IVM-based parallel branch-and-bound using hierarchical work stealing on multi-GPU systems

Jan Gmys, Mohand Mezmaz, Nouredine Melab, Daniel Tuyttens

IVM-based Parallel Branch-and-Bound using Hierarchical Work Stealing on multi-GPU Systems

Jan Gmys[1,2], Mohand Mezmaz[1], Nouredine Melab[2], and Daniel Tuyttens[1]

[1]Mathematics and Operational Research Department (MARO), University of Mons, Belgium
[2]Université Lille 1, INRIA Lille – Nord Europe, CNRS/CRIStAL, France

Abstract

Tree-based exploratory methods, like Branch-and-Bound (B&B) algorithms, are highly irregular applications which makes their design and implementation on GPU challenging. In this paper, we present a multi-GPU B&B algorithm for solving large permutation-based combinatorial optimization problems (COPs). To tackle the problem of the irregular workload, we propose a hierarchical work stealing (WS) strategy that balances the workload inside the GPU and between different GPUs and CPU cores. 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. Two variants of the algorithm, using the same hierarchical WS strategy, are proposed: one for COPs where the evaluation of nodes is costly and one for fine-grained problems. The latter variant uses a new hypercube-based WS strategy and a trigger mechanism to balance the work load inside the GPU. The proposed approach has been extensively experimented using the flowshop scheduling, the n-queens and the asymmetric travelling salesman problems as test-cases. The reported results show that the proposed hierarchical WS mechanism is capable of handling fine and coarse-grained types of workloads efficiently, reaching near-linear speed-up on up to 4 GPUs for a set of ten flowshop instances and large instances of fine-grained problems.

1 Introduction

Permutation-based optimization or constraint satisfaction problems frequently arise in industrial and economic applications such as routing, scheduling and assignment. Solving such problems consists in finding one or several permutation(s) that minimize/maximize a given cost function [1] or, respectively, satisfy a given set of constraints. The branch-and-bound (B&B) algorithm is one of the most used exact methods to solve combinatorial optimization problems (COPs). This tree-based algorithm implicitly enumerates all possible solutions by dynamically constructing and exploring a tree. This is done using four operators: branching, bounding, selection and pruning. Using the branching operator, the algorithm recursively decomposes the problem into smaller subproblems. A bounding function is used to compute lower bounds on the optimal cost of these subproblems. The pruning operator discards subproblems from the search that cannot lead to an improvement of the best solution found so far. The tree-traversal is guided by the selection operator which returns the next subproblem to be processed according to a predefined strategy. In this paper depth-first search (DFS) strategy is considered.

Backtracking, a fundamental paradigm that is frequently used to solve constraint satisfaction problems, can be seen as a special case of a DFS-B&B algorithm. In this case a feasibility function is used instead of a bounding operator and subproblems which violate at least on constraint are discarded from the search. Throughout the remainder of this paper we generically refer to both algorithms as B&B.

In this paper we study the parallelization of B&B on multi-core systems equipped with multiple graphics processing units (GPUs). Although the pruning mechanism efficiently reduces the size of the search-space the latter often remains very large and only small or moderately-sized instances can be solved in practise. 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 larger problem instances.

The efficient implementation of the B&B algorithm on GPUs is a challenging task because the GPU programming model is at odds with the algorithm's highly irregular nature [1]. The design of parallel B&B algorithms is strongly influenced by the target architecture and the characteristics of the problem being solved [2].

[1]without loss of generality the minimization case is considered in this paper

For instance, if the relative cost of evaluating the lower bounds (or the feasibility) of a subproblem is high, existing GPU-accelerated B&B algorithms in the literature use the GPU to evaluate large pools of subproblems in parallel [3–5]. These approaches use stacks or queues, implemented as linked-lists, to store and manage the B&B tree. As the GPU memory imposes limitations on the use of such dynamic data structures, these linked-list-based approaches perform at least the selection operator on the CPU, requiring costly data transfers between CPU and GPU. While these data transfers can be overlapped with GPU computations, as for example in [4], such approaches are inefficient in fine-grained situations where the evaluation of a node is less costly.

For permutation-based problems where the cost of a node evaluation is low, GPU-based B&B algorithms in the literature perform massively parallel searches on the GPU [6–8]. In these approaches the search space is, for example, implemented using bitset-representations. The principle of these approaches is the following: on the host an initial set of subproblems is generated, each serving as a root for a concurrent search on the GPU. Because of varying thread granularities, load imbalance is one of the major issues faced by such approaches.

In this work we present a B&B algorithm capable of solving both types of permutation problems on multi-core systems equipped with multiple GPUs. Our algorithm is based on the Integer-Vector-Matrix (IVM) data structure [9] which allows to implement all four B&B operators on the GPU. For the IVM-based GPU-B&B algorithm presented in [10] our previous work [11] proposes four work stealing (WS) [12] strategies for balancing the irregular workload inside a single GPU. The major contributions of this paper are the following:

- Revisiting the algorithm presented in [10] for fine-grained permutation problems, i.e., permutation problems where the cost of evaluating the tree nodes is low compared to the tree search itself. The proposed variant of the algorithm uses the same WS mechanism for load balancing as the original algorithm which uses a second level of parallelism to accelerate the bounding operator.

- A new work stealing mechanism which enables the IVM-based algorithm to use a larger number of parallel exploration processes. This strategy is based on a virtual hypercube-topology and uses a trigger mechanism (similar to mechanisms used in [13, 14]) to invoke load balancing phases.

- An extension of the single-GPU B&B algorithm to hybrid multi-core/multi-GPU systems using a hierarchical WS strategy for balancing the workload inside GPUs and between GPUs and CPU cores.

- The performance of the proposed approach is evaluated in an extensive experimental study using three well-known permutation problems with disparate node evaluation costs: the flowshop scheduling problem, the n-queens and the asymmetric travelling salesman problems (ATSP). The tuning of algorithm parameters and the efficiency of the WS approach for the different problems is thoroughly investigated. Possible limitations of the proposed approach are also discussed.

The experimental results show that, for a set of ten flowshop instances our multi-GPU algorithm scales linearly from 1 to 4 GPUs. For these instances, the 4-GPU algorithm provides an average relative speed-up of 43 over a multi-core version of the B&B algorithm using 32 CPU threads. A near-linear speed-up is also achieved for instances of the n-queens and ATSP problems if the problem size is greater than or equal to 18. For the n-queens problem the proposed algorithm explores up to 10^9 nodes per second, which is 10 times the rate achieved by a highly optimized bitset-based sequential backtracking algorithm [15].

This paper is organized as follows. Section 2 presents the B&B algorithm and explains the models used to parallelize this algorithm. Section 3 explains our proposed multi-GPU B&B algorithm: it describes the single-GPU B&B algorithm, the WS strategies used on the intra-device level, the hybrid multi-CPU/multi-GPU algorithm and the hierarchical WS scheme it uses. Finally, in Section 4 we report the obtained experimental results which evaluate the performance of our multi-GPU B&B algorithm and the efficiency of the proposed WS strategies.

2 Parallel Branch-and-Bound

2.1 Serial Branch-and-Bound

B&B algorithms explore the search space of potential solutions (permutations) by dynamically building a tree whose root node represents the initial problem to be solved. Its leaf nodes are possible solutions and internal nodes are subproblems (partial solutions) of the initial problem. The tree's construction and exploration are performed using four operators: branching, bounding, selection and pruning.

At each iteration of the algorithm, the branching operator partitions a subproblem into smaller, pairwise disjoint subproblems. For each generated subproblem a lower bound on its optimal cost is computed using the bounding

operator. The pruning operator eliminates nodes from the search such that only nodes with a lower bound smaller than the best found solution so far are kept for further exploration. The selection operator chooses the next subproblem to be processed according to a predefined search strategy. Because of its memory efficiency only the *depth-first search* (DFS) strategy is considered in this paper.

At implementation, the four B&B operators act on the data structure which stores the generated subproblems. Therefore, the design of the data structure has a great impact on their efficiency. Stacks or deques implemented as linked-lists are often used in existing works related to B&B using DFS. In [9] it is shown that the Integer-Vector-Matrix (IVM) data structure allows to speed up the exploration process and uses a smaller memory footprint than its linked-list counterpart. Thanks to its reduced memory footprint and the fact that it requires no dynamic memory allocations, IVM is well suited for the implementation of DFS on GPUs [10, 11].

2.2 Models for parallel Branch-and-Bound

As mentioned before, the target execution platform and the problem being solved, strongly influence the choice and implementation of the model for parallelizing B&B [2]. A taxonomy of parallel models for B&B algorithms is presented in [16]. This taxonomy is based on the classifications proposed in [17] and [18]. Three important models are (1) the parallel tree exploration (PTE) model, (2) the parallel evaluation of bounds (PEB) model and (3) the parallel evaluation of a bound model. The former two models (PTE and PEB) are used in our GPU-based B&B algorithm.

In the PTE model different search subspaces of the initial problem are explored simultaneously. This can be done either synchronously or asynchronously. In asynchronous mode, the independent B&B processes synchronize and communicate in an unpredictable manner. In synchronous mode the algorithm has different phases between which the B&B-processes may exchange information, like work units, control messages or the best solution found so far. On SIMD architectures the implementation of the PTE model is necessarily synchronous [13], meaning that the algorithm consists of several phases during which the B&B operators are performed in lockstep by the parallel B&B processes. Load balancing therefore needs to be done on a global scale. Compared to others, the PTE model is more frequently used. One important reason is that the degree of parallelism may be very high, in particular for large problem instances. Indeed, the degree of parallelism is only limited by the capacity of supplying the independent B&B processes with subproblems to explore. As the shape of the B&B tree is unpredictable and, in general, highly irregular, this work supply strongly depends on the efficiency of the used load balancing mechanism.

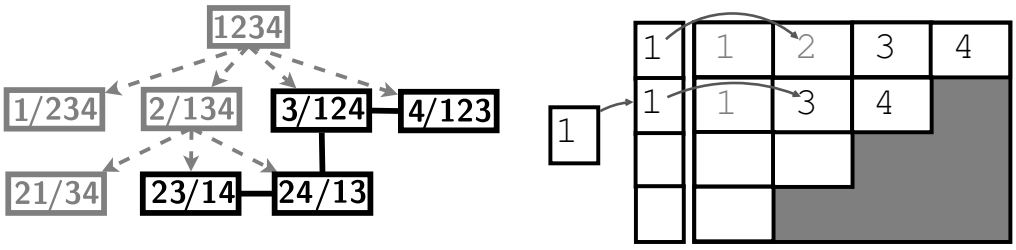
In the PEB model the lower bounds, or the feasibilities, of generated subproblems are evaluated in parallel. The degree of parallelism in this model depends on the problem size and the depth of the evaluated node in the tree. The PEB model is well-adapted in cases where the cost of the node evaluation function is high, compared to the rest of the algorithm. The model is naturally synchronous and data-parallel which makes it a suitable candidate for GPU computing. The GPU-accelerated B&B-algorithms proposed in [4,19] are based on the PEB model, offloading pools of subproblems for bounding to the GPU. It is possible to combine the PEB model with the PTE model by parallelizing the bounding operator of each independent tree exploration worker. Our algorithm may either use the combined PTE+PEB model or the PTE model alone. The choice of the model depends on the relative cost of the used node evaluation function, in terms of memory requirements and computation.

2.3 GPU-accelerated Branch-and-Bound

For instance, when B&B is used in the context of integer linear programming the evaluation of each node consists in solving a linear program. For such problems only few works in the literature [5, 20] consider GPU-acceleration, as sophisticated CPU-based LP-solvers are often able to compute lower bounds more efficiently than GPU-based LP-solvers, especially for large problem instances involving sparse matrices.

For COPs like the flowshop [3] or jobshop [21] scheduling problems the used bounding function is computationally intensive (up to 99% of the sequential algorithm [19]) but fine-grained enough to make use of a GPU-accelerated parallel evaluation of one or several lower bounds. For such problems, GPUs can be used to substantially accelerate B&B, offloading large pools of subproblems to the device for parallel evaluation. However, in scenarios where the bounding operator is less time-consuming, this approach becomes less effective because the pool management is performed in a sequential, or weakly parallel manner on the CPU. As the relative cost of the pool management increases it cannot be completely overlapped by the bounding phases performed on the GPU.

Therefore, GPU-based approaches for fine-grained problems aim at performing massively parallel tree searches, i.e., they use the PTE model without parallelizing the evaluation of bounds. In general, mapping the backtracking paradigm to the GPU has been recognized as a very challenging task [1]. Successful implementations of backtracking on the GPU depart from the general case and use some of the problem's properties, for instance an a-priori knowledge



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

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

of the tree structure. Examples include kd-tree traversal in raytracing [22], game tree search [23] and permutation problems like the asymmetric travelling salesman problem (ATSP) [24] or the n-queens problem [25].

For permutation problems bitset representations of the search space can be used. Such data structures allow an extremely fast and lightweight implementation of DFS [15] and they have been successfully used in GPU-accelerated B&B algorithms [25]. A common approach is to perform an initial search on the CPU, generating an initial set of subproblems which are used as root nodes for concurrent searches on the GPU, each search being usually performed by one GPU thread [8, 23–25]. Although good speedups for bitset-based GPU-backtracking algorithms have been reported, the static initial repartition often leads to load balance issues, especially in highly irregular and larger problem instances.

In this paper we present a B&B algorithm for multi-core/multi-GPU systems using a hierarchical work stealing (WS) mechanism for load balancing inside the GPUs and between CPUs and GPUs. We present two versions of the B&B algorithm, both using the same load balancing mechanism. The first version corresponds to the one presented in our earlier work [10, 11] and uses a combined PTE+PEB model. The efficiency of this algorithm is evaluated using the flowshop scheduling problem (FSP) as a test case. The second version uses the PTE model without parallelizing the bounding operator. Its efficiency and the performance of the proposed work stealing mechanism are evaluated by solving instances of the n-queens problem using backtracking and the ATSP problem using the partial cost of a subproblem as its lower bound.

3 IVM-based Branch-and-Bound for multi-GPU

This section describes the B&B algorithm using Integer-Vector-Matrix (IVM) for a single GPU. IVM is a data structure dedicated to permutation problems, that is introduced in [9]. An example of a pool obtained when solving a permutation problem of size 4 is used, in Subsection 3.1, to explain the IVM-based pool management. This subsection also introduces the factorial (or factoradic) number system which is used for the handling of work units. In Subsection 3.2 we present the two variants of our IVM-B&B algorithm for a single GPU device. In Subsection 3.3 our approach for load balancing on device-level is presented, and Subsections 3.4 and 3.5 explain our multi-core/multi-GPU algorithms and the host-level work stealing approach.

3.1 Serial IVM-based B&B

When a stack is used for DFS the selection operator consists in taking the first element from the stack. Branching consists in generating child nodes and, possibly, allocating them in memory before inserting them into the stack. Unlike stacks the IVM-structure requires no dynamic memory allocations to manage the pool of subproblems. IVM is illustrated in Figure 1 using as an example a pool obtained when solving a permutation problem of size $n = 4$. The left-hand side (Figure 1a) shows a tree-based representation using dashed arrows. It also shows a linked-list-based representation using horizontal and vertical solid lines. In this representation each node is a partial (subproblem) or a full permutation (solution). The elements before the "/" symbol are already scheduled while the following ones remain to be scheduled.

The right side (Figure 1b) represents the corresponding IVM structure, indicating the next subproblem to be processed. The integer I indicates the level of this subproblem. In this example the current level of the search is 1. The so-called position-vector V indicates the elements that are already scheduled (2 and 3 in this example). The matrix M contains the elements that remain to be scheduled at each level. At level 0 all n elements remain to be

Algorithm 1 Select-and-Branch

```
1: procedure SELECT-AND-BRANCH(state)
2:     state←EMPTY
3:     while (V ≤ endV) do
4:         if (row-end) then                          ▷ (V(I) > I)?
5:             cell-upward                            ▷ I − −; V(I) + +
6:         else if (cell-eliminate) then              ▷ M(I, V(I)) < 0?
7:             cell-rightward                         ▷ V(I) + +
8:         else
9:             state←EXPLORING
10:            break
11:        end if
12:    end while
13:    if (state = EXPLORING) then
14:        generate-next-line (branch)
15:    end if
16: end procedure
```

Algorithm 2 Serial IVM-based B&B

```
1: procedure SERIAL B&B
2:     initialize-intervals
3:     state←EXPLORING
4:     while (state≠EMPTY) do
5:         select-and-branch(state)
6:         if (state=EXPLORING) then
7:             bound-and-prune
8:         end if
9:     end while
10: end procedure
```

scheduled. After selecting one element on that level, $n-1$ elements remain to be scheduled (in this example 1, 3 and 4). Some of the B&B operators have been revisited for the IVM structure as follows:

- The branching of a node (decomposition) consists in increasing the current level (incrementing I) and copying all unscheduled jobs from row $I-1$ to row I in the matrix.
- To eliminate (prune) a subproblem the corresponding cell is skipped, meaning that $V(I)$ is incremented if possible, otherwise I is decremented and $V(I)$ is incremented. To flag a cell as "pruned" its value is multiplied by -1. Using this convention the branch procedure actually consists in copying the absolute values to the next row, i.e., copying $-j$ as j and j as j. Remembering the state of the node is necessary if the child nodes are evaluated in parallel.
- The selection operator is performed by setting I and V such that they point to the next subproblem to be evaluated.
- The bounding operator associates a lower bound to a subproblem and does not act on the data structure. Therefore it remains unchanged.

.

The `select-and-branch` procedure described in Algorithm 1 modifies the IVM structure such that it indicates the next subproblem to be decomposed, according to the DFS strategy. It sets the variable `state` to *exploring* if a node was found, and to *empty* if no node was found, meaning that the search has ended. Algorithm 2 shows the pseudo-code for a serial B&B algorithm.

During the exploration process the vector V behaves like a factoradic counter. In the example of Figure 1b, V successively takes the values *0000*, *0010*, *0100*, ..., *3200*, *3210*. These 24 values correspond to the numbering of the 4! solutions using the factoradic (or factorial) numbering system [26] in which the weight of the i^{th} position is equal to $i!$ and the digits allowed for the i^{th} position are $0, 1, \ldots, i$. As it can be seen in line 3 of Algorithm 1, the `select-and-branch` procedure ends with an *empty* state-variable if and only if V has reached a maximum value, designated by `endV` in line 3. Therefore, it is possible to have two IVMs $R1$, $R2$ such that $R1$ explores $[0000, X[$ and $R2$ explores $[X, 3210]$[2]. These factoradic intervals are used as work units, instead of 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 $[0, N![$.

In order to enable a B&B-process to explore any arbitrary interval $[B, E]$ an initialization procedure, as described in [10] is necessary. However it is possible to do the interval splitting between threads in such a way that no initialization procedure by the receiving IVM (thread) is required. For all interval sharing operations on device-level operation, this procedure is used. A detailed description of this interval-splitting procedure and factoradic-based work units can be found in [27].

3.2 single-GPU-based parallel B&B

In the GPU-based algorithm a fixed number T of IVM structures is allocated in device memory and the B&B operators act on these data structures in a data-parallel way. A fundamental design choice is whether to place

[2]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.

Algorithm 3 IVM-B&B for single GPU

```
1: mode∈{PTE,PTE+PEB}
2: procedure GPU-B&B(mode)
3:     while not-end do
4:         gpu-next(mode,end)
5:         steal-intra-gpu
6:     end while
7: end procedure
8: function GPU-NEXT(mode,end)
9:     switch mode do
10:        case PTE+PEB:
11:            kernel select-and-branchGPU
12:            kernel reduce(end)
13:            kernel buildMapping
14:            kernel bound
15:            kernel prune
16:        case PTE:
17:            trigger←0
18:            kernel bnb-step(trigger)
19:            kernel reduce(end)
20: end function
21: function STEAL-INTRA-GPU
22:     kernel computeIntervalLengths
23:     kernel selectVictim
24:     kernel stealWorkIntraGPU
25: end function
```

Algorithm 4 DFS kernels

```
1: kernel select-and-branchGPU
2: ivm←mapping(blockIdx,threadIdx)
3: select-and-branch(ivm)                          ▷ cf. Alg. 1
4: end kernel
5: kernel bnb-step(trigger)
6: ivm←mapping(blockIdx,threadIdx)
7: do
8:     select-and-branch(ivm)                      ▷ cf. Alg. 1
9:     if (state=EXPLORING) then
10:        boundNodes(ivm)
11:    else
12:        atomicIncrement(trigger)
13:    end if
14: while (trigger<WS-TRIG ∧ state=EXPLORING)
15: end kernel
```

the outer while-loop (Algorithm 2, line 5) inside a kernel, i.e., to try merging the entire algorithm into a single CUDA-kernel, or, to keep this loop on the CPU-side, i.e., to launch several kernels inside this loop and return to the host after each step of the algorithm. The former option seems to have some advantages: kernel launch overheads are avoided and data needs to be loaded from global memory to shared memory and registers "only once".

However, all workers (IVMs) must communicate in order to exchange work units, which requires inter-block synchronization. Also, merging the whole exploration process into a single kernel may, depending on the complexity of the bounding operator, lead to a very high register usage and therefore performance limitations. This is typically the case when the cost of the bounding operator is high, making a parallelization of the bounding operator profitable.

In [10] a GPU-based B&B algorithm using such a two-level parallelization (combined PTE+PEB model) is presented. The pseudo-code of this algorithm corresponds to selecting the PTE+PEB mode in line 1 of Algorithm 3. Choosing mode "PTE" in line 1, the pseudo-code shown in Algorithm 3 corresponds to the algorithm's variant which targets fine-grained problems where the bounding function has a small cost compared to the rest of the algorithm.

For both versions the GPU-B&B algorithm consists in alternating a node expansion phase (line 4) and a work stealing phase (line 5). This is quite similar to the SIMD tree-search a presented in [13]. In the context of GPU computing, Lauterback *et al.* also use work balancing kernels, in combination with work queues.

For the variant using parallel bounding (PTE+PEB), the node expansion phase `gpu-next` starts by calling the kernel `select-and-branchGPU` (Algorithm 4, line 1). It consists in performing the select-and-branch procedure (Algorithm 1) in parallel for all T IVMs. Then, in order to detect whether all IVMs have finished exploring their intervals, a parallel reduction (line 12) is performed on the T state-variables. It also uses a min-reduce to determine the global best solution found so far. In the bounding kernel (line 14) all generated subproblems are evaluated in parallel. As the number of children per non-empty IVM is variable, a static mapping of threads onto children subproblem leads to a high number of idle threads. Therefore, the kernel `buildMapping` (line 13) performs a compaction of the mapping of threads onto children subproblems using a parallel prefix-sum [28] computation.

In the PTE-only version of the algorithm (i.e., without parallel bounding) the select-and-branch kernel is modified. This modified search-kernel is called `bnb-step` in the pseudo-code. It is shown in Algorithm 4 (lines 5-15). It incorporates the bounding and pruning operators and allows IVMs to perform several B&B-iterations during its execution.

In order to achieve load balancing on a global scale a triggering mechanism is used. Such triggering mechanisms have been proposed for parallel tree-search algorithms on SIMD computers, for instance the CM-2 computer [13]. The mechanism is simple: for each IVM-explorer that finds its interval empty, a counter (`trigger`) is atomically incremented in global memory. When the value of `trigger` reaches a fixed value `WS-TRIG` all threads exit the kernel. Then, a parallel reduction is performed on the IVM-states, as for the PTE+PEB version.

After this, the algorithm enters a work stealing phase, which is identical for both versions of the algorithm (Algorithm 3, line 5). This phase consists of two steps. First, in a victim selection phase a one-to-one mapping of empty IVMs onto suitable victim IVMs is built. Then, in the `steal` kernel empty IVMs acquire work in parallel

from the corresponding victim IVMs. The thief-to-victim mapping must satisfy the following conditions:

To avoid unpredictable behaviour, it must guarantee that no victim-IVM is selected twice during the same WS phase. Also, only IVMs in the *exploring* state are allowed to be selected as WS victims. Moreover, the victim selection should (1) induce minimal overhead, meaning that the mapping must be built in parallel, (2) select victim IVMs whose intervals are likely to contain more work than others, (3) serve a maximum of empty IVMs during each WS phase.

The following subsection provides a more detailed description of the device-level work stealing phase.

3.3 Device-level work stealing strategies

In our previous work [11] four work stealing (WS) strategies for load balancing inside the GPU have been proposed and experimentally evaluated. These strategies are:

- The *Ring* strategy where victims are selected in round-robin fashion.
- A ring-based *Search* strategy, where an empty IVM j successively checks whether work can be stolen from its neighbors $j-1$, $j-2$, ..., $j-s \pmod T$, in a search-window of fixed length s.
- A *Large* strategy which takes into account the length of an interval. In this strategy only IVMs with an interval-length greater than the global mean interval-length are eligible WS victims.
- An adaptive strategy that shifts the beginning of the search window at each iteration and dynamically adjusts the length of this search window according to the current number of empty IVMs. It also uses the length-criterion. This strategy is called *Adapt* throughout the remainder of this paper.

All these victim selection strategies, except *Ring*, are used in combination with a steal-half granularity policy, meaning that the right half of a victim's interval is stolen. The experimental results in [11] show that the *Adapt* strategy produces the best results among these four strategies. Using $T \approx 1000$ IVMs in the PTE+PEB version of the GPU-B&B algorithm it allows to keep on average more than 97.5% of the explorers busy while consuming less than 2% of the algorithm's execution time. The pseudo-code for the *Adapt* selection policy is shown in Algorithm 6.

For the problem instances and the parallel model (PTE+PEB) considered in [11], $T \approx 1000$ IVMs generate enough subproblems per iteration to saturate the device during the bounding phase, which amounts for $> 90\%$ of the elapsed execution time. When no parallel bounding is used, the number of used IVMs T needs to be substantially higher. For instance, to achieve about 50% occupancy with the PTE version on a GeForce GTX 980 GPU, at least $T \approx 15000$ IVMs should be used (0.5 $\times$16SMs $\times$ 64warps $\times$ 32threads).

The efficiency of the *Adapt* strategy for the PTE-only version of the algorithm and a much larger number of IVMs should therefore be re-examined. Also, we propose another WS strategy, based on a hypercube topology.

The search-window used by *Adapt* aims at reducing the number of WS phases required to exchange work units between two arbitrary IVMs. However, the underlying topology of the *Adapt* strategy is an oriented ring, i.e., IVM $j > 0$ is seen as the successor of $j-1$ and IVM 0 is seen as the successor of IVM $T-1$. The maximum distance between two workers in this ring is $T-1$. Another way of reducing the distance between IVMs is to use a topology with a lower diameter.

In a 2D-ring or torus topology, for instance, the IDs of $T_1 \cdot T_2$ IVMs are written as couples (r_1,r_2) $(0 \leq r_i < T_i)$ with IVM (r_1,r_2) successively trying to steal from its two neighbours $(r_1-1 \pmod{T_1}, r_2)$ and $(r_1, r_2-1 \pmod{T_2})$.

This can be further generalized, writing an IVM-ID as a m-tuple $(\alpha_m, \alpha_{m-1}, \ldots, \alpha_i, \ldots, \alpha_1)$ $(0 \leq \alpha_i < L_i)$. Connecting all workers whose ID differ in exactly one digit, i.e., that are within Hamming distance 1, a m-dimensional hypercube is obtained. In general the nodal degree in this topology is $\sum_{i=1}^{m}(L_i - 1)$. If we have $\forall i : L_i = p$ (meaning the the IVM-IDs are written in base p) each of the $T = p^m$ IVMs has $m(p-1)$ neighbours. In the proposed *Hypercube* victim selection strategy all IVMs attempt to select one of its neighbours. As this is done synchronously and in a certain order no double selection of a victim can occur. In Algorithm 5 the pseudo-code for the *Hypercube* selection policy is shown.

To make the pseudo-code more readable the operation in line 12 of Algorithm 5 is written in terms of hypercube coordinates. However, in practise it should be carried out in terms of integer operations. To do this $(\alpha_m, \ldots, \alpha_1)$ with $0 \leq \alpha_i < L_i$ is seen as a number in a number system where the i^{th} position has weight $w_1 = 1, w_i = \prod_{j=1}^{i-1} L_i$. Using these weights w_i each IVM-ID $r = 0, \ldots, T-1$ has a unique representation $r = \sum_{i=1}^{m} \alpha_i w_i$ [29] and we have that:

$$\text{The operation } \Big\{ \textbf{return } (\alpha_m, \ldots, (\alpha_i - j) \pmod{L_i}, \ldots, \alpha_1)$$

Algorithm 5 Victim selection: *Hypercube*

```
1: procedure SELECT-HYPERCUBE
2:     for (i : 1 → m) do
3:         for (j : 1 → L_i-1) do
4:             select-hyper<<<T threads>>>(i,j,...)
5:         end for
6:     end for
7: end procedure
8: function __GLOBAL__ SELECT-HYPER(i, j, ...)
9:     ivm ← blockIdx.x*blockDim.x + threadIdx.x
10:    if (state[ivm]=empty) then
11:        %%%                          ▷ V is an integer ∈ [0, T − 1]
12:        V ← (α_m, ..., (α_i − j) mod L_i, ..., α_1)
13:        if ((state[V]=exploring) ∧ (flag[V]= 0) ∧
   (length[V]>meanLength)) then
14:            victim-map[ivm]← V
15:            flag[V]← 1
16:        end if
17:    end if
18: end function
```

Algorithm 6 Victim selection: *Adapt*

```
1: procedure ADAPTSELECT
2:     B←(B+S) (mod T)
3:     if ((nbEmpty > nbIVM/10)∧(S≤ nbIVM/2)) then
4:         S←S*2;
5:     else if (S> 1) then
6:         S←S/2;
7:     end if
8:     for (k = B→ B + S) do
9:         try-select<<<T threads>>>(k, ...)
10:    end for
11: end procedure
12: function __GLOBAL__ TRY-SELECT(k, ...)
13:    ivm ← blockIdx.x*blockDim.x + threadIdx.x
14:    if (state[ivm]=empty) then
15:        V ← ivm-k (mod T)
16:        if ((state[V]=exploring) ∧ (flag[V]= 0) ∧
   (length[V]>meanLength)) then
17:            victim-map[ivm]← V
18:            flag[V]← 1
19:        end if
20:    end if
21: end function
```

$$\text{is equivalent to } \begin{cases} \textbf{return } r - j \times w_i, & \text{if } \frac{r}{w_i} \pmod{L_i} \geq j \\ \textbf{return } r - j \times w_i + L_i \times w_i, & \text{otherwise} \end{cases}$$

For $T = L_m L_{m-1} \cdots L_1$ IVMs a m-dimensional hypercube topology can therefore be defined using two tables: $[L_m, L_{m-1}, \ldots, L_1]$ and the weights $[w_m, w_{m-1}, \ldots, w_1]$. Although the algorithm is not limited to any particular number of IVM structures, we use only powers of 2 as values for T and we choose the L_i's such that the product $T = L_m L_{m-1} \cdots L_1$ contains the factor 4 as many times as possible but not the factor 2. For example, using $T = 512 = 2^9$ IVMs, a 4-dimensional hypercube topology is defined by the tables $\{L_i\} = [8, 4, 4, 4]$ and $\{w_i\} = [2^6, 2^4, 2^2, 1]$.

3.4 Multi-core/multi-GPU-based parallel B&B

In this subsection we explain how the to extend the previously described GPU-B&B algorithm for using multiple GPUs in combination with multi-core CPUs. It should be noted that the presented algorithm only targets shared memory systems.

The algorithm launches G CPU threads controlling one GPU each and/or C CPU threads which do not use a GPU. The CPU-threads explore a single interval using a serial B&B algorithm as described in Algorithm 2 and each GPU-explorer uses a variant of the GPU-B&B algorithm as described in Subsection 3. For the sake of simplicity we suppose that all GPU-explorers use an identical number of IVMs (T). These threads are implemented using the POSIX thread library. Each of them independently explores its assigned work unit(s) until there is no more work locally available.

Algorithm 7 describes a thread of the multi-core/multi-GPU B&B algorithm. It is possible to have several CPU-threads use the same device, concurrently issuing kernels to different streams. However, each worker should generate a sufficient workload to fill one device on its own.

At the beginning each GPU-thread sets the device it uses and creates a CUDA-stream. GPU-threads are numbered from 0 to $G - 1$ and CPU-threads from G to $G + C - 1$

Initially, the entire interval $[0, N![$ is assigned to IVM 0 of thread $r = 0$. While it is possible to use an initial distribution of this interval among workers, the work stealing mechanism should be capable of quickly distributing this initial work unit among all the workers. Moreover, we found that an initial partitioning of $[0, N![$ only leads to very marginal improvements. Looking at the first iterations with an initial partitioning we notice that many workers discover that their assigned interval contains only pruned nodes, after which the algorithm quickly reaches a state similar to the one which is attained with the described initialization.

On the inter-device level the algorithm works like an asynchronous parallel tree exploration algorithm for multi-core CPUs [30]. While there exists at least one non-empty interval, *B&B-thread* applies a new iteration. In this iteration *B&B-thread* checks the status of its intervals. Three scenarios can occur:

- ***intervals-empty:*** This is true for thread r if and only if the state of all IVMs handled by thread r is *empty*. In this case, the thread r runs the *detect-end(r)* function (described below) to check whether all other threads

Algorithm 7 Pseudo-code for the multi-GPU IVM-based B&B algorithm.

```
1: r← get-ID
2: type∈{D(evice),H(ost)}
3: procedure B&B-THREAD(R)
4:     while (exist-non-empty-interval) do
5:         if (intervals-empty) then
6:             detect-end(r)
7:             r' ← select-victim(r)
8:             send-request(r')
9:             wait-receive(r')
10:        else if (got-request) then
11:            send-buf←intervals-steal(type,requester-type)
12:            intervals-send(send-buf,requester)
13:        else
14:            if (type=D) { gpu-next(mode,end)
                           { steal-intra-gpu (Algorithm 3, lines 4-5)
15:            if (type=H) { cpu-next(end) (Algorithm 2, lines 6-9)
16:            intervals-empty←update-control-variables
17:        end if
18:    end while
19:    gather-results(r)
20: end procedure
```

are also in this state. Unless the end of the algorithm is detected, the thread r sends a work-request to the thread r' chosen by the *select-victim* function. This is done by pushing its thread-ID r into the request-queue (FIFO) of thread r'. Each thread maintains a request-queue to handle multiple requests. After sending the request to thread r', thread r waits for an answer.

- **not *intervals-empty* and *got-request*:** Thread r has at least one non-empty interval and its request-queue is not empty. In this case, thread r shares its intervals in FIFO order with all threads the made a request. The *share-intervals* function is used to divide one or more intervals of thread r, storing the resulting right halves in a *send-buffer*. The definition of this work splitting operation depends on the type of thread r and the type of the requester thread. They are described in Subsection 3.5. Depending on the type of operation *send-buffer* is sent to the requester either via shared memory, or using the *cudaMemcpyPeerAsync*/*cudaMemcpyAsync* functions. Upon completion of the send-operation the waiting thread is released and waits if necessary for the copy operation to complete.

- **not *intervals-empty* and not *got-request*:** At least one of thread r's intervals is not empty and no request is received. In this case the thread moves on in the exploration process and, in the case of a GPU-thread, launches a GPU-internal load balancing phase, as described in Subsection 3. After each node expansion phase some control variables are sent to the host (***send-control-variables***, line 19). These control variables are *intervals-empty*, the best solution cost and the current number of empty and exploring IVMs.

The synchronization of B&B-threads is achieved using POSIX threads implementation of mutexes and semaphores. Aside from the inter-thread communication of work units, the algorithm takes advantage of shared data for two critical components of B&B-algorithms: termination detection and distribution of the best solution found so far. Algorithm 7 deals with these two issues as follows:

- ***detect-end*:** This function (line 8) atomically increments a global counter-variable and compares its value to the number of B&B-threads $G + C$. If both values are equal this means that thread r has detected the end of the algorithm and that all other threads are waiting for work. Therefore the detecting thread r sets the shared variable *exist-non-empty-intervals* (line 4) to *false* and releases all waiting threads. This is necessary to allow all threads to reach the function ***gather-results*** (line 24) where the detecting thread gathers exploration results and statistics for evaluation purposes.

- ***update-best-solution*:** In this function the calling thread compares its local-best solution to the shared global-best solution and updates both to the smaller cost if necessary.

In a distributed system both these components can be challenging to handle. Also, when solving large problem instances checkpointing mechanisms should be used in order to add fault tolerance.

3.5 Host-level work stealing strategies for multi-core/multi-GPU B&B

The load balancing scheme for the multi-GPU algorithm uses a work stealing (WS) approach that is hierarchical in the sense that local WS (inside each GPU) has a strict priority over global (inter-thread) WS operations. An

inter-thread WS attempt will only be initiated by a GPU-thread if all its T IVMs are empty. To select another GPU- or CPU-explorer as WS victims, the well-known and widely used *random* victim selection strategy is used [12]. In [30] it is experimentally shown that it achieves good results in the context of a multi-core B&B using factoradic work units. Using CPU- as well as GPU-explorers four types of work stealing operations must be defined: host-to-host (H2H), device-to-device (D2D), device-to-host (D2H) and host-to-device (H2D). These operation must take into account that a CPU-explorer thread handles a single IVM structure while a GPU explorer handles T IVM structures.

- In a H2H WS operation the thief steals the right half of the victims interval. Stealing the right half of an interval $[A, B]$ means that the victim keeps $[A, (A + B)/2]$ and the thief starts exploring the interval $[(A + B)/2 + 1, B]$.
- A D2D WS operation is defined as follows: the i^{th} IVM of the thief attempts to steal the right half-interval from IVM i of the victim GPU.
- In a H2D WS operation a part of the host's interval is send to the GPU and assigned to IVM 0 of the thief GPU. In order to take into account the heterogeneity of the thief and the victim the granularity policy can be adjusted according to the relative power of the explorers. For instance, a thief steals $[(1-\gamma)A+\gamma B, B]$ where $\gamma \in [0, 1]$ is set to $1/k$ if the thief is k times more powerful than the victim. We experiment the steal-half and the steal-1/k granularity policies.
- Finally a D2H WS operation is defined as follows: the thief (host) randomly selects one of the victim's IVMs and checks the status of its interval. If the status is *empty* the steal attempt fails. Otherwise the thief steals the *whole* interval, including the IVM structure. This seems reasonable as this interval represents on average only $1/T^{th}$ of the victim's total amount of work. Moreover, this aims at reducing the overhead for the victim : it uses an asynchronous copy to send the stolen IVM to the thief and sets the status of the corresponding IVM-ID to *empty*.

4 Experiments

4.1 Hardware/Experimental protocol

All the experiments are run on a computer composed of two 8-core Haswell E5-2630v3 processors, and four GeForce GTX 980 GPUs. The four Maxwell GPUs, based on the GM204 architecture are of compute capability 5.2. Version 7.5 of the CUDA Toolkit is used. The operation system installed is a CentOS 7.1 Linux distribution. Peer access for GPUs attached to the same CPU is enabled. To measure the elapsed execution time we use the *clock_gettime* function.

The execution times for different problem instances vary strongly because of varying tree sizes and node evaluation costs. In order to facilitate the presentation of the experimental results, most of them are reported in terms of achieved average node processing speed, i.e., decomposed nodes per second. For this performance metric we use the units *kn/s* (designating 10^3nodes/second) and *Mn/s* (designating 10^6nodes/second).

4.2 Test problems

Flowshop: The permutation flowshop scheduling problem (FSP) is defined by a set of n jobs and m machines, arranged in a certain order. As illustrated in Fig. 2, jobs are processed according to the chain production principle, meaning that a job cannot be processed on a machine j before it has finished processing on all upsteam machines $0, 1, \cdots, j-1$. The n jobs have to be processed in the same order on each machine, and the processing of a job cannot be interrupted. A $m \times n$ procssing time matrix contains the time required for a machine to finish the processing of a job. The goal is to find a permutation schedule that minimizes the total processing time called *makespan*. In [31], it is shown that the minimization of *makespan* is NP-hard from 3 machines upwards. The lower bound proposed by Lageweg *et al.* [32] is used in our bounding operator. This bound is known for its good results and has complexity of $O(m^2 n log(n))$.

In our experiments, the FSP instances defined by Taillard [33] are used to validate our approach. These instances are divided into 12 groups: 20x5 (i.e. group of instances defined by $n = 20$ jobs and $m = 5$ machines), 20x10, 20x20, 50x5, 50x10, 50x20, 100x5, 100x10, 100x20, 200x10, 200x20 and 500x20. For instances where m is equal to $m = 5$ or $m = 10$ the used bounding operator gives such good lower bounds that they can be solved within a few seconds using a sequential B&B. Instances where $m = 20$ and the number of jobs is equal to $n \geq 50$ are very hard

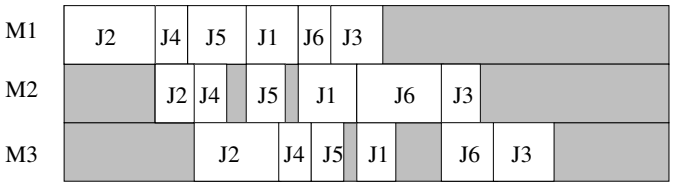


Figure 2: Example of a solution of a flowshop problem instance defined by 6 jobs and 3 machines.

to solve. For example, the resolution of $Ta056$ in [34], which is an instance of the 50×20 group, lasted 25 days with an average of 328 processors and a cumulative computation time of about 22 years. Therefore, in our experiments, the validation is performed using the 10 instances that belong to the group 20x20 (*Ta021–Ta030*).

When an instance is solved twice using a parallel tree search algorithm, the number of explored subproblems often differs between the two resolutions. However, to compare the performance of two B&B algorithms, both should explore exactly the same tree. Therefore, we choose to always initialize our algorithm by the optimal cost of the instance being solved. This initialization ensures that the algorithm explores exactly the critical subtree and allows us to evaluate its performance in absence of speedup anomalies. Using a sequential B&B algorithm the resolution time for the ten FSP instances *Ta021–Ta030* ranges from about 10 minutes to 16 hours using a single core of the E5-2650v3 CPU.

N-Queens: The n-queens problem of placing n nonattacking queens on a $n \times n$ chessboard is often used as a benchmark for algorithms that solve constraint satisfaction problems. We use the version of n-queens that consists in finding *all* solutions. N-queens is easily modeled as a permutation problem: position i of a permutation of size n designates the column in which a queen is placed in row i. To evaluate the feasibility of a partial solution, a function checks for diagonal conflicts in this partial solution. For $n \leq 14$ the n-queens problem can be solved within fractions of a second by sequential algorithms. The size of the explored tree grows exponentially, so we consider the n-queens problem for n=15–19. Making use of one axial symmetry (i.e., restricting the queen in the first row to columns 1, 2, ..., $(n+1)/2$) the size of the search space is divided by two.

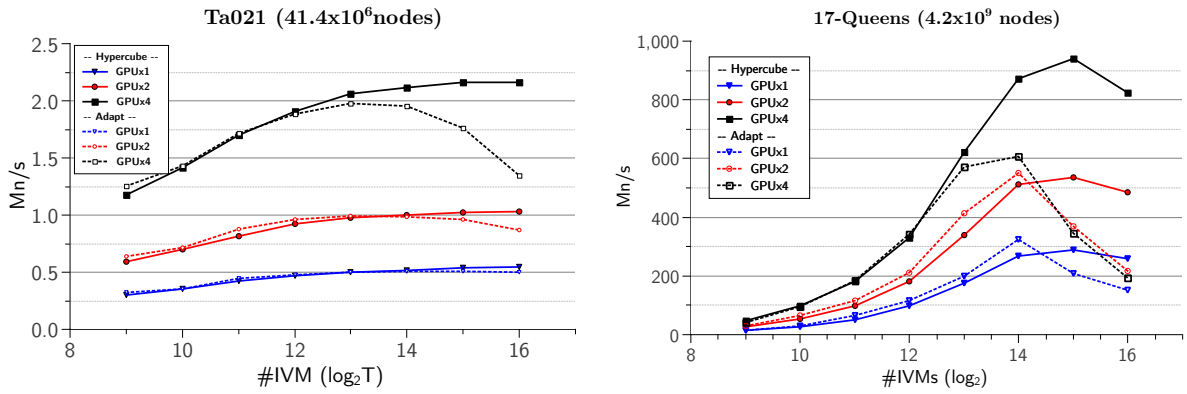
ATSP: The travelling salesman problem (TSP) is a well-known permutation COP. It consists in finding a shortest hamiltonian cycle through a given number of cities such that each city is visited exactly once. In the asymmetric version (ATSP) the cost c_{ij} for going from i to j can be different from the cost c_{ji}. The ATSP instances used in our experiments come from the instance generator proposed in [35]. Two classes of instances are used: *crane*, modelling stacker crane operations and *tsmat*, where the triangular inequality holds. We solve the ATSP problem by using a very simple bounding procedure: at each node the partial cost is evaluated and compared against the best solution found so far. Nodes whose partial cost is greater than the best solution found so far are pruned. Using such a naive bounding operator the number of nodes to explore grows exponentially with the instance size (number of cities). As for the FSP, the algorithm is always initialized at the optimal solution cost.

4.3 Evaluation of multi-GPU-B&B

In this subsection we evaluate the efficiency of both proposed variants of the GPU-B&B algorithm for single and multi-GPU systems. In particular we investigate the impact of the number of used IVM structures, i.e., scaling up the number of parallel tree explorers. We compare the *Adapt* and *Hypercube* WS strategies and study the efficiency of the proposed trigger-mechanism for the PTE variant of the algorithm.

Calibration of the number of used IVMs per GPU: Figure 3 shows the node processing rates (in Mn/s) achieved for different values of T ($T = 2^n, n = 9, 10, \ldots, 15$) and the two studied WS strategies *Adapt* and *Hypercube*. The results are shown for 1, 2 and 4 GPUs without using the CPU cores for exploration. The results on the left-hand side (Figure 3a) are obtained for FSP instance *Ta021*, using the PTE+PEB variant of the algorithm. On the right-hand side (Figure 3b) the achieved node processing rates are shown for the 17-queens problem, using the PTE-only variant of the algorithm. As an ad-hoc value, the WS trigger is set to $T/10$.

These two problem instances are used because they are large enough to justify the use of a multi-GPU system and small enough to be solved repeatedly to collect experimental data. For example, instance *Ta021* requires over 20 minutes of processing using the 16 CPU-threads on the two used multi-core CPUs and about 90 seconds using a single GPU.



(a) FSP - Ta021 (initialized at optimal solution), $41.4{\cdot}10^6$ explored nodes

(b) 17-queens, $4.2{\cdot}10^9$ explored nodes

Figure 3: Node processing rate (Mn/s) for different values of T (#IVMs) and 1, 2 and 4 GPUs. The dotted lines show the results for the *Adapt* stategy and the solid lines for the *Hypercube* strategy

As one can see in Figure 3, both WS strategies allow to achieve similar performances for a single GPU, especially when a small number of IVMs is used ($T < 4096(= 2^{12})$). However, as the number of IVMs increases, the *Hypercube* strategy clearly outperforms the *Adapt* strategy. As the number of GPUs increases (meaning that the total number of IVMs increases) this effect is even more visible.

For both WS strategies and algorithm variants, the node processing rate increases according to the number of IVMs up to $T = 2^{14}$. However, when the *Adapt* strategy is used the processing speed increases at a lower rate for $T \geq 2^{12}$. For $T > 14$ the processing speed achieved with the *Adapt* strategy decreases as more IVMs are used. The reason for this is that the cost of the victim selection becomes very high as the larger number of IVMs causes the strategy to search in large windows. This is not the case in the hypercube-based strategy which contacts a constant number of IVMs to select a WS victim. For the ATSP of similar size (running for about 10 seconds on 1 GPU) the results are similar to those for 17-queens.

Figure 3 clearly shows that the *Hypercube* strategy is better suited for the multi-GPU algorithm, and especially its PTE variant for fine-grained problems. Using *Hypercube* the highest node processing rate achieved with 4 GPUs for the 17-queens problem is about 950 Mn/s for $32,768$ IVMs, which is about 50% more than using the *Adapt* WS which stagnates at 600 Mn/s for $16,384$ IVMs.

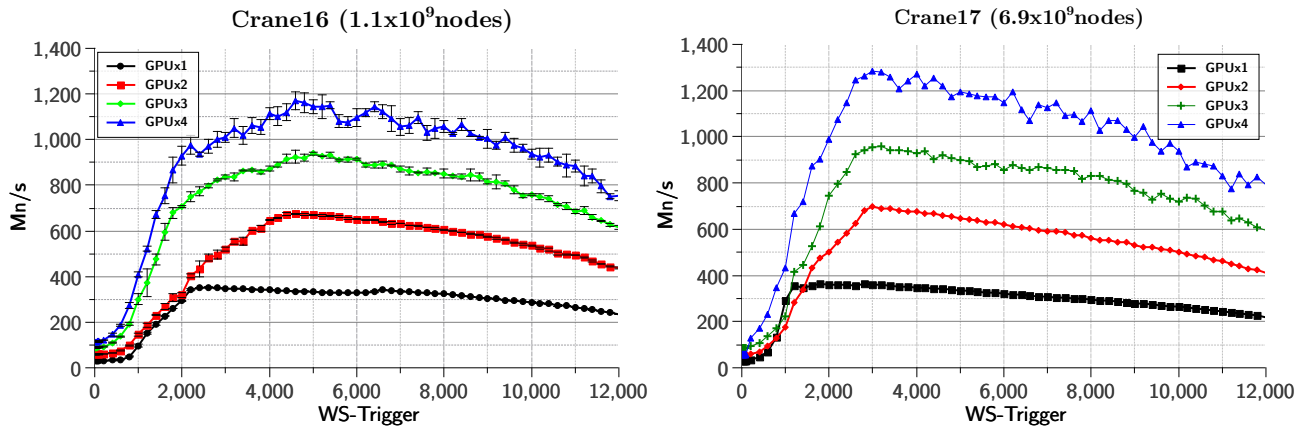
The *Hypercube* strategy allows a better scaling with the number of GPUs and used IVMs, therefore only this device-level WS strategy is used in all following experiments. Throughout the remainder of this experimental section, the value of T, is set to $T = 2^{15} = 16,384$.

Calibration of threshold-value for WS trigger: For the PTE variant a threshold for the WS trigger must be defined. In the previous paragraph we used $T/10$ as an ad-hoc value for this threshold, meaning that an intra-device work stealing phase occurs only if 10% of the IVMs have empty intervals. Figure 4 shows the variation of the node processing speed according to the value of the static WS trigger. For thresholds from 1 to $12,000$ the achieved performance for problem instances *crane16* (Figure 4a) and *crane17* (Figure 4b) is shown. The number of explored subproblems is 1.1×10^9 and 6.9×10^9 respectively.

One can see that the performance depends strongly on the calibration of the triggering mechanism. The use of this mechanism is clearly beneficial compared to a no trigger algorithm. Indeed, a WS-trigger value equal to 0 corresponds to a trigger-less version of the algorithm, where a WS phase in launched at each iteration. However, careful calibration is necessary, as a too large value deteriorates the algorithm's performance. For both test cases and a single GPU a factor 10 can be gained by using the triggering mechanism and setting is to a suitable value.

The results shown in Figure 4a are averaged values over 5 runs and shows error bars. The results shown in Figure 4b correspond to a single execution. In both figures one can notice that the results obtained for 3 and especially 4 GPUs is noisy.

Profiling of the algorithm revealed that the variation in execution time corresponds mainly to a variation in waiting time. This is due to the use of the random strategy in combination with the trigger mechanism. The random victim selection on the host level naturally introduces some randomness in the obtained results. However, this effect is amplified by the use of the trigger-mechanism. While a GPU is in the midst of a node expansion phase it is unreacheable until the end of this phase. Nevertheless, this GPU can still be selected as a WS victim, and



(a) Crane16 (b) Crane17

Figure 4: Node processing rates (in 10^6 nodes/s) in function of values for the work stealing trigger. Using T=16,384 IVMs, $1-4$ GPUs and random inter-GPU work stealing strategy.

the thief thread needs to wait until the selected victim returns to the CPU. To overcome this issue it would be necessary to send a signal to the victim device while it is exploration kernel is running. However, to the best of our knowledge, CUDA currently provides no guarantee for data coherence between a running kernel and concurrent data transfers.

These results demonstrate, on the one hand, that the triggering mechanism can be used to accelerate the exploration process. On the other hand they show that it requires careful tuning, especially in a multi-GPU setup. In all the following experiments with the PTE variant the WS-trigger threshold is set to 20%, i.e., 3277 for T=16384 IVMs per GPU.

4.4 Evaluation of hybrid multi-core/multi-GPU B&B

In this subsection we evaluate the scaling of the multi-B&B algorithm using up to 4 GPUs. The achieved performance is compared to sequential and multi-core CPU implementations for the three test problems. We also report and discuss the experimental results obtained for the hybrid multi-core/multi-GPU algorithm.

Scalability of multi-GPU B&B: In this paragraph we evaluate the speedup of our multi-GPU B&B algorithm with respect to its single-GPU counterpart and compare its performance to CPU versions of the algorithm. The results obtained when using also the two available multi-core CPUs are discussed in the following paragraph.

In Table 1 we report the achieved node processing rates for the ten instances *Ta021–Ta030* (in kn/s). In order to better show the impact of the instance-size, the instances are sorted in an increasing order according to the number of nodes explored (shown in the seconds column). The results exclude the time spend in device initialization (cudaMalloc, cudaSetDevice, cudaEnablePeerAccess calls, etc.) which amounts to about 0.5 seconds for one GPU and (surprisingly) lasts up to 3 seconds when done in parallel by four POSIX threads. All results are obtained using the *Hypercube* and *Random* strategies and 16,384 IVMs per GPU. Table 1 also shows the processing speed achieved by the multi-core version of the IVM-based B&B algorithm using 32 threads on two 8-core E5-2650v3 CPUs. This multi-core version is $18\times-20\times$ faster than its sequential counterpart.

For example, when solving FSP instance *Ta030*, the multi-core algorithm decomposes 47.6 kn/s, the GPU-B&B decomposes 556 kn/s using a single GPU and 1737 kn/s using 4 GPUs, which is 3.1 times more than using a single device. All results for the GPU algorithm are averaged over three runs. For the ten flowshop instances *Ta021-Ta030* the 4-GPU IVM-based B&B using allows to decompose on average 2,178 kn/s which is 47 times more than its multi-core counterpart, 4 times more than the single-GPU algorithm and about 800 times more than a sequential B&B algorithm. For all FSP instances, except the smallest (which lasts less than 1 second using 4 GPUs), the algorithm achieves near-linear speedup on up to 4 GPUs. For this PTE-PEB variant of the algorithm each launch of the bounding kernel is configured with a different block-size, according to the current load. In some cases a single additional block may slow down the execution which is beyond our control. Such effects may explain the fact that we observe, in some cases, a speed-up slightly greater than 4.0 when comparing the 4-GPU and single-GPU execution times. Solving FSP instance *Ta022* using 4 GPUs, the algorithm spends 87.6% of the 9.9 seconds execution time

Table 1: Node processing rates (in kn/s) obtained when solving **FSP** instances *Ta021–Ta030*, using one to four GPUs, WS-strategy: *Hypercube/Random*. [1)]For comparison the node processing speed obtained with a 32-thread multi-core B&B (2×E5-2650v3, icc 15.0, random-1/2 WS) is shown.

Inst	#nodes $\times 10^6$	CPU[1)] kn/s	GPUx1 kn/s	GPUx2 kn/s	GPUx2 x1GPU	GPUx3 kn/s	GPUx3 x1GPU	GPUx4 kn/s	GPUx4 x1GPU
30	1.6	47.6	556	1042	1.9	1479	2.7	1739	3.1
29	6.8	48.8	556	1099	2.0	1643	3.0	2145	3.9
28	8.1	48.1	533	1032	1.9	1555	2.9	2027	3.8
22	22.1	50.5	532	1066	2.0	1597	3.0	2206	4.0
24	40.1	53.6	594	1174	2.0	1748	2.9	2420	4.1
21	41.4	47.4	523	1011	1.9	1532	2.9	2117	4.0
25	41.4	48.4	489	958	2.0	1442	3.0	1975	4.0
27	57.1	51.0	610	1211	2.0	1825	3.0	2423	4.0
26	71.3	55.4	578	1142	2.0	1711	3.0	2314	4.0
23	140.8	46.3	536	1029	1.9	1534	2.9	2178	4.1
AVG	**43.1**	**49.7**	**551**	**1077**	**2.0**	**1607**	**2.9**	**2155**	**3.9**

Table 2: Node processing rates (in Mn/s) obtained when solving **n-queens** (n= 15 − 19) and **ATSP** instances (tsmat15-19, crane15-19), using one to four GPUs, WS-strategy: *Hypercube/Random*.

Inst	#nodes $\times 10^6$	CPU[1)] Mn/s	GPUx1 Mn/s	GPUx2 Mn/s	GPUx2 x1GPU	GPUx3 Mn/s	GPUx3 x1GPU	GPUx4 Mn/s	GPUx4 x1GPU
15-queens	91	63.4[1)]	232	364	1.6	397	1.7	437	1.9
16-queens	563	86.4[1)]	286	497	1.7	633	2.2	710	2.5
17-queens	4,224	95.1[1)]	290	555	1.9	748	2.6	952	3.3
18-queens	29,350	89.8[1)]	281	567	2.0	779	2.8	999	3.6
19-queens	242,419	n/a	274	522	1.9	770	2.8	1040	3.8
AVG	**55,329**	**83.7**[1)]	**273**	**501**	**1.8**	**665**	**2.4**	**827**	**3.0**
crane15	218	23.1[2)]	350	549	1.6	700	2.0	726	2.1
crane16	1,089	26.6[2)]	362	625	1.7	929	2.6	1137	3.1
crane17	6,957	27.3[2)]	364	718	2.0	1015	2.8	1322	3.6
crane18	37,932	26.6[2)]	312	597	1.9	868	2.8	1132	3.6
crane19	245,308	n/a	301	572	1.9	849	2.8	1102	3.7
tsmat14	90	27.6[2)]	271	417	1.5	461	1.7	472	1.7
tsmat15	6,608	32.9[2)]	422	823	2.0	1172	2.8	1507	3.6
tsmat16	4,019	27.7[2)]	372	698	1.9	973	2.6	1183	3.2
tsmat17	241,801	n/a	388	764	2.0	1126	2.9	1457	3.8
AVG	**54,407**	**27.4**[2)]	**339**	**609**	**1.8**	**845**	**2.4**	**1043**	**3.0**

[1)] sequential bitset-based backtracking algorithm (Jeff Somers [15], on E5-2630v3, gcc 15.0).
[2)] our multi-core IVM-based B&B using 16 threads (on 2×E5-2630v3, gcc 15.0).

in the bounding phase, 1.8% in intra-device WS, 0.7% of the time waiting for work on the inter-device level, and about 10% in the selection, reduction, remapping and pruning kernels.

In Table 2 reports the achieved node processing rates (in Mn/s) for the n-queens ($n = 15$–19) problem counting the total number of solutions, and for solving ATSP instances *crane15-19*, *tsmat14-17*. As shown in the second column of Table 2 the size of explored tree grows exponentially according to the instance size n. Results for $n < 15$ are not shown as these instances are solved within fractions of a second. For $n > 19$ and *tsmat18* the exectution time exceeds the imposed time limit of 30 minutes using a single GPU. In order to compare the performance of our GPU algorithm to CPU-based performances, Table 2 also shows the processing rates obtained by Jeff Somers' highly optimized sequential bitset-based n-queens algorithm [15]. This algorithm is the fastest sequential algorithm for n-queens we were able to find in the literature. For $n = 14 - 18$ this algorithm decomposes on average 55.3 Mn/s which is approximately 40 times faster than our sequential IVM-based version for n-queens. For the ATSP problem the results are shown for the IVM-based multi-core algorithm using 16 threads.

For large n-queens instances ($n = 18, 19$) our 4-GPU-B&B algorithm is capable of decomposing up to $1,000$ Mn/s, finding all solutions to the 19-queens problem within 4 minutes. Profiling shows that the WS approach generates small overhead: solving ATSP instance *tsmat15*, 4.8% of the execution time (4.3 seconds) is spend in the work stealing phase. For larger instances the portion of time spend in WS is lower, as explorers run our of work less frequently.

Evaluation of hybrid multi-core/multi-GPU B&B: In this paragraph the efficiency of the hybrid multi-core/multi-GPU algorithm is evaluated. As shown by the experimental results reported in the previous paragraph, a single GPU can be used to process about 500 kn/s when solving FSP instances of the group *Ta021–Ta030*. This rate is approximately 200 times higher than the 2.5 kn/s processed by a sequential B&B algorithm. Hence there is little potential for accelerating the 4-GPU algorithm by using the 16 available CPU-cores in addition to the GPUs.

Table 3: Averaged execution times for FSP instance *Ta028* (100 executions) for comparing the multi-GPU only and the hybrid algorithm (using 16 CPU threads)

	GPUx1			GPUx4		
	t_{avg}	t_{min}	t_{max}	t_{avg}	t_{min}	t_{max}
no CPU (+0 CPU)	14.7±0.7%	14.6	14.9	3.9±0.6%	3.87	3.96
hybrid (+16 CPU)	16.1±22.1%	14.4	46.2	4.4±2.8%	3.84	5.09

In effect, such a hybrid algorithm could theoretically reach a node processing rate of about $4 \times 500\text{kn/s} + 16 \times 2.5\text{kn/s} = 2040\text{kn/s}$. This means for example that instance *Ta022* (22.1×10^6 nodes) could be solved within 10.8 seconds by the hybrid algorithm instead of 11.0 seconds when using the multi-GPU system alone: an improvement of approximately 2%. For comparison, on 50 executions of *Ta028* with 4 GPUs the measured relative standard error on the execution time is ±0.7%.

Therefore, in terms of performance, the programming effort required does not seem worth such a limited improvement. However, in the perspective of integrating the GPU-based B&B-algorithm in a larger hybrid cluster-system, we believe that study of this smaller hybrid setup may provide useful insights. The obtained experimental results show that the proposed approach for the hybrid multi-core/multi-GPU algorithm raises granularity issues.

For host-to-device WS operations, we have tested both granularity policies described in Subsection 3.5. Besides the steal-half policy, we tested the policy where a WS victim keeps only the $1/200^{th}$ part of its interval.

For both granularity policies our experiments show that the hybrid algorithm is on average slightly slower than its multi-GPU-only counterpart. More importantly, the results show that the hybrid algorithm is less stable than the multi-GPU algorithm.

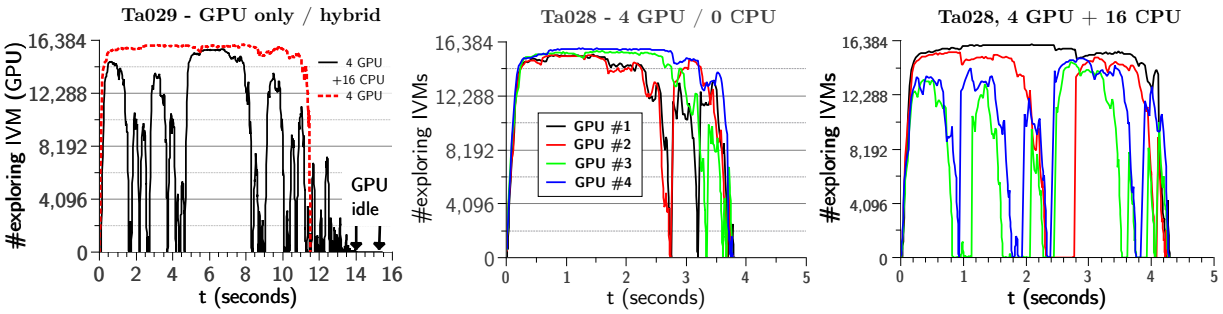
For example, performing 40 resolutions of 17-queens using the 4-GPU algorithm an average execution time of 4.55sec±2.87% (relative standard deviation) is obtained. For 40 resolutions of the same instance with the hybrid algorithm (4GPU/16CPU) the obtained average execution time is 4.67sec±6.10%, meaning that the hybrid algorithm is less stable than the B&B using multi-GPU only. This instability is amplified when only 1 GPU is used in combination with 16 CPU-explorers. In Table 3 the average elapsed time for 100 resolutions of FSP instance *Ta028* is shown for the different combinations of 1/4 GPUs and 0/16 CPUs. The obtained average is shown with the relative standard deviation (100%×standard-deviation/mean) and the observed minimum and maximum values. While the obtained minimum values are lower for the hybrid algorithm, there is an average slowdown and an increased instability. For a single GPU combined with 16 CPUs we observe two extreme outlier values, one execution lasting 46.0 seconds, one 27.9 seconds.

These results can be explained with the help of Figure 5, showing the evolution of the workload in each GPU (in terms of active IVMs) during explorations of FSP instances *Ta022* and *Ta028*. As one can see on the left-hand side (Figure 5a) slowdown occurs mainly because of host-to-device WS operations. After stealing an interval from a CPU-explorer the GPU spends a few iterations below its maximal power before the stolen interval is completely divided among the T IVM structures. Thus, in order to use the GPUs efficiently, they must be allocated large chunks of work. Another effect that slows down the hybrid exploration process is a much slower shut-down phase if a CPU thread is holding the final chunks of works. In this example the hybrid exploration process lasts about 4 seconds longer than the GPU-only exploration. Figure 5b shows the GPU workloads during a resolution of *Ta028* using 4 GPUs alone, for comparison with a resolution using 16 additional CPU threads (Figure 5c). The comparison shows that, in the presence of CPU-explorers the GPUs run out of work more frequently, because they give away large chunks of work in the early stage of the exploration.

We attempt to deal with this issue by cutting the work units into different chunk sizes. However, it is impossible to know in advance how much work each interval actually contains. The results presented in this subsection show that the approach which consists in treating CPU and GPU workers symmetrically is questionable because it threatens the stability of the algorithm. While the presented multi-GPU approach shows good results in a multi-GPU-only system, a worker-farmer approach (as in [34]) should be able to better handle the heterogeneity in a B&B algorithm targeting large heterogenous clusters.

5 Conclusion and future work

In this paper, we have presented a GPU-based B&B algorithm for multi-core systems equipped with multiple GPUs. Our algorithm uses Integer-Vector-Matrix (IVM) data structures instead of a conventional linked-lists of nodes. Based on intervals of factoradics, we have proposed a new hierarchical work stealing (WS) scheme which balances work loads on two levels: inside and GPUs and among GPUs and CPU cores.



(a) *Ta029* using 1 GPU and 1 GPU+16 CPUs (b) *Ta028* using 4 GPUs (c) *Ta028* using 4 GPUs+16 CPUs

Figure 5: Number of active IVMs during a resolution of FSP instances.

Inside each GPU, workers synchronously exchange work units (intervals) during periodical WS phases. These WS phases are followed by node expansion phases which form the computational core of our algorithm. We have presented two variants of the algorithm. A first one which targets problems with costly bounding function: it uses a second level of parallelism where subproblems are evaluated in parallel and WS phases are launched after each node expansion phase. The second variant targets fine-grained search trees: the node expansion phase includes a mechanism which triggers work stealing phases if the number of idle workers reaches a certain threshold.

Both variants use the same WS strategies. For load balancing inside the device we have proposed a hypercube-based victim selection strategy which we have compared experimentally to a self-adjusting ring-based strategy presented in our previous work [11]. For load balancing between different GPUs and CPU cores an asynchronous random WS strategy is used. The workers belonging to the same GPU behave like a single worker on the inter-GPU level, stealing work collectively from workers in other devices once all of them run out of work.

Our approach has been tested extensively. The results obtained for a set of 10 flowshop instances show that our multi-GPU B&B algorithm using 4 GPUs (1) provides an average 3.9× speedup over the single-GPU algorithm and (2) is capable of processing 43 times as many nodes as an optimized multi-core B&B algorithm using 2 8-core CPUs. Also, the experimental results for the n-queens problem and the travelling salesman problem show that the proposed load balancing scheme using the hypercube topology and trigger mechanism is capable of handling coarse and fine-grained work loads.

The investigation of the hybrid approach using 16 CPU-cores in addition to 4 GPUs indicate some challenging granularity issues regarding the future integration of our multi-GPU B&B algorithm into a cluster-based B&B using other many-core accelerators, like Intel Xeon Phi. In the hybrid approach presented in this paper CPU cores and GPUs are directly exchanging work units between each other, although the latter are shown to be capable of sustaining 200 times higher node processing rates. Compared to the multi-GPU-only approach this leads to a slight average slowdown and diminishes the algorithm's stability. Therefore, our future work will investigate whether a farmer-worker approach allows a better integration of the IVM-based GPU-B&B algorithm into heterogeneous platforms. Furthermore we plan to enable to validate our algorithm to handle different types of permutation problems by automatically choosing the suitable variant of the proposed algorithm.

References

[1] Jenkins J, Arkatkar I, Owens JD, Choudhary A, Samatova NF. Lessons learned from exploring the backtracking paradigm on the gpu. *Proceedings of the 17th International Conference on Parallel Processing - Volume Part II*, Euro-Par'11, Springer-Verlag: Berlin, Heidelberg, 2011; 425–437. URL `http://dl.acm.org/citation.cfm?id=2033408.2033458`.

[2] Bader DA, Hart WE, Phillips CA. Parallel algorithm design for branch and bound. *Tutorials on Emerging Methodologies and Applications in Operations Research: Presented at INFORMS 2004, Denver, CO* 2005; **76**.

[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* 2013; **25**(8):1121–1136, doi: 10.1002/cpe.2931.

[4] Vu TT, Derbel B, Melab N. Adaptive Dynamic Load Balancing in Heterogenous Multiple GPUs-CPUs Distributed Setting: Case Study of B&B Tree Search. *7th International Learning and Intelligent OptimizatioN Conference (LION)*, Lecture Notes in Computer Science: Catania, Italy, 2013. URL `https://hal.inria.fr/hal-00765199`.

[5] Lalami M, El-Baz D. GPU Implementation of the Branch and Bound Method for Knapsack Problems. *IEEE 26th Intl. Parallel and Distributed Processing Symp. Workshops PhD Forum (IPDPSW)*, Shanghai, CHN, 2012; 1769–1777, doi:10.1109/IPDPSW.2012.219.

[6] Carneiro T, Muritiba A, Negreiros M, Lima de Campos G. A New Parallel Schema for Branch-and-Bound Algorithms Using GPGPU. *23rd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2011; 41–47, doi:10.1109/SBAC-PAD.2011.20.

[7] Plauth M, Feinbube F, Schlegel F, Polze A. Using dynamic parallelism for fine-grained, irregular workloads: a case study of the n-queens problem. *2015 Third International Symposium on Computing and Networking (CANDAR)*, IEEE, 2015; 404–407.

[8] Zhang T, Shu W, Wu MY. Optimization of n-queens solvers on graphics processors. *Proceedings of the 9th International Conference on Advanced Parallel Processing Technologies*, APPT'11, Springer-Verlag: Berlin, Heidelberg, 2011; 142–156. URL `http://dl.acm.org/citation.cfm?id=2042522.2042533`.

[9] Mezmaz M, Leroy R, Melab N, Tuyttens D. A Multi-Core Parallel Branch-and-Bound Algorithm Using Factorial Number System. *28th IEEE Intl. Parallel & Distributed Processing Symp. (IPDPS)*, Phoenix, AZ, 2014; 1203–1212, doi:10.1109/IPDPS.2014.124.

[10] Gmys J, Mezmaz M, Melab N, Tuyttens D. A GPU-based Branch-and-Bound algorithm using Integer–Vector–Matrix data structure. *Parallel Computing* 2016; :–doi:http://dx.doi.org/10.1016/j.parco.2016.01.008. URL `http://www.sciencedirect.com/science/article/pii/S0167819116000387`.

[11] Gmys J, Mezmaz M, Melab N, Tuyttens D. IVM-based Work Stealing for Parallel Branch-and-Bound on GPU. *11th Intl. Conf. on Parallel Processing and Applied Mathematics*, Krakov, Poland, 2015. URL `https://hal.inria.fr/hal-01248329`.

[12] Blumofe RD, Leiserson CE. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)* 1999; **46**(5):720–748.

[13] Karypis G, Kumar V. Unstructured tree search on simd parallel computers: A summary of results. *Proceedings of the 1992 ACM/IEEE Conference on Supercomputing*, Supercomputing '92, IEEE Computer Society Press: Los Alamitos, CA, USA, 1992; 453–462. URL `http://dl.acm.org/citation.cfm?id=147877.148046`.

[14] Lauterback C, Mo Q, Manocha D. Work distribution methods on gpus. *Technical Report TR009-16*, University of North Carolina 2009. URL `gamma.cs.unc.edu/GPUCOL/GPU-Workqueues.pdf`.

[15] Somers J. The N-Queens problem: A study in optimization. URL `http://jsomers.com/nqueen_demo/nqueens.html`, Accessed: 2016-07-17.

[16] Melab N. Contributions à la résolution de problèmes d'optimisation combinatoire sur grilles de calcul. LIFL, USTL November 2005. Thesis HDR.

[17] Cung V, Dowaji S, Cun BL, Mautor T, Roucairol C. Parallel and distributed branch-and-bound/A* algorithms. *Technical Report 94/31*, Laboratoire PRISM, Université de Versailles 1994. URL `http://citeseer.ist.psu.edu/cung94parallel.html`.

[18] Gendron B, Crainic T. Parallel Branch and Bound Algorithms: Survey and Synthesis. *Operations Research* 1994; **42**:1042–1066.

[19] 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* 2014; **26**(16):2667–2683, doi:10.1002/cpe.3155.

[20] Meyer X, Chopard B, Albuquerque P. A branch-and-bound algorithm using multiple gpu-based lp solvers. *20th Annual International Conference on High Performance Computing*, IEEE, 2013; 129–138.

[21] Adel D, Bendjoudi A, El-Baz D, Abdelhakim AZ, *et al.*. Gpu-based two level parallel b&b for the blocking job shop scheduling problem. ; URL `http://dl.cerist.dz/handle/CERIST/794`.

[22] Foley T, Sugerman J. Kd-tree acceleration structures for a gpu raytracer. *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, ACM, 2005; 15–22.

[23] Rocki K, Suda R. Parallel minimax tree searching on gpu. *International Conference on Parallel Processing and Applied Mathematics*, Springer, 2009; 449–456.

[24] Carneiro T, Nobre RH, Negreiros M, de Campos GAL. Depth-first search versus Jurema search on GPU branch-and-bound algorithms: A case study. *NVIDIA's GCDF - GPU Computing Developer Forum on XXXII Congresso da Sociedade Brasileira de Computação (CSBC)* 2012; .

[25] Feinbube F, Rabe B, von Löwis M, Polze A. Nqueens on cuda: Optimization issues. *2010 Ninth International Symposium on Parallel and Distributed Computing*, IEEE, 2010; 63–70.

[26] Knuth DE. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc.: Boston, MA, USA, 1997.

[27] Leroy R. Parallel branch-and-bound revisited for solving permutation combinatorial optimization problems on multi-core processors and coprocessors. PhD Thesis, Université Lille 1 2015.

[28] Harris M, Sengupta S, Owens JD. Parallel prefix sum (scan) with CUDA. *GPU Gems* 2007; **3**(39):851–876.

[29] Cantor G. Ueber die einfachen Zahlensysteme. *Zeitschrift für Mathematik und Physik* 1869; **14**:121–128.

[30] Gmys J, Leroy R, Mezmaz M, Melab N, Tuyttens D. Work stealing with private integer–vector–matrix data structure for multi-core branch-and-bound algorithms. *Concurrency and Computation: Practice and Experience* 2016; :n/a–n/adoi:10.1002/cpe.3771. URL `http://dx.doi.org/10.1002/cpe.3771`, cpe.3771.

[31] Garey MR, Johnson DS, Sethi R. The Complexity of Flowshop and Jobshop Scheduling. *Mathematics of Operations Research* 1976; **1**(2):pp. 117–129.

[32] Lageweg BJ, Lenstra JK, Kan AHGR. A General Bounding Scheme for the Permutation Flow-Shop Problem. *Operations Research* 1978; **26**(1):53–67, doi:10.1287/opre.26.1.53.

[33] Taillard E. Benchmarks for basic scheduling problems. *Journal of Operational Research* 1993; **64**:278–285.

[34] Mezmaz M, Melab N, Talbi EG. A grid-enabled branch and bound algorithm for solving challenging combinatorial optimization problems. *21th IEEE Intl. Parallel and Distributed Processing Symp. (IPDPS)*, Long Beach, CA, 2007; 1–9.

[35] Cirasella J, Johnson DS, McGeoch LA, Zhang W. The asymmetric traveling salesman problem: Algorithms, instance generators, and tests. *Revised Papers from the Third International Workshop on Algorithm Engineering and Experimentation*, ALENEX '01, Springer-Verlag: London, UK, UK, 2001; 32–59. URL `http://dl.acm.org/citation.cfm?id=646679.702307`.