Many-core Branch-and-Bound for GPU accelerators and MIC coprocessors

Nouredine Melab, Jan Gmys, Mohand Mezmaz, Daniel Tuyttens

Many-core Branch-and-Bound for GPU accelerators and MIC coprocessors

N. Melab, J. Gmys, M. Mezmaz and D. Tuyttens

Abstract Coprocessors are increasingly becoming key building blocks of High Performance Computing platforms. These many-core energy-efficient devices boost the performance of traditional processors. On the other hand, Branch-and-Bound (B&B) algorithms are tree-based exact methods for solving to optimality combinatorial optimization problems (COPs). Solving large COPs results in the generation of a very large pool of subproblems and the evaluation of their associated lower bounds. Generating and evaluating those subproblems on coprocessors raises several issues including processor-coprocessor data transfer optimization, vectorization, thread divergence, and so on. In this paper, we investigate the offload-based parallel design and implementation of B&B algorithms for coprocessors addressing these issues. Two major many-core architectures are considered and compared: Nvidia GPU and Intel MIC. The proposed approaches have been experimented using the Flow-Shop scheduling problem and two hardware configurations equivalent in terms of energy consumption: Nvidia Tesla K40 and Intel Xeon Phi 5110P. The reported results show that the GPU-accelerated approach outperforms the MIC offload-based one even in its vectorized version. Moreover, vectorization improves the efficiency of the MIC offload-based approach with a factor of two.

N. Melab
Inria Lille - Nord Europe,
CNRS/CRIStAL, Université Lille 1, France
e-mail: Nouredine.Melab@univ-lille1.fr

J. Gmys, M. Mezmaz and D. Tuyttens
Mathematics and Operational Research Department (MathRO),
University of Mons, Belgium
e-mail: \{Jan.Gmys,Mohand.Mezmaz,Daniel.Tuyttens\}@umons.ac.be

1 Introduction

Coprocessors are increasingly becoming key building blocks of High Performance Computing platforms. In addition to their energy efficiency, they boost the performance of traditional processors through the combination of a larger number of processing cores, vector-SIMD processing and multi-threading. Currently, the most used coprocessors (Top500 ranking [1] of June 2017) are Nvidia GPU accelerators and Intel MIC coprocessors. The former are composed of a large number of slim cores while the latter integrate a relatively smaller number of streamlined largish cores relying on SIMD processing. Today, coprocessors allow to achieve peak performance of the order of several TeraFlops. Nevertheless, it is often difficult for the programmers to extract a large portion of the theroretically available performance. Indeed, the specific features of these coprocessors raise several issues including the optimization of data transfer between the processor and its coprocessor, vectorization, data placement optimization, etc. More details on the these hardware features and related challenging issues are given in the next sections.

In this paper, the focus is put on the Branch-and-Bound (B&B) algorithm [12]. Recently, the parallelization of B&B has been revisited for multi-core (clusters of) processors [3] and GPU [14, 11, 6] and their combination [5], [19]. In this paper, we investigate the parallelization of Branch-and-Bound (B&B) algorithms for coprocessors (GPU and MIC[1]). B&B algorithms are well-known methods for solving to optimality NP-hard optimization problems[2]. They are based on an implicit enumeration of the solutions composing the search space associated with the problem to be tackled. The search space is explored by dynamically building a tree whose root node designates the original problem. Each internal or intermediate node represents a subproblem obtained by the decomposition of the subproblem associated with its parent. The leaf nodes designate potential solutions or subproblems that cannot be decomposed. The construction of the B&B tree and its exploration are performed using four operators: *branching*, *bounding*, *selection* and *elimination*. The algorithm proceeds in several iterations to progressively improve the best solution found so far. The generated and not yet examined subproblems are kept in a pool, that is initialized with the original problem. At each iteration, the selection operator is used to select a subproblem from this pool, according to some strategy (depth-first, best-first,...). The *branching operator* performs its decomposition into other subproblems. The *bounding operator* calculates a lower bound of each generated subproblem. Each subproblem having a lower bound higher than the upper bound is discarded by the *elimination operator*. It means that this subproblem will not be decomposed.

[1] GPU and MIC stand for respectively Graphics Processing Unit and Many Integrated Cores.

[2] An optimization problem consists of minimizing or maximizing a cost function. Without loss of generality, in this paper the minimization case and the permutation Flow-Shop scheduling problem are considered.

The bounding operator is the most time consuming part of a B&B algorithm. Indeed, in [6], it is shown that this operator represents on average between 98 % and 99 % of B&B applied to the Flow-Shop problem. Such result demonstrates that the bounding operator needs massively parallel computing. The GPU-accelerated bounding has been investigated in [13]. The reported results show that the CPU-GPU data transfer is costly. Therefore, it is recommended to perform also the branching operator on GPU in order to generate locally the subproblems and evaluate their lower bounds. However, this raises some issues related to the highly irregular (in shape and size) nature of B&B tree: thread divergence, thread mapping, etc. Regarding the parallelization of B&B on MIC coprocessors, it has been addressed in our work proposed in [15] using the native mode. In this mode, the coprocessor is standalone and executes the whole B&B algorithm. However, to the best of our knowledge, the parallelization of B&B on MIC coprocessors has not been addressed using the offload (GPU-like) mode.

The objective of this paper is to investigate the parallel bounding model combined with the parallel tree exploration model of B&B algorithms to allow highly efficient solving of large instances of the Flow-Shop problem on GPU accelerators and Intel MIC Xeon Phi coprocessors. In Section 2, we first present the general design of the coprocessor-accelerated B&B. In Section 3 (respectively Section 4), we describe the implementation of the GPU-accelerated (respectively Phi-accelerated) approach. In Section 5, we report some experimental results comparing the two coprocessor-based many-core implementations. Finally, some conclusions are drawn in Section 6.

2 Coprocessor-accelerated B&B: the general design

In this section, the general coprocessor-accelerated parallel approach is first presented. Then, the Flow-Shop permutation scheduling problem is presented as it is used as a use case to validate the approach.

2.1 The parallel model

As mentioned above, the coprocessors are many-core devices dedicated to massively parallel computing. Therefore, to take maximum advantage of the computational power provided by these coprocessors these latter should be fed by a large number of computations. In our proposed parallel coprocessor-based approach, as illustrated in Figure 1, several B&B trees are explored in order to generate multiple pools maximizing the use of the coprocessor cores. During the exploration of each pool, except the selection and pruning operators which are performed on the processor the branching and bounding operators are executed on the coprocessor. As

shown in Figure 1, on the processor side, at each iteration of the exploration process a set of tree nodes (whose size is a user-parameter) is selected. The selected set of nodes is offloaded to the coprocessor to be processed.

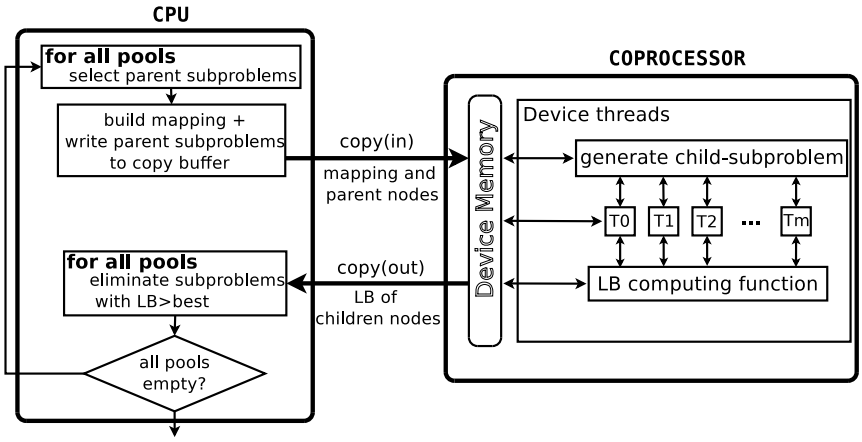

Fig. 1: Coprocessor-based B&B

On the coprocessor side, as illustrated in Algorithm 10, each parent node is processed by a thread, which starts by getting its identifier *thdId* (line 2). Using that identifier and a mapping strategy, the thread determines the parent node to branch generating a subproblem child (line 3). If the generated subproblem is a leaf (solution) this latter is evaluated (line 5), otherwise its associated lower bound (LB) is computed (line 7) and inserted in a global pool of nodes with their associated bounds (line 9). This pool is returned back to the CPU host. Every child having a lower bound greater than the cost of the best solution found so far is pruned on CPU. All the non-pruned children are inserted into the pools. The process is iterated until the exploration is completed and the optimal solution is found.

3 GPU-based implementation of B&B

In this section, we first present the parallelization model on GPU. To do that we recall the hardware view of GPU, its parallel programming model and its associated algorithmic challenging issues. Then, we show how these issues are dealt with in the implementation of the GPU-accelerated B&B.

Algorithm 1 Kernel of the computation of the lower bounds on the coprocessor.
input: parent-subproblems, mapping
output: lower bounds of children subproblems

```
1:  kernel EVALUATEONCOPROCESSOR
2:      thdId←getThreadId()
3:      child-subproblem←generateChild(thdId, mapping, parent-subproblems)
4:      if isLeaf(child-subproblem) then
5:          LB←evaluateSolution(child-subproblem)
6:      else
7:          LB←computeBound(child-subproblem)
8:      end if
9:      poolOfBounds[thdId]←LB
10: end kernel
```

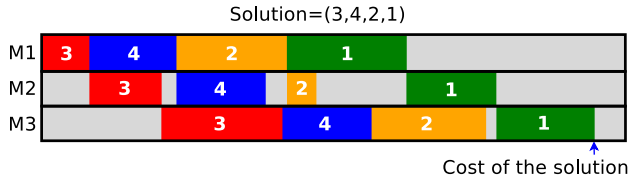

Fig. 2: Illustration of the Flow-Shop problem.

3.1 The Flow-Shop scheduling problem (FSP)

The Flow-Shop scheduling problem is a permutation problem which consists in scheduling n jobs on m machines [8]. For instance, in Figure 2, 4 jobs designated by different colors are scheduled on 3 machines.

The scheduling must be done with respect to two constraints: each machine cannot be simultaneously assigned to more than one job (to more than one color), and the execution order of the jobs (the colors) is the same on all the machines. The objective is to minimize the makespan, i.e. the termination date of the last job on the last machine. In this example, the solution consists in scheduling first the red job, then the blue job and orange one, and finally the green job. The solution can be coded as a permutation $(3,4,2,1)$ and its cost is the termination date of the green job on the machine M3. In this work, the lower bound function proposed by Lageweg *et al.* [10] is used in our bounding operator (Algorithm 2). This lower bound is mainly based on Johnson's theorem [9] which provides a procedure for finding an optimal solution for Flow-Shop scheduling problem with 2 machines. This bound is known for its good results and has a computational complexity of $O(m^2 nlog(n))$, where n is the number of jobs and m the number of machines. For large values of the parameters m and n, the problem is time-intensive. More details on the lower bound and its computational complexity can be found in [13].

3.2 Parallelization on GPU

For a long time, GPU computing has been used to speed up image and video processing. Since 2006, with the introduction by Nvidia of its Cuda software toolkit the use of GPUs has been extended to numerous other application domains including combinatorial optimization. The popularity of Cuda is due to its simplicity as it is an extension of the C language with data parallel features. The principle is easy: the programmer writes a code for one thread (kernel) and can instantiate it on a large number of threads to allow massive parallel computing on GPU. In addition, Cuda is portable between successive generations allowing transparent scalability of Cuda applications.

Before the Cuda parallel model is presented, let us recall the hardware architecture of a GPU device. As shown in Figure 3, a GPU is a coprocessor, coupled to a CPU through a PCI Express bus. In the Cuda vocabulary, the processor is called "host" and the GPU is called "device". The GPU is composed by a set of streaming multi-processors (processors) including each a pool of 32-bit or 64-bit SIMD processors (processing cores). For instance, a Kepler GPU device contains 15 processors of 192 Cuda cores for a total of 2880 Cuda cores [17]. A GPU is also composed of several memories including global and local off-chip memories, and a shared memory, registers and a cache memory. These memories have different characteristics in terms of size and access latency. For instance, the global memory is big and has a long latency while registers are small and fast memories.

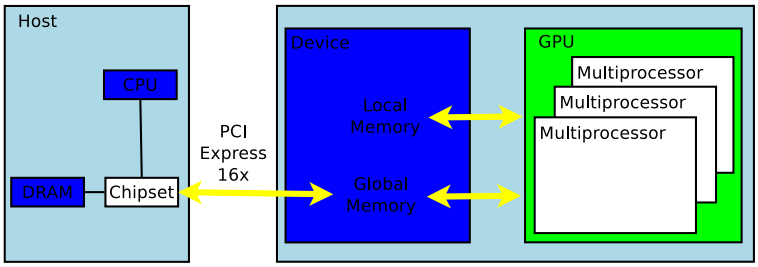

Fig. 3: Hardware view: GPU = coprocessor of CPU.

From software programming point of view, as illustrated in Figure 4 a GPU Cuda-based parallel program is composed of two parts: a "host" part and a "device" part. The host part is a serial or weakly parallel code because the number of CPU cores is small compared to the number of GPU cores. The device part is massively parallel because a GPU contains from hundreds to thousands of processing cores. During the execution of a parallel program the host part offloads streams of threads to the GPU device to be executed according to a two-level parallelism: at the higher level the processors (or SMX) execute the thread kernel according to the Sin-

gle Program Multiple Data (or SPMD) model. At the lower level (intra-SMX), the threads are executed according to the Single Instruction Multiple Data (or SIMD) or Single Instruction Multiple Thread (or SIMT) model. Indeed, inside each processor the instruction flow composing a thread kernel is executed according to the SIMD model.

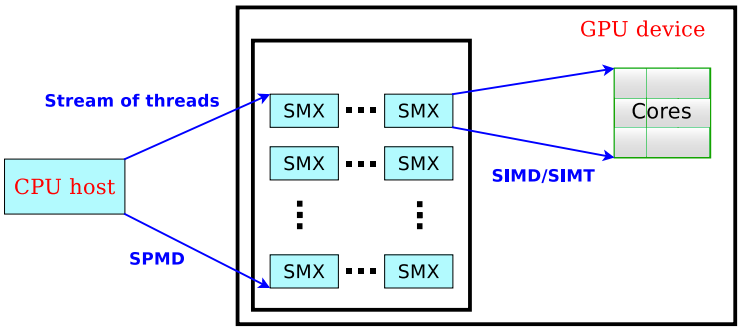

Fig. 4: Software view: Parallel program = weakly parallel/serial host code + massively parallel device code.

The threads offloaded from CPU host to GPU device are organized by the programmer in a hierarchical way into grids of blocks of threads. Grids are arrays $1D$ or $2D$ of blocks and blocks are arrays $1D$, $2D$ or $3D$ of threads. The thread organization corresponds to the organization of application data which are often vectors, matrices or volumes. As shown in Figure 5, the blocks are assigned to the SMXs by the Cuda runtime. Inside each SMX each block is split into warps i.e. pools of 32 threads. Warps are scheduling units i.e. the threads are executed by pools of 32. This allows to overlap the memory access latency by computation. Context switching is very fast as each thread has its own registers.

To sum up, from algorithmic and software programming point of view at least three issues should be addressed: (1) the optimization of the data transfer between CPU and GPU; (2) the optimization of the data placement on the hierarchy of memories of the GPU having different sizes and latencies; and (3) thread or branch divergence management especially for irregular applications.

3.3 Parallelization of B&B for GPU

The implementation on GPU of the coprocessor-based B&B according to the general design presented in Section 2 requires to address the challenging issues quoted

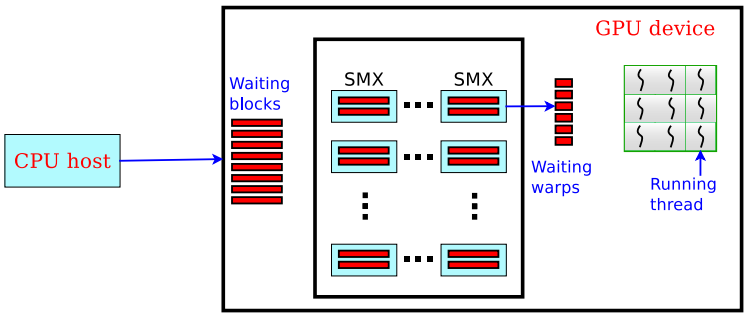

Fig. 5: Software view: Parallel program = grid(s) of block(s) of threads executed as warps of 32 threads.

above. First, to deal with the CPU-GPU data transfer optimization the branching operator, which generates tree nodes or subproblems, is moved to GPU. The execution of the branching operator on the GPU device allows one to avoid the transfer of the branched parent nodes from CPU to GPU which is costly. However, this raises other issues related to thread granularity and mapping. Indeed, if a parent node is processed entirely (branching and bounding) by a single thread there will be a load imbalance leading to thread/branch divergence. In fact, the parent nodes may have different numbers of children as they are located at different levels in the B&B tree. To deal with this problem the processing of each thread is limited to a single node, meaning that each thread generates and evaluates only one child of the parent node.

Second, to tackle the problem related to data placement optimization on GPU we have followed the recommendation proposed in [13]. Indeed, as illustrated in Algorithm 2, the implementation of the lower bound algorithm includes 6 data structures: the matrix PTM of the processing times of the jobs, the matrix of lags LM, the Johnson's matrix JM, the matrix RM of the earliest starting times of jobs, the matrix QM of their lowest latency times and the matrix MM containing the couples of machines. The algorithm needs as input a subproblem defined as a permutation with some jobs already scheduled at its beginning and/or its end.

The semantics of these data structures is not the focus of this paper. For more details on these ones and on the lower bound please refer to [13]. The focus is rather put here on the optimization of the placement of these data structures on the different memories of the GPU device. Due to the limited size of the shared memory, the matrices do not fit in all together, especially for large problem instances. Based on the complexity analysis of these data structures and an experimental study conducted in [13], it is suggested to put in the shared memory the Johnson's and/or processing time matrices (JM and PTM). The other data structures are mapped either to the global memory or to the constant memory. Such data placement allows one to achieve accelerations of more than $\times 100$ compared to a single-CPU core

Algorithm 2 Computation of lower bound (un-vectorized)
input: subproblem = {permutation, nbFixed (#jobs fixed)} constant data (MM, JM, PTM, LM)
output: lower bound (LB) of subproblem

```
1:  n := #jobs
2:  function COMPUTEBOUND
3:      RM, QM, SM ← InitTabs(permutation, nbFixed)
4:      LB ← 0
5:      for (k = 0 → N(N−1)/2) do
6:          tmp0, tmp1, ma0, ma1 ← InitFun(k, nbFixed, MM, RM)
7:          for (j = 0 → n) do                                   ▷ ∼ 70% of time
8:              job←JM[k][j]
9:              if (SM[job]==0) then
10:                 tmp0 += PTM[ma0][job]
11:                 tmp1 = max(tmp1, tmp0 + LM[k][job]) + PTM[ma1][job]
12:             end if
13:         end for
14:         tmp1←EndFun(tmp0, tmp1, k, nbFixed, QM)
15:         LB = max(tmp1, LB)
16:     end for
17:     return LB
18: end function
```

serial execution of B&B. In our implementation, MM, JM, PTM and LM are put on the constant memory. A part of PTM is then moved to the shared memory. The other matrices are stored on the global memory.

Third, the parallelization of irregular applications such as B&B applied to permutation problems due to the thread or branch divergence issue [6]. In our implementation, the irregularity is due to two factors: as the tree nodes have different levels they require different amounts of work (number of children) ; on the other hand, the lower bound function, as it can be seen in Algorithm 2, includes several conditional instructions and loops. To deal with the first factor, each thread handles only one child as mentioned above. Therefore, all the threads perform the same amount of work. To tackle the second problem, we have reused the refactoring approach proposed in [6] even if the achieved performance improvement is not significant as the size of the factorized branches is small.

Finally, the mechanisms used on GPU are not experimented individually here because their efficiency has been demonstrated in [6, 13, 4]. These three citations can be used for further details on the mechanisms. However, the performance of the whole GPU-accelerated B&B is evaluated and compared to the performance of the Xeon Phi-based B&B in Section 5.

4 MIC-based implementation of B&B

In this section, we first present the parallelization model on MIC architecture. To do that we recall the hardware view of Intel Xeon Phi, its parallel programming model and its associated algorithmic challenging issues. Then, we show how these issues are dealt with in the implementation of the Intel Xeon Phi-accelerated B&B.

4.1 Parallelization on Intel Xeon Phi

The market of accelerators has been dominated by Nvidia during several years. Since recently, they are faced to the competition of Intel with its Many Integrated Cores (MIC)-based Xeon Phi. This latter is a coprocessor coupled to the processor through a PCI Express bus [7]. As illustrated in Figure 6, a typical platform consists of one to two Intel Xeon processor(s) (CPUs) and one to eight (two in this figure) Intel Xeon Phi coprocessors per host. Several of such platforms may be interconnected to form a cluster or supercomputer.

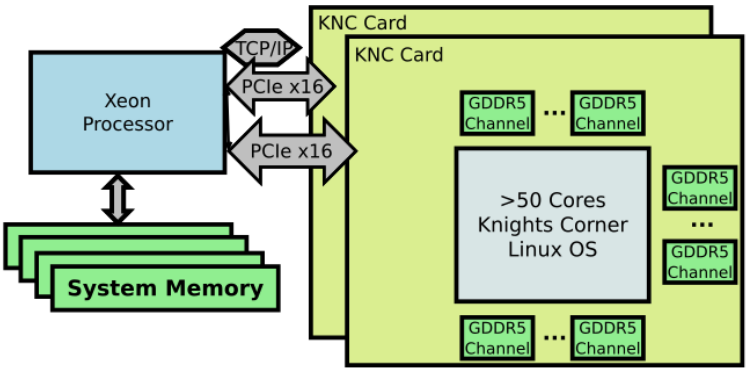

Fig. 6: Hardware view: Intel Xeon Phi = coprocessor of CPU.

From the hardware point of view, the Xeon Phi board has one Knights Corner (KNC) processor, the first production chip based on the MIC architecture, and 8 GB of GDDR5 RAM. As illustrated in Figure 7, KNC integrates up to 60 CPU-cores interconnected by a high-speed bi-directional ring, and runs at over 1 GHz. It connects to its private external memory with a peak bandwidth of over 320 Gbps. The cores are based on the Intel Pentium architecture. Each core has 32 KB of L1 data and instruction cache, 512 KB of L2 data cache, and a 512-bit vector Floating Point Unit (FPU). This latter performs fused-multiply-add (FMA) operations. Therefore, the peak performance is about 32 (resp. 16) GFlops in single (resp. double) precision.

Consequently, the KNC delivers a peak performance of about 2 (resp. 1) TFlops in single (resp. double) precision.

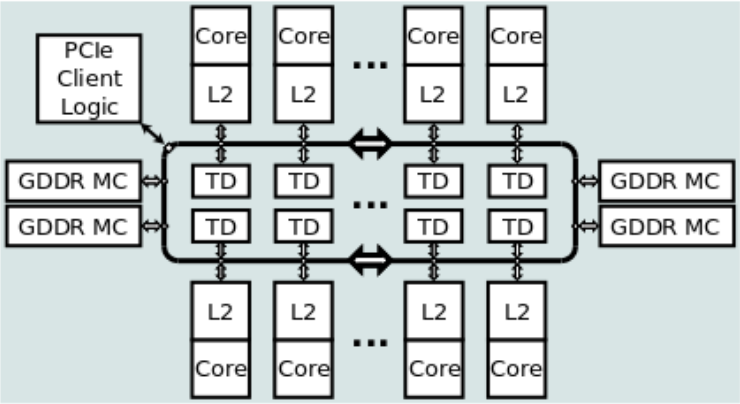

Fig. 7: Hardware view: Knights Corner core.

From a programming standpoint, the key is to treat the Intel Xeon Phi coprocessor as an x86-based SMP-on-a-chip with over 50 cores, with multiple threads per core and 512-bit SIMD instructions. From programming language point of view, Intel Xeon Phi is more accessible than Nvidia GPU because it can be programmed using standard programming environments such as OpenMP, MPI, Cilk Plus and Posix Threads. However, to achieve higher performance one should consider two fundamental features: scaling through locality and Simultaneous Multi-Threading (SMT) and vectorization. On the other hand, as an Intel Xeon Phi coprocessor runs an operating system (Linux) and has its own IP address, there are two ways to involve it in a parallel program:

- A processor-centric "offload" mode where the program is viewed, like for GPUs, as running on processors and offloading work to coprocessors. Another issue which arises with the offload mode is, like for GPU, the optimization of the data transfer between the processor and the coprocessor.
- A "native" mode where the program runs natively on only coprocessors or on coprocessors and processors together. In the latter case, the two devices may communicate with each other by various methods. At least two other issues arise in this case: the optimization of the data/task partitioning and the communication between the processor and the coprocessor. The challenge for data/task partitioning is the load balancing between the "big cores" of the processor and the "little" cores of the coprocessor. Regarding the communication, the challenge is to overlap the communications by the computation, to manage data locality, etc.

4.2 Offload-based parallelization of B&B for Intel Xeon Phi

In this paper, we investigate the offload mode for the parallelization of B&B algorithms on Intel Xeon Phi coprocessors. In the proposed approach, the processor-coprocessor data transfer optimization approach is the same as for GPU. Indeed, the branching operator is performed on Phi, this allows one to reduce the cost of data transfer between the two devices. On the other hand, all the data structures which are not modified between offloading operations are offloaded once.

One of the major mechanisms allowing performance improvement on Intel Xeon Phi is vectorization. Different levels are provided ranging from compiler-based automatic easy-to-use vectorization to manual and programmer control vectorization. In our work, as quoted previously, the most consuming part of the B&B algorithm is computation of the lower bound function (Algorithm 2). In this paper, we propose a vectorization method of the lower bound function focusing on its most compute-intensive portion and the main data-dependencies. This portion of code is the inner for-loop (line 7-13) which consumes about 70% of the bounding time. The body of this inner loop is executed $\frac{J^2\times(J-1)}{2}$ times. Regarding data dependencies, the statement in line 11, including a dependency of current $tmp1$ on $tmp1$ from previous iteration prevents vectorization (*icc* does not auto-vectorize it). In addition, except for line 15, the iterations of the outer loop are independent (private variables: tmp0, tmp1, ind0, ind1, current). However, only the inner loop of a loop nest may be vectorized [3].

In order to allow more vectorization than provided by the compiler, the order of the nested loops must be inverted as illustrated in the vectorized lower bound function (Algorithm 3). For auto-vectorization by the compiler it is preferable to write small separate loops, rather than merging into a single loop. The outer loop is thus split into 3 separate serial loops and a max-reduce operation (line 21) in order to isolate the k-dependent instructions from the inner-loop. The cost to pay for this is to declare the scalars $tmp0$, $tmp1$, $ma0$, $ma1$ as arrays (resp. $Tmp0$, $Tmp1$, $Ma0$, $Ma1$) of size $\frac{J^2\times(J-1)}{2}$. The same strategy on GPU severely breaks down performance due to the memory problem (these intermediate variables are no longer stored into registers). Even with the highest optimization level activated ($-O3$) the Intel compiler (*icc*) still needs the hint "#pragma ivdep to vectorize the inner loop (line 10) successfully. The two other for-loop are auto-vectorized.

5 Experimentation

In this section, we present an experimental study of the proposed many-core approaches using GPU and Intel Xeon Phi and compare them. We first present the

[3] http://d3f8ykwhia686p.cloudfront.net/1live/intel/CompilerAutovectorizationGuide.eps

Algorithm 3 Computation of lower bound (vectorized)
input: subproblem = {permutation, nbFixed (#jobs fixed)}, constant data (MM, JM, PTM, LM)
output: lower bound (LB) of subproblem

```
1: n := #jobs
2: function COMPUTE LB VECTORIZED
3:     RM, QM, SM ← InitTabs(permutation, nbFixed)
4:     LB ← 0
5:     for (k = 0 → n(n−1)/2) do
6:         Tmp0[k], Tmp1[k], Ma0[k], Ma1[k] ← InitFun(k, nbFixed, MM, RM)
7:     end for
8:     for (j = 0 → J) do                                   ▷ permute loop-order
9:         #pragma ivdep
10:        for (k = 0 → n(n−1)/2) do                        ▷ inner loop vectorizable
11:            job←JM[j][k]                                 ▷ transpose JM
12:            if (SM[job]==0) then
13:                Tmp0[k] += PTM[Ma0[k]][job]
14:                Tmp1[k] ← max(Tmp1[k], Tmp0[k] + LM[k][job]) + PTM[Ma1[k]][job]
15:            end if
16:        end for
17:    end for
18:    for (k = 0 → n(n−1)/2) do
19:        Tmp1[k]←EndFun(Tmp0[k], Tmp1[k], k, nbFixed, QM)
20:    end for
21:    LB←max-reduce(Tmp1[])
22:    return LB
23: end function
```

hardware and software testbeds and some parameter setting used for our experiments. Then we report and discuss some experimental results.

5.1 Hardware and software testbed and parameter setting

As shown in Table 1, all the experiments are run on a computer from Grid'5000 [2] which is a single socket Sandy Bridge machine (Intel Xeon E5-2650, 64 GB RAM), equipped with one MIC coprocessor Intel Xeon Phi 5110P (60 physical cores at 1.053 GHz, 320 GB/s of memory bandwidth). The operating system is a Centos 6 (OS officially supported by Intel for the Xeon Phi) and the compiler is Intel *icc* for Intel devices. The GPU device is an NVIDIA Tesla K40 based on the Kepler GK110B architecture. It includes 2880 Cuda cores at 0.745 GHz, 288 GB/s of memory bandwidth. For the GPU-accelerated implementation, the NVIDIA CUDA Toolkit release 5.0.35 is used together with the gcc version 4.4.7. For all experiments the compilation level 3 (-O3) is used. On the other hand, the UNIX `time` command is used to measure the elapsed execution time for each Flow-Shop instance. The GFLOPS (SP) row is obtained using the following computations:

- K40: $2(\frac{flops}{cycle}) \times 15(SM) \times 192(\frac{cores}{SM}) \times 0.745(\frac{GHz}{core}) = 4291\ GFlops.$

- MIC: $32(\frac{flops}{cycle}) \times 60(cores) \times 1.053(\frac{GHz}{core}) = 2022\ GFlops.$

Finally, the last row (Thermal Design Power, TDP) indicates that the hardware configurations of the two compared coprocessors (Tesla K40 and Xeon Phi 5110P) are equivalent in terms of energy consumption.

	NVIDIA Tesla K40	Intel Xeon-Phi 5110P
#Physical cores	15	60
#Logical cores	2 880	240
clock(Ghz)	0.745	1.053
GFLOPS(SP)	4 291	2 022
GFLOPS(DP)	1 430	1 011
SIMD	N/A	512-bit
L2 Cache(MB)	1.5	30
Mem BW(GB/s)	288	320
TDP (Watt)	235	225
Launch date	Oct'13	Oct'12

Table 1: Hardware execution platform

As quoted in Section 2, the coprocessors are many-core devices dedicated to massively parallel computing. Therefore, they need to be fed by a large number of computations. To do that, many B&B trees are explored in order to generate multiple pools maximizing the use of the coprocessor cores. Before the experiments are performed, the number M of pools to be created is calibrated through a series of experiments on the problem instance $Ta028$. The experimental results are reported in Figure 8, showing the execution time as an average over 10 independent runs. Error bars show the corresponding standard deviation. Based on the figure, the number of pools is fixed to 800 (resp. 2200) for the GPU-accelerated (resp. Xeon Phi-based) approach.

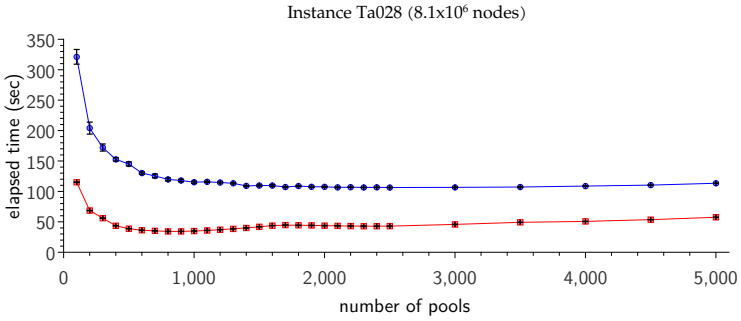

Fig. 8: Elapsed time vs. Number of pools.

5.2 Experimental results

The performance analysis of the contributions of this paper we have compared the two many-core approaches: the GPU-accelerated approach and the MIC-accelerated approach. In order to get a fair comparison between them the offload mode is considered for the MIC-based approach. In our experiments, simultaneous multi-threading (SMT with 2 threads per core) is considered for the MIC-based approach. The two approaches have been experimented using the 10 instances ($Ta021 - Ta030$) of the Taillard's problem 20 jobs on 20 machines. The best solution found so far is initialized to the optimal solution to guarantee that the amount of work (explored nodes) is the same for each of the experimented approaches. Therefore, the objective of the resolution is to prove the optimality of the initial solution. Obviously, to provide the optimal solution the same algorithm is used. One has just to initialize the best solution found so far to infinity for a minimization problem. The obtained experimental results are reported in Table 2.

		CPU-seq	GPU	Xeon-Phi	
Inst.	nodes ($\times 10^6$)			Vect	No-Vect
21	41.4	21 752	174	571	1 220
22	22.1	10 425	90	287	617
23	140.8	70 386	585	1 856	4 116
24	40.1	17 635	153	457	1 021
25	41.4	22 446	177	599	1 203
26	71.4	30 706	273	829	1 787
27	57.1	25 073	211	597	1 489
28	8.1	4 043	34	109	239
29	6.8	3 262	28	87	193
30	1.6	795	7	24	50
Avg	**43.1**	**20 653**	**173**	**542**	**1 193**

Table 2: Exploration time (in seconds) for solving Flow-Shop instances *Ta021-Ta030* initialized at optimal solution

The first two columns of the table contain respectively the numbers of the 10 solved problem instances and their associated search space sizes in millions of nodes. The following columns designate the exploration time in seconds obtained using respectively the GPU-accelerated approach, and the vectorized and non-vectorized MIC-based approaches using 236 threads.

In [18], the authors report some experimental results demonstrating the impact on data transfer overhead of the Coprocessor Offload Infrastructure (COI) daemon in the offload mode. The COI daemon runs the services required to support data transfer for offload on a dedicated core. The reported results show that it is beneficial to avoid using this core for user code, i.e., one should use only 59 cores. Following this recommendation, we have used 236 threads (corresponding to 59 cores) in our

experiments for the offload-based MIC approach. Indeed, according to our experimental results, using all 60 cores (240 threads) for computations incurs performance penalties up to 80%.

From the last row of the table, two major observations can be made. First, the GPU-accelerated approach clearly outperforms the MIC-based approach even in its vectorized version. Second, vectorization allows one to speed up the MIC-based approach with a factor of two.

6 Conclusion

According to the recent Top500 ranking (July 2015), it is confirmed that hybrid HPC platforms including coprocessors is the trend towards the exascale era. On the other hand, it appears that the market of hybrid HPC is dominated by Nvidia followed by Intel with its Xeon Phi. In this paper, we have revisited the parallelization of B&B algorithms for many-core coprocessors, in particular Nvidia GPU and Intel Xeon Phi. From the design point of view, we have combined two hierarchical parallel models: the parallel tree exploration model and the parallel bounding. The bounding operator is performed on the coprocessor because, on the one hand, it is the most time-consuming part of the B&B algorithm. On the other hand, it is massively data parallel and thus well-suited for coprocessors. In addition, the branching operator, which generates tree nodes during the exploration process, is also performed on the coprocessor to minimize the cost of their offloading from the processor to the coprocessor.

Such coprocessor-based design of B&B algorithms gives rise to other issues: thread mapping, thread/branch divergence and data placement optimization on GPU, and vectorization on Intel Xeon Phi. From the GPU side, we have reused some recommendations proposed in [4]. For the MIC-based approach, we have proposed a vectorization method for the lower bound function. The many-core implementations have been experimented on the 10 instances of the 20 jobs-on-20 machines problem using equivalent hardware configurations in terms of energy consumption. The reported results show that the GPU-accelerated approach outperforms the MIC offload-based one even in its vectorized version. Moreover, vectorization improves the efficiency of the MIC offload-based approach with a factor of two.

In the near future, we plan to extend this work to a cluster-level parallelization combining multi-core GPU-accelerated and MIC-based computing. From an application point of view, the objective is to solve to optimality challenging difficult and unsolved Flow-Shop instances as we did for one 50×20 problem instance with grid computing [16]. Finally, we plan to investigate other lower bound functions to deal with other combinatorial optimization problems.

Acknowledgement

Experiments presented in this paper were carried out using the Intel Xeon Phi of Digitalis (http://digitalis.imag.fr) and Grid'5000 testbed (https://www.grid5000.fr). Therefore, we would like to thank the Digitalis staff and especially Pierre Neyron from LIG/CNRS Lab for making the MIC processor fully functional and available. Grid'5000 is supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations.

References

1. Top500 HPC international ranking, http://www.top500.org/. URL http://www.top500.org/
2. et al., R.B.: Grid'5000: A large scale and highly reconfigurable experimental grid testbed. IJHPCA **20**(4), 481–494 (2006)
3. Barreto, L., Bauer., M.: Parallel branch and bound algorithm-a comparison between serial, openmp and mpi implementations. In Journal of Physics: Conference Series **256** (2010)
4. Chakroun, I.: Parallel heterogeneous branch and bound algorithms for multi-core and multi-gpu environments. PhD Thesis from Université Lille 1 (2013)
5. Chakroun, I., Melab, N., Mezmaz, M., Tuyttens, D.: Combining multi-core and gpu computing for solving combinatorial optimization problems. Journal of Parallel Distributed Computing **73**(12), 1563–1577 (2013). URL http://dx.doi.org/10.1016/j.jpdc.2013.07.023
6. 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)
7. Chrysos, G.: Intel Xeon Phi X100 Family Coprocessor - the Architecture. Intel, Inc., https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner (2012)
8. Garey, M., Johnson, D., Sethi, R.: The complexity of flow-shop and job-shop scheduling. Mathematics of Operations Research **1**, 117–129 (1976)
9. Johnson, S.: Optimal two and three-stage production schedules with setup times included. Naval Research Logistis Quarterly **1**, 61–68 (1954)
10. Lageweg, B., Lenstra, J., Kan, A.R.: A General bounding scheme for the permutation flow-shop problem. Operations Research **26**(1), 53–67 (1978)
11. Lalami, M.: Contribution à la résolution de problèms d'optimisation combinatoire: méthodes séquentielles et parallèles. Université de Toulouse III - Paul Sabatier. (2012)
12. Lawler, E.L., Wood, D.E.: Branch-and-bound methods: A survey. Operations Research **14**(4), 699–719 (1966). DOI 10.1287/opre.14.4.699
13. 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)
14. Melab, N., Chakroun, I., Mezmaz, M., Tuyttens, D.: A gpu-accelerated branch-and-bound algorithm for the flow-shop scheduling problem. In: 2012 IEEE Intl. Conf. on Cluster Computing, CLUSTER 2012, Beijing, China, September 24-28, 2012, pp. 10–17 (2012)
15. Melab, N., Leroy, R., Mezmaz, M., Tuyttens, D.: Parallel branch-and-bound using private ivm-based work stealing on xeon phi MIC coprocessor. In: 2015 Intl. Conf. on High Performance Computing & Simulation, HPCS 2015, Amsterdam, Netherlands, July 20-24, 2015, pp. 394–399 (2015)

16. Mezmaz, M., Melab, N., Talbi., E.G.: A grid-enabled branch and bound algorithm for solving challenging combinatorial optimization problems. In: In Proc. of 21th IEEE Intl. Parallel and Distributed Processing Symp. (IPDPS). Long Beach, California (2007)
17. Nvidia: NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110/210. Whitepaper (2014)
18. Saini, S., Jin, H., Jespersen, D., Cheung, S., Djomehri, M., Chang, J., Hood, R.: Early multi-node performance evaluation of a knights corner (KNC) based NASA supercomputer. In: 2015 IEEE Intl. Parallel and Distributed Processing Symposium Workshop, IPDPS 2015, Hyderabad, India, May 25-29, 2015, pp. 57–67 (2015)
19. Vu, T.T.: Heterogeneity and locality-aware work stealing for large scale branch-and-bound irregular algorithms. PhD Thesis from Université Lille 1 (2014)