Multi-core versus Many-core Computing for Many-task Branch-and-Bound applied to Big Optimization Problems

Nouredine Melab, Jan Gmys, Mohand Mezmaz, Daniel Tuyttens

Multi-core *versus* Many-core Computing for Many-task Branch-and-Bound applied to Big Optimization Problems

N. Melab[a], J. Gmys[a,b], M. Mezmaz[b], D. Tuyttens[b]

[a] *Inria Lille - Nord Europe,*
CNRS/CRIStAL, Université Lille 1, France
Email: {Nouredine.Melab
[b] *Mathematics and Operational Research Department (MathRO),*
University of Mons, Belgium
Email: {Jan.Gmys,Mohand.Mezmaz,Daniel.Tuyttens}@umons.ac.be

Abstract

On the road to exascale, coprocessors are increasingly becoming key building blocks of High Performance Computing platforms. In addition to their energy efficiency, these many-core devices boost the performance of multi-core processors. In this paper, we revisit the design and implementation of Branch-and-Bound (B&B) algorithms for multi-core processors and Intel Xeon Phi coprocessors considering the offload mode as well as the native one. In addition, two major parallel models are considered: the master-worker and the work pool models. We address several parallel computing issues including processor-coprocessor data transfer optimization and vectorization. The proposed approaches have been experimented using the Flow-Shop scheduling problem (FSP) and two hardware configurations equivalent in terms of energy consumption: Intel Xeon E5-2670 processor and Intel Xeon Phi 5110P coprocessor. The reported results show that: (1) the proposed vectorization mechanism reduces the execution time by 55.4% (resp. 30.1%) in the many-core (resp. multi-core) approach ; (2) the offload mode allows a faster execution on MIC than the native mode for most FSP problem instances ; (3) the many-core approach (offload or native) is in average twice faster than the multi-core approach ; (4) the work pool parallel model is more suited for many/multi-core B&B applied to FSP than the master-worker model because of its irregular nature.

Keywords: Muti-core, Coprocessor/Many-core, Intel Xeon Phi, Parallel Branch-and-Bound, permutation Flow-Shop.

1. Introduction

In addition to multi-core processors, coprocessors are increasingly becoming key building blocks of High Performance Computing platforms. Besides their energy efficiency, they boost the performance of traditional multi-core processors through the combination of a larger number of processing cores, vector-SIMD processing and multi-threading. Currently, the most used coprocessors (Top500 ranking of November 2015) [1] are Nvidia GPU accelerators and Intel MIC[1] coprocessors. The former are composed of a large number of slim cores while the latter, the

[1]MIC stands for Many Integrated Cores

focus of this paper, integrate a relatively smaller number of streamlined largish cores relying on SIMD processing. From now on, for brevity "MIC" is used to designate Intel Xeon Phi and "MC" to represent multi-core. Today, MIC coprocessors allow to achieve peak performance of the order of one TeraFlops. Nevertheless, it is often difficult to extract the largest portion of the theoretically 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. Several research works have recently addressed these issues in different application areas including supervised deep learning [21], DNA sequence analysis [17], astrophysics (dynamics of astrophysical objects) [10], hydrodynamics (smoothed particle hydrodynamics simulation of a nebula) [7], etc. However, in combinatorial optimization the parallelization for Xeon Phi coprocessors is rarely addressed. The overall objective of this paper is to revisit the parallel design and implementation of Branch-and-Bound (B&B) algorithms for MIC coprocessors. For comparison (MC *versus MIC*), a multi-core implementation is also considered using the same parallelization approach.

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 node. 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 smaller subproblems. The *bounding operator* calculates a lower bound of each generated subproblem. Each subproblem having a lower bound higher than the current upper bound (best solution found so far) is discarded by the *elimination operator*. It means that this subproblem will not be decomposed.

Recently, the parallelization of B&B algorithms has been revisited for multi-core (clusters of) processors [2] and GPU [15, 12, 4] and their combination [3, 22]. In [4], it is shown that the bounding operator represents on average between 98 % and 99 % of B&B applied to the Flow-Shop problem (considered as a case study in this paper). Such result demonstrates that the bounding operator needs massively parallel computing. The GPU-accelerated bounding has been investigated in [16]. 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. In this paper, we reuse this idea within the context of MIC coprocessors using the offload (GPU-like) mode. Regarding the parallelization of B&B on MIC coprocessors, it has only been previously addressed in our work [18] using the native mode without vectorization. The parallelization of the algorithm is based on the work stealing model. In the native mode, the coprocessor is standalone and executes the whole B&B

[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.

algorithm. To the best of our knowledge, the parallelization of B&B on MIC coprocessors has not been previously addressed using the offload mode. The major contributions of this paper are the following:

- Revisiting the parallelization of B&B and its implementation on MC processors and MIC coprocessors. The objective 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 MC processors and MIC coprocessors. Two parallel models are considered: master-worker and work-pool [11]. Unlike in the master-worker model, in the work pool model the work distribution is initiated by the workers (not the master) which makes this model more suited for irregular applications such as B&B.
- For the parallelization on MIC, investigating the offload-based approach as well as the native-oriented one.
- Proposing a vectorization mechanism for multi-core processors as well as for MIC coprocessors. Vectorization has not been considered in our previous work [18].
- Finally, evaluating and discussing the performance of the proposed approaches considering different criteria: multi-core *vs.* many-core, offload *vs.* native, vectorized *vs.* non-vectorized, and master-worker *vs.* work pool. The experiments have been performed using the Flow-Shop scheduling problem and two hardware configurations equivalent in terms of energy consumption: Intel Xeon E5-2670 and Intel Xeon Phi 5110P.

The rest of the paper is organized as the following. In Section 2, we first present the IVM-based serial B&B algorithm. IVM is a new data structure dedicated to permutation problems and presented in the next section. In Section 3, we detail the multi-core and many-core parallelization approaches. In Section 4, we report some experimental results comparing the different approaches. Finally, some conclusions are drawn in Section 5.

2. IVM-based serial B&B

As quoted in the introduction, B&B is a tree-based exploration algorithm using four operators: selection, bounding, elimination and branching. At implementation, these four operators act on the data structure that stores the generated subproblems. Therefore, the design of the data structure has a great impact on the efficiency of these operators. A stack implemented as a linked-list is often used in existing works related to B&B algorithms using depth-first selection. In [18], it is shown that the IVM data structure, which stands for Integer-Vector-Matrix, allows to speed up the exploration using a smaller memory footprint than its linked-list counterpart. Therefore, IVM is reused in this paper to deal with permutation optimization problems. A permutation optimization problem consists in finding a permutation of n objects which optimizes an objective function. For instance, for the Flow-Shop problem, considered as a case study in this paper, it consists in finding the schedule of n jobs on m machines that minimizes the makespan. In other words, the objective is to find a permutation of n jobs that minimizes the due date of the last job on the last machine. The IVM data structure for the Flow-Shop problem is illustrated in Figure 1 where the left side (Figure 1.(a)) is the tree-based representation using top-down gray rows and the Linked-list-based representation using horizontal and vertical solid lines. In the

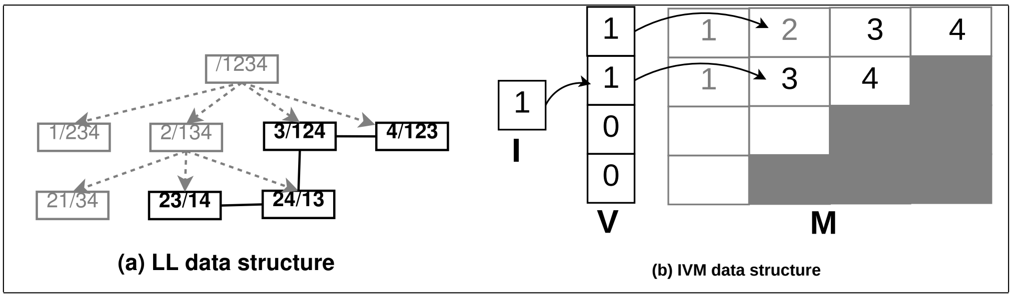

Figure 1: LL and IVM-based representations

tree-based representation, each node designates a partial (subproblem) permutation or a full (solution) permutation. The jobs before the "/" symbol are scheduled while the following ones are to be scheduled.

On the other hand, IVM illustrated in Figure 1.(b) indicates the next subproblem to be solved. The integer I of IVM gives the level (position of "/" + 1) of this subproblem. In this example, the level of the next subproblem is 1. The vector V indicates the jobs that are already scheduled (jobs 2 and 3 in the example) and thus the position of the subproblem among its sibling nodes in the tree. The matrix M, which is triangular, contains the jobs to be scheduled at each level: all the n jobs (for a problem with n jobs) for the first row, $n - 1$ jobs for the second row, and so on. To use the IVM data structure in the B&B algorithm, some of its operators are revisited as the following:

- The bounding operator just associates a lower bound to a subproblem, so it is not changed. The three other operators act on the data structure, so they have to be redefined.
- The selection operator indicates the next subproblem (23/14 in the example) to be solved using the integer and the vector.
- The bounding operator associates a lower bound to the selected subproblem. If the lower bound is smaller than the cost of the best solution found so far, the subproblem is decomposed.
- The branching (decomposition) operator is performed by modifying the IVM data structure. The modification of IVM consists in increasing the level, meaning the value of I, and copying in the matrix M the next jobs to be scheduled to the next row.
- The elimination operator withdraws a subproblem if its lower bound is greater than the cost of the best solution found so far. Its implementation consists in incrementing the index in V if possible. Otherwise, the value of the integer I is decremented and the value of V is incremented.

3. Many-core and Multi-core B&B

In this section, we first briefly describe the MIC-based hardware architecture and its associated parallel programming model. Then, we detail the offload-based and native-oriented

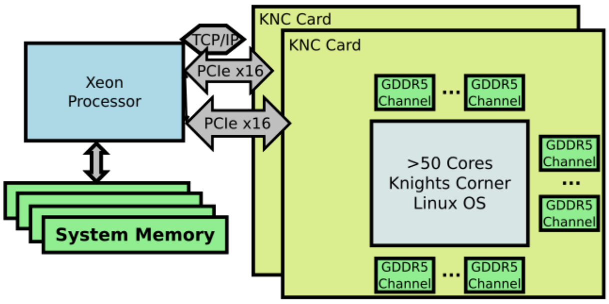

Figure 2: Hardware view: Intel Xeon Phi = coprocessor of CPU.

parallelization approaches for MIC coprocessors. The parallelization approach for multi-core processors is the same as the native-based one for MIC coprocessors.

3.1. MIC-based architecture and parallelization

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. As illustrated in Figure 2, 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. Multiple such platforms may be interconnected to form a cluster or supercomputer [6].

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 3, 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 kept coherent by a global-distributed tag directory (TD), 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.

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 three fundamental features: scaling through locality, Simultaneous Multi-Threading (SMT) and vectorization. These last two are often combined as it is done in this paper and other papers such as [17] and [21]. In [17], thread-level parallelism is used in splitting the DNA sequence into chunks, and vectorization is applied to the transition function of finite automata (loop vectorization). In [21], thread parallelism is used to divide the input over the available threads, allowing threads to process samples concurrently. Vectorization is applied in convolutional layers to the computations of partial derivatives and weight gradients. On the other hand, as an Intel Xeon Phi coprocessor runs an operating system (Linux) and has its own

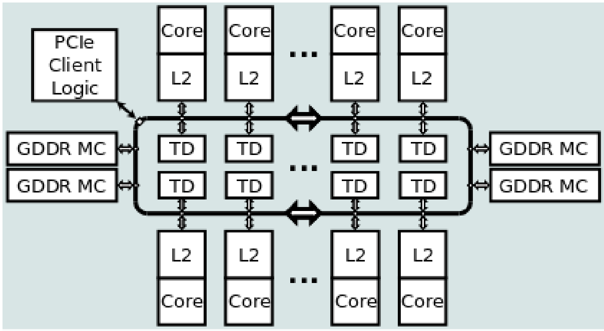

Figure 3: Hardware view: Knights Corner core.

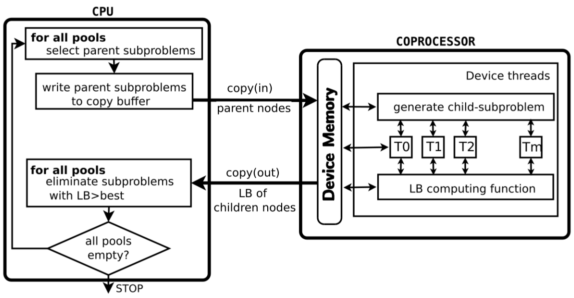

Figure 4: Offload-oriented many-core B&B.

IP address, there are two ways to involve it in a parallel program: a processor-centric "offload" mode and a "native" mode. In the next sections, we investigate the two approaches.

3.2. Offload-based parallelization of B&B for MIC coprocessors

The parallelization approach of the B&B algorithm uses the thread-based Master-Worker parallel model. The master thread performs the selection and pruning operators of the B&B while the worker threads execute the branching and bounding operators. To take maximum advantage of the computational power provided by the massively parallel MIC coprocessors these latter should be fed by a large number of computations (many-task computing). To achieve that as illustrated in Figure 4, the master thread maintains multiple IVM-based pools of tree nodes generated and evaluated by the worker threads. 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.

On the coprocessor side, as illustrated in Algorithm 1 (Line 10-23), each thread generates and evaluates one or more children subproblems. Each child subproblem is uniquely identified by the corresponding parent subproblem and an index indicating its position among sibling nodes (Line 13-15). After mapping a thread onto a parent node and a child index, the thread generates

Algorithm 1 Pseudo-code for the offload-based Branch-and-Bound algorithm.

```
1: procedure BRANCH-AND-BOUND
2:     N := problem-size
3:     while (true) do
4:         for (i : 0 → #IVM) do
5:             poolOfParentNodes[i]←selectNextSubproblem(i)
6:         end for
7:         if (no subproblemFound) then
8:             STOP
9:         end if
10:        #pragma offload(mic:0) in(poolOfParentNodes) out(poolOfLowerBounds)
11:        {
12:        for (i : 0 → (#IVM × N)) do            ▷ parallelize this loop using OpenMP directives
13:            ivm←i/N
14:            childId←i (mod N)
15:            child-subpb←generateChild(poolOfParentNodes[ivm], childId)
16:            if isLeaf(child-subpb) then
17:                LB←evaluateSolution(child-subpb)
18:            else
19:                LB←computeLB(child-subpb)
20:            end if
21:            poolOfLowerBounds[i]←LB
22:        end for
23:        }
24:        for (i : 0 → #IVM) do
25:            branchOnCPU(i)                      ▷ Can be overlapped with computation of bounds
26:        end for
27:        for (i : 0 → #IVM) do
28:            eliminateSubproblems(poolOfLowerBounds[i], i)
29:        end for
30:    end while
31: end procedure
```

the child subproblem. For children that are leaves the cost of the solution is evaluated. The children which are internal nodes are evaluated (bounded) using Algorithm 2 and their associated lower bounds are returned to the CPU. In order to avoid transferring the resulting children nodes the branching operator is also performed on the CPU (this operation can be overlapped with the computation of lower bounds using the `signal` and `wait` specifiers). Every child having a lower bound greater than the cost of the best solution found so far is pruned on CPU. The selection and pruning operators (Lines 5 and 26) are performed in parallel using all the processing cores of the CPU. The process is iterated until the exploration is completed and the optimal solution is found.

The implementation on Intel Xeon Phi of the coprocessor-based B&B requires to address the challenging issues quoted in previous section. First, to deal with the processor-coprocessor data transfer optimization the branching operator, which generates tree nodes or subproblems, is moved to the coprocessor. The execution of the branching operator on the coprocessor allows one to avoid the transfer of the branched parent nodes from CPU to Xeon Phi 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. 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 tasks assigned to threads are child nodes instead of parent nodes.

The semantics of the algorithm and its data structures is not the focus of this paper. For more details on these ones and on the lower bound please refer to the Ph.D thesis of I. Chakroun [5].

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

```
n := #jobs
procedure COMPUTE LB
    RM, QM, SM ← InitTabs(permutation, nbFixed)
    LB ← 0
    for (k = 0 → n(n−1)/2) do
        tmp0, tmp1, ma0, ma1 ← InitFun(k, nbFixed, MM, RM)
        for (j = 0 → n) do
            job←JM[k][j]
            if (SM[job]==0) then
                tmp0 += PTM[ma0][job]
                tmp1 = max(tmp1, tmp0 + LM[k][job]) + PTM[ma1][job]
            end if
        end for
        tmp1←EndFun(tmp0, tmp1, k, nbFixed, QM)
        LB = max(tmp1, LB)
    end for
    return LB
end procedure
```

3.2.1. Vectorization

One of the major mechanisms allowing performance improvement on Intel Xeon Phi is vectorization [20]. 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 the calculation of the lower bound function (Algorithm 2). In the rest of this section, we first present FSP then we propose a method for the vectorization of its associated lower bound function.

Flow-Shop 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 5, we have 4 jobs designated by different colors to be 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

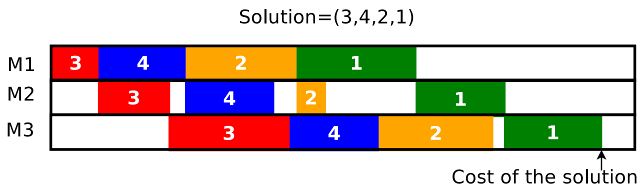

Figure 5: Illustration of the Flow-Shop problem.

job on the machine M3. In this work, the lower bound function proposed by Lageweg *et al.* [13] 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 [16].

Vectorization of the lower bound of FSP. The focus is put on the most compute-intensive portion of the lower bound function and its main data-dependencies. This portion of code is the inner for-loop (Lines 7-13) which consumes about 70% of the bounding time. The body of this inner loop is executed $n^2 \times (n-1)/2$ times, n being the number of jobs. Regarding data dependencies, the statement in Line 11, including a dependency of current $tmp1$ on $tmp1$ from previous iteration prevents vectorization (*icc* do not auto-vectorize it). In addition, except for Line 15 the iterations of the outer loop are independent (private variables: tmp0, tmp1, ind0, ind1 and current). However, only the inner loop 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).

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

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: procedure 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 procedure
```

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 20) in order to isolate the k-dependent instructions from the inner-loop. The cost to pay for this is to declare the scalars $tmp0$, $tmp1$, $ma0$ and $ma1$ as arrays (resp. $Tmp0$, $Tmp1$, $Ma0$ and $Ma1$) of size $\frac{n(n-1)}{2}$. The same strategy on GPU severely breaks down performance due to the memory problem (these intermediate variables are no longer stored into registers). In order to improve performance and assist the compiler in vectorizing the loops all arrays are aligned at 64 byte boundaries. For static arrays this is achieved by using `__attribute__((aligned(64)))`. For dynamically allocations the `_mm_malloc` function is used and the statement `__assume_aligned(arr, 64)` informs the compiler immediately before the concerned loop that the starting address of array `arr` is a multiple of 64 bytes. 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-loops are auto-vectorized.

3.3. Multi-core and native-based many-core parallelization of B&B

As pointed out in Section 3.1, in the native mode the Xeon Phi coprocessor is used as an independent many-core device. Therefore, the parallel B&B algorithm can be executed entirely on the coprocessor. For a fair comparison between MC and MIC, the same parallelization approach is used for the two architectures. We have considered the two parallelization models: master-worker and work pool. In the master-worker approach illustrated in Figure 6, as for the offload-based parallelization approach (for a fair comparison), the master thread maintains a set of IVM-based pools of subproblems and performs an iterative exploration process until the optimal solution is found. At each iteration, it selects a subset of parent subproblems from those pools to be sent to the worker threads which perform their decomposition and the evaluation of the generated children. These later are sent back to the master thread together with their associated lower bounds. The selection and pruning operators are performed in parallel using all the

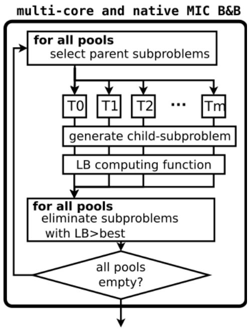

Figure 6: Native-oriented many-core or multi-core B&B.

processing cores of the CPU.
In the work pool-oriented parallelization approach, the master thread plays the same role. However, the parent subproblems are distributed to the worker threads on their request. As pointed out in the introduction this approach is well-suited in our case because the calculation function of the lower bounds is irregular. The master thread stops the exploration process when all the pools of subproblems are empty.

The implementations of the master-worker and work pool approaches using OpenMP are quite similar. The major difference between them is the parameter of the "schedule" clause in the work sharing parallel loop, which is "static" (resp. "dynamic") for the master-worker (resp. work pool) implementation. Finally, the two implementations are vectorized using the same method as for the offload-based many-core approach, presented in Section 3.2.1.

4. Experimentation

In this section, we propose an experimental study of the proposed many-core approaches for Intel Xeon Phi (offload and native) and compare them to their multi-core counterpart. We first describe the hardware and software testbeds and give some parameter settings used for our experiments. Then, we report and discuss some experimental results.

4.1. Hardware and software testbeds and parameter setting

For the multi-core implementation, we have used OpenMP version 4.0. All the experiments are run on hardware described in Table 1. The compiler used for the CPU and MIC implementations is Intel *icc* version 15.0 for Intel devices. 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. Finally, the GFLOPS(DP) row is obtained using the following computations:

- MIC: $16(flops/Ghz) \times 60(cores) \times 1.053(GHz/core) = 1010.88GF/s$
- CPU: $8(flops/Ghz) \times 8(cores) \times 2.6(GHz/core) = 166GF/s$

The last row indicates that the two hardware configurations (considering a two-socket multi-core processor) are equivalent in terms of energy consumption. For Intel Xeon E5-2670, as the server

	Intel Xeon E5-2670	Intel Xeon-Phi 5110P
#Physical cores	8	60
#Logical cores	16	240
clock(Ghz)	2.6	1.053
GFLOPS(DP)	166	1011
SIMD	256-bit	512-bit
Cache(MB)	20	30
Mem BW(GB/s)	51.2	320
Watt	115	225

Table 1: Hardware execution platform

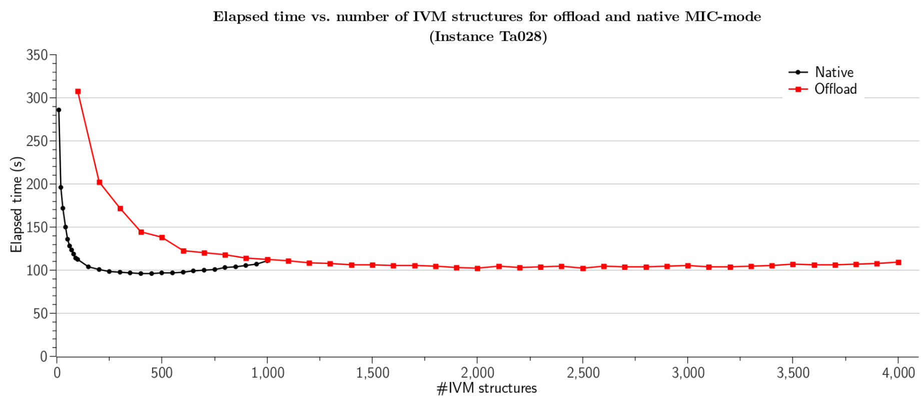

Figure 7: Elapsed time vs. Number of IVMs for MIC-B&B in offload and native mode.

includes two sockets its power (in Watt) value is equal to $115 \times 2 = 230$.

As quoted in Section 3.1, 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 must be explored simultaneously in order to generate multiple (IVM-based) pools maximizing the use of the coprocessor cores. Before the experiments are performed, the number M of IVM-based pools to be created is calibrated through a series of experiments on the problem instance $Ta028$ for the MIC (offload and native) and MC approaches. The experimental results are reported in Figure 7. Based on the figure, the number of IVMs is fixed to 2500 (resp. 240 and 32) for the offload-based MIC approach (resp. native-based MIC approach and multi-core approach).
In addition, in our experiments multithreading (2 threads per physical core) is used for the MIC-based approach. This parameter is determined experimentally as shown in Figure 8. The problem instance Ta028 (8.1 millions of decomposed subproblems) is considered for this calibration.

4.2. Experimental results

The performance analysis consists in comparing the performance of the parallel multi-core approach and the two many-core MIC-based approaches. The objective is to evaluate the impact on performance of the hardware architecture (multi-core *vs.* many-core), the programming mode

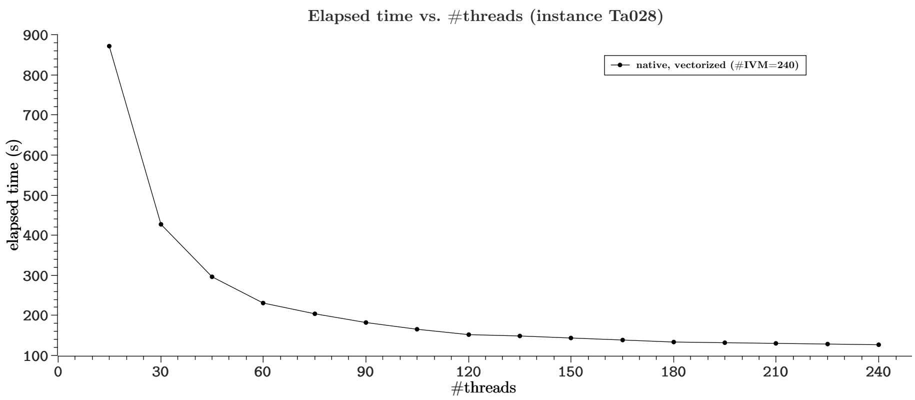

Figure 8: Impact of multithreading on the performance of MIC-based approach

of MIC (offload *vs.* native), the vectorization mechanism (vectorized *vs.* non-vectorized), and the parallelization model (master-worker *vs.* work pool). The different 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. This allows to prevent from the well-known speed up anomalies investigated for instance in [14].

4.2.1. Impact of hardware architecture, programming mode and vectorization

We first investigate three performance factors: the hardware architecture (MC *vs.* MIC), the programming mode of MIC (offload *vs.* native), and the vectorization mechanism (with *vs.* without). Table 2 reports the experimental results obtained on the $Ta021 - Ta030$ Taillard's instances. During the exploration of these instances on average 43.1 millions of subproblems are decomposed. The largest instance is $Ta023$ with 140.8 millions of decomposed subproblems. The experimental results shown in Table 2 are obtained using a static task assignment in the parallel bounding phase.

Table 2: Exploration time (in seconds) obtained using the multi-core and MIC master-worker approach.

	Nodes	Xeon Phi offload		Xeon Phi native		Multi-core	
Inst.	($\times 10^6$)	No-Vect	Vect	No-Vect	Vect	No-Vect	Vect
21	41.4	1,137	535	1,410	629	1,599	1,103
22	22.1	566	269	700	306	785	562
23	140.8	3,756	1,705	4,643	2,007	5,307	3,709
24	40.1	938	431	1,200	517	1,391	971
25	41.4	1,144	570	1,400	641	1,603	1,069
26	71.4	1,651	766	2,120	929	2,371	1,714
27	57.1	1,342	559	1,699	668	1,982	1,372
28	8.1	228	108	276	118	296	209
29	6.8	183	84	218	91	241	173
30	1.6	51	25	55	24	60	44
AVG	**43.1**	**1,100**	**505**	**1,372**	**593**	**1,564**	**1,092**

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 double-columns designate the exploration times in seconds obtained using respectively the MIC-accelerated offload-based approach, the MIC-accelerated native-based approach, and the multi-core approach. Each double-column designates the exploration times obtained without (NoVect) and with vectorization (Vect). The MIC offload-based (resp. native-based) approach uses 236 ((resp. 240) threads while the multi-core hyperthreading-based approach uses 32 threads. In [19], 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.

From the last row of the table, three major observations can be made. First, the proposed vectorization mechanism allows one to speed up the MIC-based (resp. multi-core) approach by about 55.4% (resp. 30.1%) in average. The improvement ratio is computed as follows: $\frac{NoVect-Vect}{NoVect}$. One can notice that the improvement is close to double on MIC than on MC because the size of the registers on Xeon Phi 5110P (512 bits) is double than the size of registers on Xeon E5-2670 (256 bits). Second, the offload-based MIC-accelerated approach is on average faster than its native-based counterpart whatever is the tackled problem instance.

Using the offload-based approach there is, on the one hand, the additional cost induced by processor-coprocessor data transfer. On the other hand, the partially sequential IVM-management is performed on the CPU and thus with a higher clock rate. The following subsection investigate this issue more closely. Finally, the MIC-accelerated approaches outperform their energy-equivalent multi-core counterpart for all problem instances in their non-vectorized as well as vectorized versions. For instance, considering their vectorized versions the offload-based (resp. native-based) MIC approach is 2.16× (resp. 1.84×) faster than its multi-core counterpart.

4.2.2. Impact of the parallelization model

Another series of experiments has been performed using the work pool parallel model with vectorization considering the MC and MIC architectures. The obtained results are reported in Table 3. The two first columns are the same as those of Table 2. The three following double-columns designate the exploration times (in seconds) obtained using respectively the MIC offload-based vectorized approach, the MIC native-based vectorized approach and the multi-core vectorized approach. Each double column comprises the time obtained using the master-worker (schedule=static) model and the time obtained using the work pool (schedule=dynamic).

From the last row of the table, two major observations can be made. First, for the two architectures the work pool model allows a faster resolution for all the tackled problem instances. This can be explained by the irregular nature of the lower bound calculation function illustrated in Figure 9. Subproblems with smaller lower bound calculation cost do not penalize the subproblems with a greater lower bound evaluation cost. Second, the benefit of using dynamic instead of static task assignment varies according to the target architecture and model. Using a dynamic instead of a static task assignment in the offload mode allows to decrease the average execution time by 6% while it allows an improvement of 17% (resp. 22%) in the native (resp. MC) case. These numbers correlate to the size of the task-pools evaluated in parallel. Because of averaging,

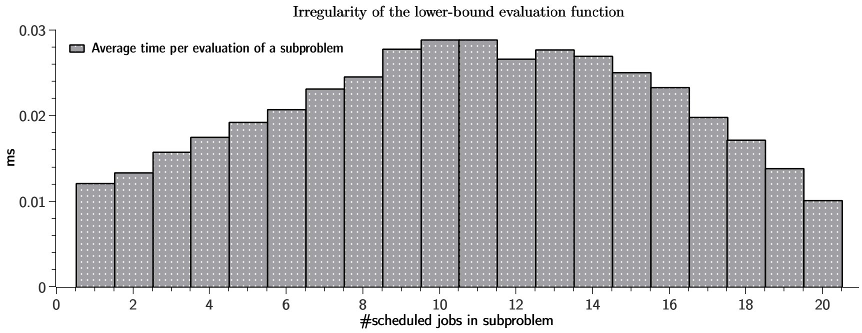

Figure 9: Illustration of the irregular nature of the bounding function. Average elapsed time (1000000 evaluations) required for computing the lower bound for a subproblems according to its depth in the tree.

when a very large pool is used a static task repartition is indeed likely to lead to smaller relative differences between per-thread workloads than the repartition of a small task-pool.

Considering the work pool model, the average resolution time for the offload and native models are nearly equivalent. The native-based algorithm outperforms its offload-based counterpart for the three smallest instances $Ta028 - Ta030$, while the latter is faster for all other instances. Considering the work pool model, the offload-based (resp. native-based) MIC approach is 1.78× (resp. 1.72×) faster than its multi-core counterpart.

4.2.3. Breakdown of execution time

In Figure 10 the average execution time (for instances $Ta021 - Ta030$) is decomposed into three parts: bounding, IVM-management and overhead induced by the parallelization of the bounding phase. The IVM-management time includes the selection of parent nodes and node elimination. The overhead is measured as follows. The parallel node evaluation is invoked twice: a first time without computing the lower bounds and a second time actually carrying out the evaluation of subproblems. The purpose of the first operation is to isolate the transferring

Table 3: Exploration time (in seconds) obtained using the vectorized multi-core and MIC master-worker and work pool approaches.

	Nodes	Xeon Phi offload Vect		Xeon Phi native Vect		Multi-core Vect	
Inst.	($\times 10^6$)	static	dynamic	static	dynamic	static	dynamic
21	41.4	535	501	629	520	1,103	843
22	22.1	269	254	306	259	562	434
23	140.8	1,705	1,608	2,007	1,673	3,709	2,862
24	40.1	431	396	517	411	971	749
25	41.4	570	540	641	559	1,069	834
26	71.4	766	714	929	747	1,714	1,320
27	57.1	559	524	668	545	1,372	1,083
28	8.1	108	102	118	99	209	163
29	6.8	84	80	91	78	173	133
30	1.6	25	24	24	20	44	34
AVG	**43.1**	**505**	**474**	**593**	**491**	**1,092**	**845**

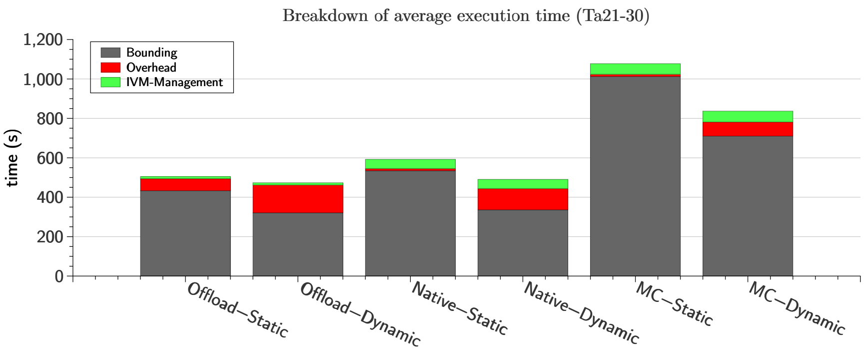

Figure 10: Time spend in bounding, IVM-management and overhead.

and marshaling of data, thread creation and the scheduling of the node evaluation loop from the computation of lower bounds. The time labeled as "overhead" in Figure 10 is obtained by measuring the time required for this first operation. The bounding time is obtained by subtracting the measured overhead from the time required to perform the actual evaluation of the lower bounds.

Compared to a master-worker model the work pool approach allows to reduce the time required to perform the parallel bounding operation, which reduces the overall execution time. On the other hand, Figure 10 clearly shows that the work pool model is costlier to manage than its static master-worker counterpart. However, for both the MC and MIC-based approaches the benefits resulting from a better workload repartition outweigh this increased overhead.

Unsurprisingly, the offload-based approach incurs the highest overhead among the three considered approaches. Compared to the native execution model, the offload model spends less time managing the pool of subproblems, as this part is performed with a higher clock rate on the CPU. This allows the offload-based approach to use a higher number of IVMs without being penalized by a high IVM-management cost. As mentioned, especially in the case of a static task assignment, a larger pool-size may also improve the performance of the bounding phase, reducing the load imbalance between threads. Indeed, comparing the results obtained for the statically scheduled offload and native models, one can notice that the offload-based algorithm spends less time in the bounding phase than its native-based counterpart. This difference disappears when switching to the dynamic task assignment. This indicates that the relative penalty induced by load imbalance is lower for larger overall work loads, i.e. a larger number of IVMs.

5. Conclusion

According to the recent Top500 ranking (November 2015) [1], 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, more exactly Intel Xeon Phi which emerged recently as a strong concurrent to Nvidia GPU. 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 including processor-coprocessor data transfer optimization and vectorization on Intel Xeon Phi. To deal with the first issue we have reused some recommendations proposed in [5] as quoted above. For the second issue, we have proposed a vectorization mechanism for the lower bound function. The different 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: Xeon Xeon E5-2670 and Xeon Phi 5110P. The reported results show that first, the vectorization mechanism allows in average an improvement of 55.4% (resp. 30.1%) on Xeon Phi 5110P (resp. Xeon E5-2670). Second, the offload mode allows a faster execution than the native mode due the processing of sequential and weakly parallel parts on the higher-clocked CPU. Hyper-threading (2 threads per physical core) is used to speed up the execution on Xeon Phi. Third, the MIC-based parallelization is up to twice faster than its multi-core counterpart whatever is the used mode on MIC: offload or native. Finally, the work pool parallel model is more suited than the master-worker model for B&B applied to the Flow-Shop problem because of its irregular nature.

In the near future, we plan to extend our work to deal with parallel exact optimization on hybrid architectures combining multi-core processors, Xeon Phi coprocessors and GPU accelerators. Work partitioning between the three devices will be a particular challenging issue [7]. Then, we will extend the resulting hybrid approach with a cluster-level parallelism. This will allow us to solve efficiently the hardest unsolved Taillard's instances of the Flow-Shop problem. Finally, we believe that the conclusions drawn from the experiments reported in this paper will be useful for the parallelization of other tree-based applications. Therefore, in the long term, we will try to generalize the approach to other tree-based exploration algorithms such as B&X (X=cut, price, ...).

References

[1] Top500 HPC international ranking, `http://www.top500.org/`.

[2] L. Barreto and M. Bauer. Parallel branch and bound algorithm-a comparison between serial, openmp and mpi implementations. In *Journal of Physics: Conference Series*, volume 256, page 012018. IOP Publishing, 2010.

[3] I. Chakroun, N. Melab, M. Mezmaz, and D. Tuyttens. Combining multi-core and gpu computing for solving combinatorial optimization problems. *Journal of Parallel Distributed Computing*, 73(12):1563–1577, 2013.

[4] I. Chakroun, M. Mezmaz, N. Melab, and A. Bendjoudi. Reducing thread divergence in a gpu-accelerated branch-and-bound algorithm. *Concurrency and Computation: Practice and Experience*, 25(8):1121–1136, 2013.

[5] Imen Chakroun. Parallel heterogeneous branch and bound algorithms for multi-core and multi-gpu environments. *PhD Thesis from Université Lille 1*, 2013.

[6] George Chrysos. 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.

[7] J. Dokulil, E. Bajrovic, S. Benkner, S. Pllana, M. Sandrieser, and B. Bachmayer. High-level Support for Hybrid Parallel Execution of C++ Applications Targeting Intel® Xeon Phi$^{\text{TM}}$ Coprocessors. *Procedia Computer Science*, 18:2508 – 2511, 2013. 2013 International Conference on Computational Science.

[8] M.R. Garey, D.S. Johnson, and R. Sethi. The complexity of flow-shop and job-shop scheduling. *Mathematics of Operations Research*, 1:117–129, 1976.

[9] S.M. Johnson. Optimal two and three-stage production schedules with setup times included. *Naval Research Logistis Quarterly*, 1:61–68, 1954.

[10] I.M. Kulikov, I.G. Chernykh, A.V. Snytnikov, B.M. Glinskiy, and A.V. Tutukov. Astrophi: A code for complex simulation of the dynamics of astrophysical objects using hybrid supercomputers. *Computer Physics Communications*, 186:71 – 80, 2015.

[11] Vipin Kumar. *Introduction to Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

[12] M.E. Lalami. Contribution à la résolution de problèms d'optimisation combinatoire: méthodes séquentielles et parallèles. *Université de Toulouse III - Paul Sabatier.*, 2012.

[13] J.K. Lenstra, B.J. Lageweg, and A.H.G. Rinnooy Kan. A General bounding scheme for the permutation flow-shop problem. *Operations Research*, 26(1):53–67, 1978.

[14] B. Mans and C. Roucairol. Performances of parallel branch and bound algorithms with best-first search. *Discrete Applied Mathematics*, 66(1):57–74, 1996.

[15] N. Melab, I. Chakroun, M. Mezmaz, and D. Tuyttens. 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*, pages 10–17, 2012.

[16] Nouredine Melab, Imen Chakroun, and Ahcène Bendjoudi. 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.

[17] S. Memeti and S. Pllana. Accelerating DNA Sequence Analysis Using Intel(R) Xeon Phi(TM). In *Proceedings of the 2015 IEEE Trustcom/BigDataSE/ISPA - Volume 03*, TRUSTCOM-BIGDATASE-ISPA '15, pages 222–227. IEEE Computer Society, 2015.

[18] M. Mezmaz N. Melab, R. Leroy and D. Tuyttens. Parallel branch-and-bound using private ivm-based work stealing on xeon phi mic coprocessor. In *IEEE Proc. of Intl. Conf. on High Performance Computing & Simulation*, HPCS'2015, 2015.

[19] S. Saini, H. Jin, D.C. Jespersen, S. Cheung, M.J. Djomehri, J. Chang, and R. Hood. 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*, pages 57–67, 2015.

[20] M. Stanic, O. Palomar, I. Ratkovic, M. Duric, O. Unsal, A. Crista, and M. Valero. Evaluation of Vectorization Potential of Graph500 on Intel Xeon Phi. In *in Proc. of Intl. Conf. on High Performance Computing & Simulation (HPCS'2014), Bologna, Italy*, pages 47–54, Jul. 2014.

[21] A. Viebke and S. Pllana. The Potential of the Intel (R) Xeon Phi for Supervised Deep Learning. In *Proc. of the 2015 IEEE 17th Intl. Conf. on High Performance Computing and Communications, 2015 IEEE 7th Intl. Symp. on Cyberspace Safety and Security, and 2015 IEEE 12th Int. Conf. on Embedded Software and Systems*, HPCC-CSS-ICESS '15, pages 758–765. IEEE Computer Society, 2015.

[22] Trong-Tuan Vu. Heterogeneity and locality-aware work stealing for large scale branch-and-bound irregular algorithms. *PhD Thesis from Université Lille 1*, 2014.