



(a) pool of subproblems (b) IVM-structure

Fig. 1: Example of a pool of subproblems and an IVM-structure obtained when solving a permutation problem of size four

3.2 Work unit: interval of factoradics

Throughout the exploration process, the position-vector behaves like a counter. In the example of Fig. 1b, the vector successively takes the values *0000*, *0010*, *0100*, …, *3200*, *3210*. These 24 values correspond to the numbering of the 4! solutions using a numbering system in which the weight of the i^{th} position is equal to $i!$ and the digits allowed for the i^{th} position are $0, 1, \ldots, i$. Applied to the numbering of permutations, the French term *numération factorielle* was first used in 1888 [9], while Knuth [7] uses the term *factorial number system*. The term *factoradic*, which seems to be of more recent date is used, for instance, in [12]. This mixed radix numeral system satisfies the conditions of what Cantor called a *simple* number system in [1].

Subtrees of the B&B tree can be identified with intervals of factoradics. In the example of Fig. 1b, the algorithm explores $[0000, 3210[$. It is possible to have two IVMs[3] $R1$, $R2$ such that $R1$ explores $[0000, X[$ and $R2$ explores $[X, 3210[$. These factoradic intervals are used as work units, instead of conventional sets of nodes. If $R2$ ends exploring its interval before $R1$ does, then $R2$ steals a portion of $R1$'s interval. Therefore, $R1$ and $R2$ can exchange their interval portions until the exploration of all $[0, N![$. With the exception of rare works such as [15], work units exchanged between processes are sets of nodes. To enable an IVM to explore any interval $[A, B[$ an initialization process is necessary. The correct initialized state of the IVM is such that it would be the same if the new position vector $V = A$ had been reached through the exploration process. Therefore, the initialization process differs from the normal B&B process only in the selection operator. Instead of depth-first selecting the next node, an initializing IVM selects at each level k the node pointed by $V(k)$ as long as the selected subproblem is promising. If a pruned node is selected, the initialization process is finished and the IVM resumes exploration, searching for the next node to decompose. Thus, the initialization process may last for 1 to N iterations.

3.3 GPU-based parallel B&B

IVM is well-adapted to a GPU implementation of the parallel B&B algorithm. Contrary to a conventional pool of subproblems, IVM requires no dynamic memory allocation and provides good data locality. A fixed number T of IVM structures is allocated in device memory and the B&B operators, implemented as CUDA-kernels, act on these data structures in parallel. Figure 2 provides an overview of the GPU-based algorithm. Depending on the current state of an IVM different actions are performed. An IVM cycles through three different states: *exploring*, *empty* or *initializing*.

In the `goToNext` kernel, all exploring IVMs try to select the next subproblem to decompose. If no next subproblem is found, the interval is empty and the IVM-state is set to *empty*. Initializing IVMs skip this depth-first selection procedure as the next node to decompose is pointed by the position vector the IVM is initializing at. If the initialization process is terminated, then the state is set to *exploring*. All non-empty IVMs move to the next row in the matrix.

For IVMs with non-empty intervals the `decode` kernel reads the IVM structure and builds the father subproblem to be decomposed. The kernel `bound` uses this data and computes the lower bounds for the children subproblems. The granularity is the computation of the lower bound for a child subproblem. The degree of parallelism depends on the number of non-empty IVMs and on the average depth of these explorers in the B&B-tree. In order to maintain a high device occupancy, the number of empty IVMs should be as low as possible. Also the number of initializing IVMs should be low, because the bounding procedure for these IVMs contributes to computational overhead. As the number of children per non-empty IVM is variable a `remap` phase precedes the bounding kernel. Using a parallel prefix sum [5] computation, the remapping performs a compaction of the mapping of threads onto children subproblems. It also detects the

[3] In the rest of this paper the term IVM designates the data structure as well as, by extension, the exploration process associated with a part of the B&B-tree.

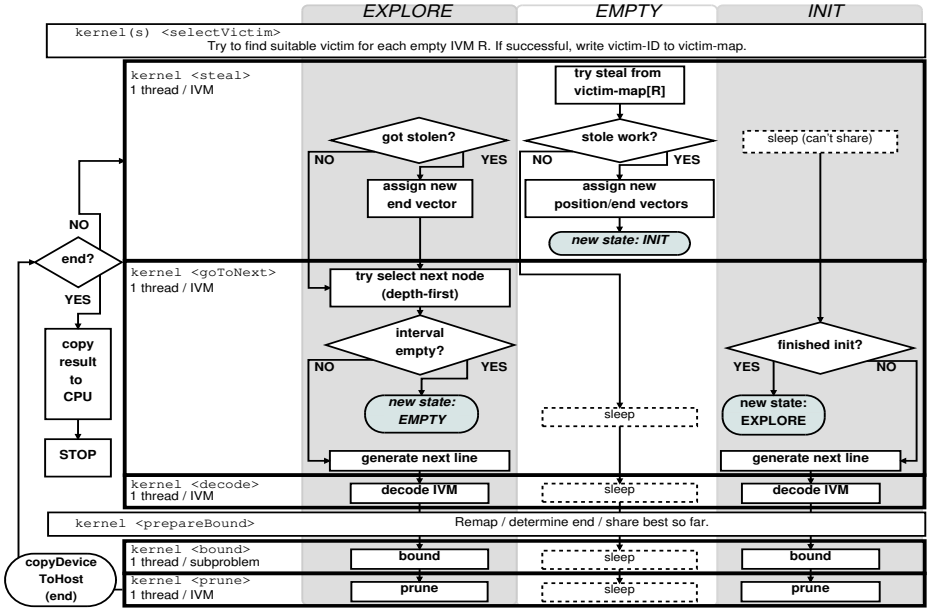


Fig. 2: Flowchart of the CUDA B&B algorithm

end of the algorithm ($\Leftrightarrow$ all IVMs empty) and determines global best solution found so far using a min-reduce. The `prune` kernel uses the lower bounds computed in the bounding kernel to decide whether a subproblem is pruned or not.

Empty IVMs remain unchanged during an iteration, they try to steal work in the `steal` kernel. A work stealing (WS) operation is determined by a victim selection strategy, defining which IVM to steal from, and a granularity policy, defining how much of a victim's work unit is stolen. In the `selectVictim` phase a one-to-one mapping of empty IVMs onto suitable victim IVMs is built. As in the `steal` kernel all empty IVMs try to acquire work in parallel, two empty IVMs should never select the same victim. Initializing IVMs can not be selected as WS victims, because this could result in a deadlock situation. Moreover, the victim selection operation should (1) induce minimal overhead, thus the mapping should be build in parallel, (2) select victim IVMs whose intervals are likely to contain more work than others, (3) serve a maximum of empty IVMs.

4 Work stealing strategies

This section explains the IVM-based WS strategies for the GPU-B&B algorithm. In [11] several WS strategies for multi-core IVM-based B&B algorithms have been proposed. For instance, the `Ring-1/T` strategy can be directly transposed from the asynchronous multi-core WS to the synchronous GPU case. The proposed WS strategies are extensions of this strategy. They are described in Alg. 1 and correspond to different choices for a set of parameters.

Algorithm 1 Victim selection

Ring-1/T: $\mathbf{B}=1$; $\mathbf{S}=0$; $\mathbf{C}=0$; *Search-1/T:* $\mathbf{B}=1$; $\mathbf{S}=25$, $\mathbf{C}=0$;
Large-1/2: $\mathbf{B}=1$; $10<\mathbf{S}<\mathrm{T}$; $\mathbf{C}=1$; *Circle-1/2:* $\mathbf{B}=$ `iter%T`; $10<\mathbf{S}<\mathrm{T}$; $\mathbf{C}=1$;

```
1: procedure SELECTVICTIM(B, S, C)
2:    for (k =B→ B+S) do
3:       beg<<<T threads>>>(k, victim-map, C,...)
4:    end for
5: end procedure
6: procedure <<< >>> BEG(k, victim-map, C,...)
7:    ivm ← blockIdx.x*blockDim.x + threadIdx.x
8:    if (state[ivm]=empty) then
9:       V ← (ivm-k)%T
10:      if (state[V]=exploring AND flag[V]=0 AND length[V]>C*meanLength) then
11:         victim-map[ivm]← V
12:         flag[V]← 1
13:      end if
14:   end if
15: end procedure
```

The **Ring**-selection policy consists in selecting a victim in round-robin fashion, meaning that an empty IVM $R \in \{1,...,T\}$ tries to steal a portion of work from IVM $(R-1)\%T$. In Alg. 1 this corresponds to parameters $B=1$ and $S=0$. If the state of the selected potential victim $(R-1)\%T$ is *exploring*, then work can be stolen. No conditions on the length of the victim's interval are imposed, which corresponds to setting $C=0$ in Alg. 1. To use this information in the `steal`-kernel, `victim-map[R]` is set to $(R-1)\%T$. A thief steals all but the T^{th} part of its victim's interval. In [11] it is shown that this granularity policy better suits the *Ring* selection strategy than a $1/2$-policy.

The **Search**-1/T selection policy extends the *Ring* strategy by checking successively if work can be stolen from IVMs $(R-1)\%T$, $(R-2)\%T$, ... , $(R-S)\%T$. Seaching the entire ring ($S=T$) results in excessive overhead, so for experimental purposes the length of the search window S is set to 25. The idea behind this strategy is to avoid the following situation. Using the *Ring* policy, if a group of l empty IVMs is queued behind a group of exploring IVMs, then it takes at least l iterations to serve all IVMs in that group. In order to avoid multiple selections of the same victim, a flag-variable is introduced (Alg. 1, line 10). The *Search* policy aims at increasing the probability that an empty IVM succeeds its WS attempt at a given iteration.

The idea behind the **Large** selection policy is to steal from larger intervals as they are likely to contain more nodes to decompose. This requires computing the length of each interval at each iteration. In order to avoid the costly operation of sorting the IVM-IDs by their corresponding interval-lengths, the current mean interval-length is computed prior to the victim selection phase. Having an interval larger than average is added as a criterion for the eligibility of an IVM as a WS victim. In Alg. 1 this corresponds to setting $C=1$. However, this length-criterion increases the probability that no victim is found in the search window of fixed length S. The parameter S is therefore allowed to float between 10 and T. If more than 10% of IVMs are empty at a given iteration, then S is incremented by one, otherwise S is decremented by one. Auto-tuning this parameter requires copying the number of empty IVMs to the host at each iteration. Choosing large intervals justifies the granularity policy that consists in stealing half of the victim's interval.

Table 1: Exploration time (in seconds) for solving flowshop instances *Ta021-Ta030*

Inst.	$\times 10^6$ Nodes	Ring-1/T (CPU) Time	Ring-1/T (GPU) Time	Rate	Search-1/T (GPU) Time	Rate	Large-1/2 (GPU) Time	Rate	Circle-1/2 (GPU) Time	Rate
21	41.4	1371	386	3.6	338	4.1	280	4.9	250	5.5
22	22.1	668	247	2.7	229	2.9	146	4.6	129	5.2
23	140.8	4466	1002	4.6	934	4.8	915	5.4	813	5.5
24	40.1	1142	359	3.2	254	4.5	255	4.5	219	5.2
25	41.4	1422	431	3.3	327	4.3	280	5.1	250	5.7
26	71.4	1975	459	4.3	443	4.5	429	4.6	384	5.2
27	57.1	1600	404	3.9	370	4.3	336	4.8	301	5.3
28	8.1	263	120	2.2	79	3.3	59	4.4	52	5.1
29	6.8	216	95	2.3	88	2.5	50	4.4	42	5.1
30	1.6	58	39	1.5	36	1.6	20	2.9	12	4.8
Avg	**43.1**	**1318**	**354**	**3.7**	**310**	**4.3**	**277**	**4.8**	**245**	**5.4**

In the **Circle** selection policy the search window is shifted by one position at each iteration of the algorithm. In Alg. 1 this corresponds to setting B = iter%T. Like in the *Large* policy the parameter *S* is adapted dynamically and half of the victim's interval is stolen. The main idea behind this policy is to reduce the length of the search window as a large value for *S* induces most overhead.

5 Experiments

Hardware/Experimental protocol. For all experiments an NVIDIA Tesla K20m GPU, and version 5.0.35 of the CUDA Toolkit are used. As explained in Section 2, the medium-sized instances *Ta021-Ta030* [17] are used. The execution times are compared to a multi-core B&B [11] running on 2 Sandy Bridge E5-2650 processors with 32 threads. When an instance is solved twice using a multi-threaded B&B, the number of explored subproblems often differs between the two resolutions. Therefore, we choose to always initialize our B&B by the optimal solution of the instance to be solved. With this initialization, the number of decomposed nodes is the same in each exploration. The number of used IVMs is $T = 768$.

Experimental results. Table 1 shows the exploration time for flow shop instances *Ta021-Ta030* using different WS strategies. It also shows the rate comparing the GPU-based approaches to a multi-core IVM-based B&B using 32 threads (= 32 IVMs) and an asynchronous *Ring-1/T* WS strategy. For all instances the best performance is achieved with the *Circle* WS strategy. On average, this approach performs the exploration the B&B-trees 5.4 times faster than the multi-core version.

Comparing the different WS strategies, the decreasing exploration time can be explained by the fact that more nodes are decomposed per iteration. Table 2 shows the *IVM-efficiency* ($= 100\% \times \frac{\#\text{decomposed nodes}}{\#\text{iter} \times \text{T}}$) which provides a measure for the efficiency of the proposed WS strategies. Indeed, with an ideal work load balance each IVM decomposes a node per iteration. In that case *IVM-efficiency*$= 100\%$ and the number of required iterations (Table 2) is minimal for the fixed number of IVMs. The results show that the *Circle* strategy is close to that ideal situation – per iteration on average only 2.5% of the $T = 768$ IVMs are either empty or initializing.

Table 2: Average percentage of IVMs in *exploring* state (IVM-Efficiency) and average number of iterations performed to complete the exploration

	Ta021	Ta022	Ta023	Ta024	Ta025	Ta026	Ta027	Ta028	Ta029	Ta030	**Avg**	#Iterations
Ring-1/T	52.0	38.4	74.5	48.2	45.6	71.0	63.9	24.6	29.1	14.8	**46.2**	96781
Search-1/T	67.4	46.3	84.2	84.2	73.8	80.4	79.0	53.8	36.7	19.9	**62.5**	74402
Large-1/2	93.3	92.9	93.4	93.2	93.3	93.2	93.4	91.5	90.4	76.4	**91.1**	60108
Circle-1/2	99.4	98.9	99.8	99.4	99.3	99.7	99.5	96.8	96.8	85.3	**97.5**	56368

Finally, the overhead induced by WS is evaluated. Figure 3 shows, for the four WS strategies, the average time spent in different phases of the algorithm. For all WS strategies, the bounding phase consumes $> 85\%$ of the total execution time. Using the *Ring-1/T* strategy on average 327 seconds are spent in the bounding kernel, against 226 seconds using the *Circle-1/2* strategy. This corresponds to the observations regarding the efficiency of that kernel (Tab. 2). Compared to basic ring, the search-window of length $S = 25$ greatly improves the efficiency of the WS, at the cost of spending $\approx 3\%$ instead of $\approx 0.1\%$ in victim-selection. The *Large* strategy further improves the work load balance, but victim-selection amounts for $\approx 9\%$ (including computation of interval-length) as the average value for the auto-tuned parameter S increases to 110. Extending the latter to the *Circle* strategy allows a modest further improvement of the load balance and, more importantly, as the average value for S decreases to 15, it achieves this with a victim selection cost of $< 2\%$.

6 Conclusion and future work

In this work, we have presented a GPU-based B&B algorithm which is, to the best of our knowledge the pioneering to perform the entire exploration process on the GPU. We focused on the challenge of balancing the irregular work load using work stealing (WS). The results show that the performance of B&B on GPU significantly depends on work load balancing. Compared to a 32-threaded multi-core version the GPU-based counterpart provides an average acceleration of 5.4 using the most efficient of the proposed WS strategies. As a future work we plan to investigate the use of other hardware accelerators

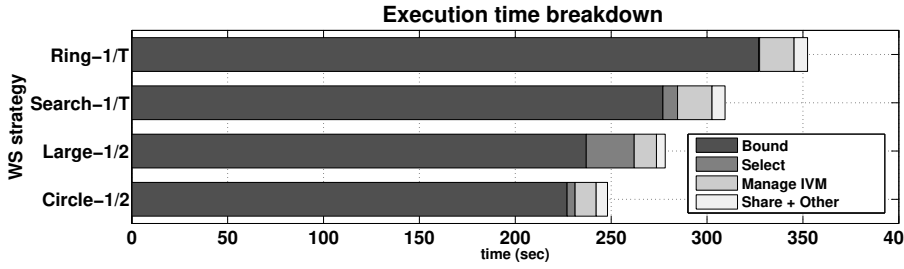


Fig. 3: Average elapsed time for solving instances *Ta021-Ta030* and its repartition among different phases of the algorithm

like Intel Xeon Phi for B&B and to extend the IVM-based B&B to a cluster-based version using many-core accelerators. Solving Taillard's 50-job flowshop instances within a reasonable amount of time requires indeed the combined computing power of multiple multi-core and many-core processors. We also plan to enable the algorithm to use a library of bounding functions in order to validate our results on other permutation-based optimization problems.

References

1. Cantor, G.: Ueber die einfachen Zahlensysteme. Zeitschrift für Mathematik und Physik 14, 121–128 (1869)
2. Carneiro, T., Muritiba, A., Negreiros, M., Lima de Campos, G.: A New Parallel Schema for Branch-and-Bound Algorithms Using GPGPU. In: 23rd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD). pp. 41–47 (2011)
3. Chakroun, I., Mezmaz, M., Melab, N., Bendjoudi, A.: Reducing thread divergence in a GPU-accelerated branch-and-bound algorithm. Concurrency and Computation: Practice and Experience 25(8), 1121–1136 (2013)
4. Garey, M.R., Johnson, D.S., Sethi, R.: The Complexity of Flowshop and Jobshop Scheduling. Mathematics of Operations Research 1(2), pp. 117–129 (1976)
5. Harris, M., Sengupta, S., Owens, J.D.: Parallel prefix sum (scan) with CUDA. GPU Gems 3(39), 851–876 (2007)
6. Johnson, S.M.: Optimal two- and three-stage production schedules with setup times included. Naval Research Logistics Quarterly 1(1), 61–68 (1954)
7. Knuth, D.E.: The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1997)
8. Lageweg, B.J., Lenstra, J.K., Kan, A.H.G.R.: A General Bounding Scheme for the Permutation Flow-Shop Problem. Operations Research 26(1), 53–67 (1978)
9. Laisant, C.A.: Sur la numération factorielle, application aux permutations. Bulletin de la Société Mathématique de France 16, 176–183 (1888)
10. Lalami, M., El-Baz, D.: GPU Implementation of the Branch and Bound Method for Knapsack Problems. In: IEEE 26th Intl. Parallel and Distributed Processing Symp. Workshops PhD Forum (IPDPSW). pp. 1769–1777. Shanghai, CHN (May 2012)
11. Leroy, R., Mezmaz, M., Melab, N., Tuyttens, D.: Work Stealing Strategies For Multi-Core Parallel Branch-and-Bound Algorithm Using Factorial Number System. In: Programming Models and Applications on Multicores and Manycores (PMAM). pp. 111–119. Orlando, FL (February 2007)
12. McCaffrey, J.: Using permutations in .NET for improved systems security (2003)
13. Melab, N.: Contributions à la résolution de problèmes d'optimisation combinatoire sur grilles de calcul. LIFL, USTL (November 2005), thesis HDR
14. Melab, N., Chakroun, I., Bendjoudi, A.: Graphics processing unit-accelerated bounding for branch-and-bound applied to a permutation problem using data access optimization. Concurrency and Computation: Practice and Experience 26(16), 2667–2683 (2014)
15. Mezmaz, M., Melab, N., Talbi., E.G.: A grid-enabled branch and bound algorithm for solving challenging combinatorial optimization problems. In: 21th IEEE Intl. Parallel and Distributed Processing Symp. (IPDPS). pp. 1–9. Long Beach, CA (March 2007)
16. Mezmaz, M., Leroy, R., Melab, N., Tuyttens, D.: A Multi-Core Parallel Branch-and-Bound Algorithm Using Factorial Number System. In: 28th IEEE Intl. Parallel & Distributed Processing Symp. (IPDPS). pp. 1203–1212. Phoenix, AZ (May 2014)
17. Taillard, E.: Benchmarks for basic scheduling problems. Journal of Operational Research 64, 278–285 (1993)