Preliminary Experiments with XKaapi on Intel Xeon Phi Coprocessor

Joao Vicente Ferreira Lima, Francois Broquedis, Thierry Gautier, Bruno Raffin

Preliminary Experiments with XKaapi on Intel Xeon Phi Coprocessor

João V. F. Lima*‡, François Broquedis*, Thierry Gautier†, Bruno Raffin†
* Grenoble Institute of Technology, France
† INRIA, Grenoble, France
‡ Federal University of Rio Grande do Sul (UFRGS), Brazil
jvlima@inf.ufrgs.br, francois.broquedis@imag.fr, thierry.gautier@inrialpes.fr, bruno.raffin@inria.fr

***Abstract*—This paper presents preliminary performance comparisons of parallel applications developed natively for the Intel Xeon Phi accelerator using three different parallel programming environments and their associated runtime systems. We compare Intel OpenMP, Intel CilkPlus and XKaapi together on the same benchmark suite and we provide comparisons between an Intel Xeon Phi coprocessor and a Sandy Bridge Xeon-based machine. Our benchmark suite is composed of three computing kernels: a Fibonacci computation that allows to study the overhead and the scalability of the runtime system, a NQueens application generating irregular and dynamic tasks and a Cholesky factorization algorithm. We also compare the Cholesky factorization with the parallel algorithm provided by the Intel MKL library for Intel Xeon Phi. Performance evaluation shows our XKaapi data-flow parallel programming environment exposes the lowest overhead of all and is highly competitive with native OpenMP and CilkPlus environments on Xeon Phi. Moreover, the efficient handling of data-flow dependencies between tasks makes our XKaapi environment exhibit more parallelism for some applications such as the Cholesky factorization. In that case, we observe substantial gains with up to 180 hardware threads over the state of the art MKL, with a 47% performance increase for 60 hardware threads.**

I. Introduction

Nowadays computing platforms expose a great number of heterogeneous processing units. Large-scale applications from the industry usually require mixing different parallelization paradigms to exploit such machines at their full potential. However, designing parallel environments and runtime systems that support multiple paradigms on a portable and efficient way is still challenging. In [1], [2], we introduced XKaapi, a runtime system for multi-CPUs and multi-GPUs architectures designed to support recursive tasks with data-flow dependencies. XKaapi-based applications achieve good performance thanks to a low-overhead scheduler that comes with NUMA-aware heuristics to improve data locality.

GPU and multi-GPU programming has been considered as a promising way to exploit HPC platforms efficiently, especially when trying to reach a high performance/energy ratio. However, GPUs are known to be complex to program efficiently [3] as GPU programming requires a strong expertise to write optimized code. The applied optimizations may also not be compatible from a specific GPU generation to the next, making GPU kernels difficult to maintain.

With the introduction of the Intel Xeon Phi coprocessor, Intel proposed a strong evolution in the way to develop applications for accelerators. Many researchers and research projects have recently moved their focus on this architecture [4]–[9], trying to position the Intel Xeon Phi as a good candidate for executing efficient high-performance parallel applications. For instance, the European DEEP project aims at studying the contribution of the Intel Xeon Phi technology in the design of a novel architecture for *paving the way* to Exascale. But how to program it efficiently?

Achieving high performance on multicore architectures requires several threads of control running mostly independent code, with limited synchronizations to ensure a smooth progress of the computation. Programming directly with threads is considered as highly unproductive and error prone [10]. Many parallel programming environments have been proposed to exploit such architectures, and two of them, Cilk and OpenMP, behave especially well for executing fine-grain parallelism. Cilk [11] promotes a fork-join parallel paradigm with theoretical guarantees on the expected performance. OpenMP [12] relies on code annotations to generate parallel programs. Both Cilk and OpenMP have basic constructs to create independent tasks and to parallelize independent loops. Moreover, the OpenMP user can also guide the way loop iterations are scheduled among the threads. Intel provides a rich set of parallel programming environments like Pthreads [13], OpenMP, CilkPlus based on Cilk and TBB [14] running on Xeon Phi, letting the application programmer choose the one that best suits his needs. The availability of all these environments clearly positions the Intel Xeon Phi coprocessor as a reliable target for accelerated applications.

Even if these programming environments relieve the application programmer's pain, they may not be suited for large-scale shared-memory architectures like the 240-threads Intel Xeon Phi coprocessor. In particular, several studies [15], [16] show that the strong synchronizations imposed by both OpenMP and Cilk execution models artificially limit the available parallelism: in these programming models, the synchronization construct block the running thread until previously spawned tasks have completed their execution. For instance, in a classical nested loops formulation of matrix factorization [15], [16], both OpenMP and Cilk enforce periodic synchronizations that block the execution of tasks from the next iteration. These studies emphasize data-flow approaches that are able to expose fine-grain one-to-one synchronizations between tasks: the runtime system can detect concurrent tasks as soon as their inputs are produced.

Such data-flow programming model is a promising approach to take into account data transfers between disjoint

memory address spaces. It was successfully validated on multi-CPUs / multi-GPUs architectures [1], [17]–[19] containing up to 8 GPUs sharing a complex hierarchy of PCIe buses [2] and on large-scale distributed platforms [18], [20].

Although many research papers and technical reports aim at comparing parallel programming environments, only a few of them target the Intel Xeon Phi [4]–[9].

In this paper, we present preliminary performance evaluations of the XKaapi data-flow runtime on native Intel Xeon Phi applications: our goal is to study the strengths and the weaknesses of XKaapi to program native applications. The contributions of this paper are :

- A novel experimental evaluation on porting a high-performance data-flow programming environment for the Intel Xeon Phi coprocessor;
- A performance comparison with respect to the native programming environments CilkPlus and OpenMP provided by Intel on micro benchmarks and matrix Cholesky factorization code;
- A performance comparison of the Xeon Phi architecture with respect to the performance reached by a classical CPU Intel Xeon architecture running the same parallel programming environments.

Section II overviews the XKaapi's parallel programming model and the difficulties encountered during the port of XKaapi to the Intel Xeon Phi coprocessor. Section III reports experimental evaluations compared to OpenMP and CilkPlus, the native parallel programming environments proposed by Intel to develop Xeon Phi's applications. Section IV presents related works on runtime systems and experiments using the Xeon Phi coprocessor. Section V concludes the paper and suggests future works.

II. XKaapi on Intel Xeon Phi

A. Overview of XKaapi

The XKaapi[1] task model [20], as in Cilk [21], Intel TBB [14], OpenMP-3.0 or StarSs [18], [22], enables non-blocking task creation: the caller creates the task and proceeds with the program execution. The semantics remain sequential, as in XKaapi's predecessors Athapascan [23] and KAAPI [20], which was specialized for multi-CPU/multi-GPU iterative applications [17].

XKaapi has several APIs (C, Fortran, C++, prototype of compiler directives) to program heterogeneous parallel architectures. In this paper, code fragments are presented using the C++ API. More information about heterogeneous multi-CPUs and multi-GPUs parallel programming and scheduling can be found in Lima et al. [1] and Gautier et al. [2].

1) Data-Flow Task Definition and Creation: A XKaapi program is composed of sequential code and some annotations or runtime calls to create tasks. The parallelism in XKaapi is explicit, while the detection of synchronizations is implicit [20]: the dependencies between tasks and the memory transfers are automatically managed by the runtime.

[1] http://kaapi.gforge.inria.fr

A task is a function call that returns no value except through the shared memory and the list of its effective parameters. Depending of the APIs, tasks are created either using code annotations (`#pragma kaapi task` directive) if the XKaapi prototype compiler is used, or library functions (`kaapic_spawn` call using XKaapi's C API, or template function `ka::Spawn`), or by low level runtime function calls.

Tasks share data if they have access to the same memory region. A memory region is defined as a set of addresses in the process virtual address space. This set has the shape of a multidimensional array. The user is responsible for indicating the mode each task uses to access memory: the main access modes are *read*, *write*, *reduction* or *exclusive* [20], [23]. When required, the runtime computes true dependencies (Read after Write dependencies) between tasks thanks to the access modes [2], [20]. At the expense of memory copying, the scheduler may solve false dependencies through variable renaming.

2) Dynamic Scheduling by Work Stealing: The runtime creates a system thread for each computation resource to be used. On multi-CPU, a resource is a core. A thread creates tasks and pushes them on its own work queue, which is represented as a stack. The enqueue operation is very fast, typically about ten cycles on the last x86/64 processors [24]. As in Cilk, a running XKaapi task can create children tasks. This is not the case for the other data-flow programming software previously mentioned [19], [22], [25], except StarSs recent extension OmpSs [18]. Once a task ends, the thread executes its children following a First-in First-out (FIFO) order by popping tasks from its own work queue.

During task execution, if a thread finds a stolen task, it suspends its execution and switches to the work stealing scheduler that waits for dependencies to be met before resuming the task. Thanks to Cilk [11], [21], the work stealing technique has become popular and is often considered when it comes to dynamically balance the work load among processing units. The work stealing principle can be summarized as follows. An idle thread, called a thief, initiates a steal request to a random selected victim. On reply, the thief receives a copy of one ready task, leaving the original task marked as stolen. Coherency between a thief and its victim is ensured by a variant of the Cilk's T.H.E protocol [21].

To find a ready task, a thief thread iterates through the victim's queue from the least recently pushed task to the most recently one and it computes true data-flow dependencies for each task. The iteration stops on the first task found ready.

In addition, XKaapi runtime allows the cooperation among thieves, i.e., multiple steal requests to the same processor with reduction of steal requests [26]. Previously, when $k > 1$ steal requests were sent on the same processor, only one request could be served. If a thief finds the victim's on a stealing state, it aborts the steal request and selects another victim randomly. The XKaapi cooperation divides the work into $k + 1$ pieces when k steal requests target the same processor.

B. Adaptation to Intel Xeon Phi

The Intel Xeon Phi is made of several cores (up to 61 on the 7100 serie). The cores have their own memory that

is cache coherent using a full MESI coherency protocol. Remote memory accesses are managed by the communication network (a full-duplex ring among the cores). The instruction set is based on the classical x86 instruction set with specific extensions to address SIMD capabilities and large vector operations. Moreover, the processor does not reorder memory read and write instructions, which releases the application programmer from guarding memory accesses with expensive memory barriers.

The Intel Xeon Phi can be seen as a set of hyperthreaded cores that share a global memory organized by chunks, which is not very far from a multicore NUMA architecture. Porting XKaapi source code to the Xeon Phi was not difficult, mainly requiring to specialize memory barriers and atomic operations to take into account the Xeon Phi specificities.

XKaapi thread binding was also modified to fit the Xeon Phi architecture. Assuming the coprocessor has p physical cores and each core supports h hardware threads, the total number of logical processors on the Intel Xeon Phi is $p * h$. Instead of distributing XKaapi threads in a sequential order from 0 to $p * h - 1$, XKaapi fills all physical cores with one thread in a round-robin fashion until all threads are created. An execution with c threads where $c > p$ will set up a distribution like: $0, h, 2 * h, 3 * h, ..., (p-1) * h, 1, h+1, 2(h+1), ...$

The work stealing scheduler was not adapted to the internal topology of the architecture. Indeed, several previous works [2], [20], [24] have demonstrated the good scalability of XKaapi even at fine grain, so we decided to keep it unmodified for this first evaluation.

III. Experiments

This section presents the experimental results of the XKaapi runtime system on both an Intel Xeon Phi coprocessor and an Intel SandyBridge processor. We describe the benchmarks and the obtained performance on both platforms, with an emphasis on the impact of the optimizations implemented in XKaapi. All times reported in this section are average of more than 30 executions with a warm-up phase of 2 runs.

A. Platform and Environment

All the applications were executed natively on the Intel Xeon Phi environment. The Xeon Phi used is a 5110P with 60 cores running at 1.053 Ghz and sharing 8 GB of memory. Each core has support to 4 hardware threads, for a total of 240 threads.

The Intel Sandy Bridge platform, hereafter called *SandyBridge*, contains 4 Intel Xeon E5-4620 multicore processors for a total of 32 cores running at 2.20GHz and sharing 384 GB of main memory. On this machine, hyperthreading was activated and enabled 2 hardware threads per core.

The software environment used on SandyBridge was the following: the operating system was a Debian distribution with a 3.8.11 Linux kernel; the OpenMP and CilkPlus applications were compiled with the Intel C/C++ compiler 13.1.3 and executed using the corresponding runtime system from Intel. Intel Xeon Phi's firmware version was 1.14.4616 and comes with version 13.0.1 of the Intel C/C++ compiler, MPSS 2.1.6720-13 and compiler_xe_2013.1.117. We evaluated XKaapi version 2.1 with the modifications described in section II of this paper. XKaapi applications were compiled with the same Intel compilers used to compile OpenMP and CilkPlus applications.

B. Fibonacci

This benchmark computes the n-th Fibonacci number using a naive recursive computation. The purpose of this micro-benchmark is to compare the overheads and scalability of the runtime systems that come with the OpenMP, the CilkPlus and the XKaapi programming environments on both the Intel Xeon Phi coprocessor and the SandyBridge machine.

Tseq=0.27s	OpenMP	CilkPlus	XKaapi
#thread =1	9.07	5.18	2.87
4	13.27	1.29	0.72
8	7.84	0.66	0.36
16	4.33	0.34	0.19
24	3.10	0.24	0.13
32	2.30	0.19	0.10
48	2.88	0.16	0.09
64	5.01	0.15	0.08

TABLE I. Times (in seconds) for Fibonacci N=38 on Intel SandyBridge.

Tseq=3.77s	OpenMP	CilkPlus	XKaapi
#thread=1	65.64	33.21	15.52
10	33.12	3.34	1.58
20	17.54	1.66	0.79
40	9.29	0.83	0.39
60	6.30	0.56	0.27
120	3.86	0.38	0.18
180	3.27	0.37	0.17
240	3.18	0.37	0.18

TABLE II. Times (in seconds) for Fibonacci N=38 on Intel Xeon Phi.

The code executed by each of the three environments generates the same number of tasks: each recursive call creates two child tasks (with `#pragma omp task`, `cilk_spawn` or `ka::Spawn`) to compute the $n-1$ and $n-2$ Fibonacci numbers in parallel, and then synchronizes the created tasks (with `#pragma omp taskwait`, `cilk_sync` or `ka::Sync`) before returning the sum of the two subresults. The recursion stops when the computation of the Fibonacci number is less than 2. The codes were compiled with Intel `icpc` and the `-O3` option. On SandyBridge, threads were explicitly bound to physical cores for OpenMP and XKaapi.

1) Overall Analysis: Table I and Table II report experimental results on SandyBridge and Intel Xeon Phi respectively. The results obtained by this benchmark shows XKaapi has the lowest overhead among the three tested environments. As highlighted in [2], [24], XKaapi intrinsic overheads due to the computation of the data-flow dependencies between tasks are incurred by steal operations. If the number of steal operations is very small compared to the number of created tasks, as in Fibonacci [21], data-flow related overheads do not impact XKaapi's performance obtained.

2) Parallel Programming Environment Scalability: To study the scalability of runtime systems, we compared the execution times obtained using each environment against the time of the parallel program executed on a single core, *i.e.* $S = T_1/T_p$. The speedups for CilkPlus and XKaapi environments were similar. On Sandbrige, CilkPlus reached a speedup

of 27 on 32 physical cores and 35 on 64 hardware threads while XKaapi reached respectively speedups of 28 and 36. On Intel Xeon Phi, CilkPlus reached a speedup of 59 on 60 hardware threads and 90 on 240 hardware threads while XKaapi reached respectively speedups of 57 and 86.

3) OpenMP Performance Issues: On the contrary, Intel OpenMP exhibited poor performance for this micro-benchmark with fine grain recursive tasks: on SandyBridge, taking the same definition as above, the speedup reached 4 on 32 cores and fell down to 1.8 on 64 hardware threads. One reason could be the fact that the 1-core execution is optimized to avoid task creation, performing simple function calls as for the sequential code. The reference time T_1 does not include overheads that only appears when several cores are used. In opposite, these overheads are present in the T_1 timing on the Intel Xeon Phi. Nevertheless, the maximum reported speedup with OpenMP is 20 compared to 90 obtained by CilkPlus. We already noticed that the overheads of GNU/GCC libGOMP runtime system [24] on fine grain task-based programs were large, and smaller for Intel's OpenMP implementation. On this task-based program, the Intel OpenMP runtime could be improved to achieve better performance, for instance by using the approach described in Broquedis et al. [24].

4) Comparing both Architectures: On this micro-benchmark, on 1-core execution, the Intel Xeon Phi was about 14 times slower than SandyBridge (sequential code), but only 6.4 times slower with CilkPlus and 5.4 slower with XKaapi. If we look at the maximum performance on both architectures, the Intel Xeon Phi was only between 2.12 and 2.4 times slower than SandyBridge on the tested benchmark.

The SandyBridge machine is made of 4 sockets with 8 cores and 16 hardware threads. On this benchmark, the performance of the Intel Xeon Phi was close to the one of a single SandyBridge socket.

C. NQueens

The NQueens benchmark is based on the Takaken [27] optimized sequential code to compute the number of solutions for the NQueens problems. It has been parallelized using XKaapi since 2007 [20] and we adapted it to OpenMP and CilkPlus. We have decided not to consider the OpenMP BOTS NQueens program as baseline as it runs slower than Takaken's code, mainly because it does not take symmetries of the configuration into account. Sequential execution of our code is about 1200 times faster than BOTS NQueens for $N = 16$ using the same `icpc` compiler with the `-O3` option.

1) Implementation: The principle of the parallelization is a recursive exploration of the different configurations of the chessboard: a set of possible configurations is generated at each recursive call, taking symmetries into account [27]. Each configuration is explored by an independent task. On final recursion, possible solutions are accumulated in a global variable. The parallelism is generated until a threshold, then the code performs sequential exploration.

The OpenMP, CilkPlus and XKaapi codes generate the same independent tasks. The main difference between the three environments resides in the way solutions are accumulated. As the original code relies on a 3D vector of solutions holding each of the 3 considered symmetries, the OpenMP version uses a critical region to accumulate the solutions. The CilkPlus version behaves similarly, using a mutex to implement the same kind of critical region. So, for each accumulation, these runtime sytems perform an *a priori* synchronization before accessing the global variable.

2) Accumulation in XKaapi: The XKaapi version creates tasks with access to the global variable declared as "*cumulative write access*" [20], [23], which allows to accumulate arbitrary data with a user-defined associative operator. When a thief thread steals a task, the runtime creates a new *per thief thread* data that the stolen task and its descendants use for the accumulation. When the stolen task completes, the new data is accumulated to the victim thread's data. At the end, the global variable contains the final accumulated result. This mechanism enables the XKaapi runtime to reduce the required synchronizations compared to OpenMP and CilkPlus.

3) Threshold Impact: Figure 1 illustrates the impact of the grain size in the NQueens benchmark. Both CilkPlus and OpenMP results can be explained by the overhead when executing fine-grain tasks (greatest threshold in the Figure). On the other hand, XKaapi seems to be able to efficiently execute applications at finer task grains while limiting the negative impact of runtime-related overheads on the overall execution time. We note that setting the threshold to bigger values will generate more parallelism, creating more fine-grain tasks.

On 64 hardware threads (32 cores) of SandyBridge, the minimum time for XKaapi was obtained with a threshold of $t = 6$, which is different from the one used by CilkPlus ($t = 3$) and OpenMP ($t = 5$). On the 240 threads of Intel Xeon Phi, we set the threshold to $t = 3$ for all the environments.

4) Scalability: Figure 2 reports the speedup $S = T_p/T_{seq}$ for NQueens ($N = 17$) on SandyBridge and Intel Xeon Phi. As noted before, each programming environment should impose a threshold due to the overhead related to task management. For each environment, we report the performance obtained using the best threshold.

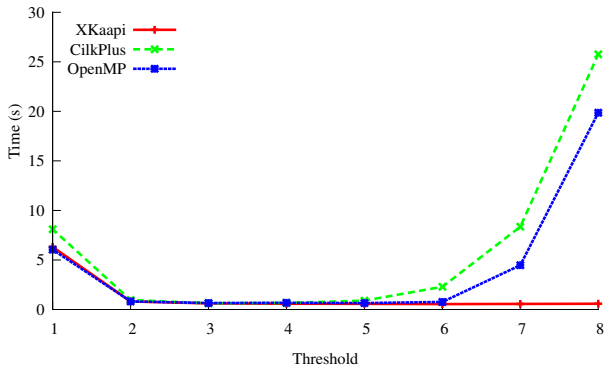
Like for the Fibonacci benchmark, XKaapi had the smallest overhead among all tested environments. The T_1 execution time of XKaapi was a little faster than the pure sequential program.

5) Comparing both Architectures: For NQueens ($N = 17$), the best execution time on Intel Xeon Phi obtained by XKaapi was $1.20s$ with 240 threads. On SandyBridge, the best execution time on 16 hardware threads over 8 physical cores was $1.33s$. For this benchmark, we can confirm that the Intel Xeon Phi coprocessor may be able to reach better performance than a single SandyBridge socket.

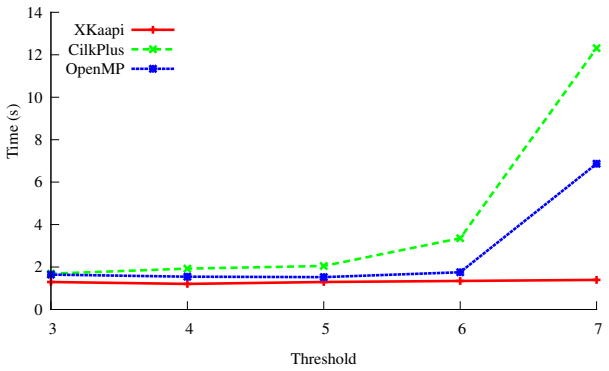
D. Cholesky

The Cholesky factorization (POTRF) decomposes an $n \times n$ real symmetric positive definite matrix A into the form $A = LL^T$ where L is an $n \times n$ real lower triangular matrix with positive diagonal elements [28].

1) Parallel programs: Figure 3 shows the pseudo-code of both the XKaapi and the CilkPlus versions. The main difference between the two versions is the absence of synchronization in the XKaapi code thanks to data-flow dependencies

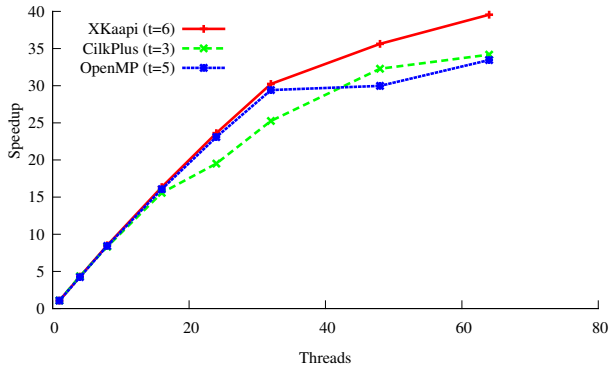


(a) Intel SandyBridge with 32 threads.

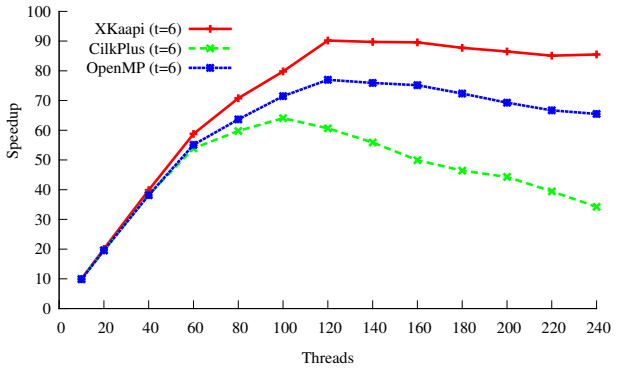


(b) Intel Xeon Phi with 240 threads.

Fig. 1. Time *versus* threshold for NQueens benchmark (N=17) for XKaapi, CilkPlus and OpenMP.



(a) Intel SandyBridge.



(b) Intel Xeon Phi.

Fig. 2. Scalability of the NQueens benchmark (N=17) for XKaapi, CilkPlus and OpenMP. Each programming environment runs with the threshold obtaining the best performance. Speedups were computed against the sequential execution time: $21.67s$ on Intel SandyBridge and $114.95s$ on Intel Xeon Phi.

between tasks: the user is responsible for indicating the mode each task uses to access memory: the main access modes are *read*, *write* (or *reduction*, but not used here) that let the runtime system build dependencies between tasks that access to same memory region. The XKaapi code illustrates the C++ interface that does not require a specific compiler: in the notation `A(ri,rj)`, `ri` and `rj` denote a range of indexes so that `A(ri,rj)` is a sub-matrix of matrix `A`. Thanks to this finer knowledge of tasks dependencies, the runtime system can schedule ready tasks between two main iterations in k [15], [16]. The OpenMP version is similar to the CilkPlus version by replacing `cilk_spawn` by `#pragma omp task` and `cilk_sync` by `#pragma omp taskwait`.

Our experiments use the same parallel version of the Cholesky factorization, as found in PLASMA [29]. The algorithm has been re-implemented in two versions: block version (same as in PLASMA) and block-recursive version [2]. The block-recursive Cholesky is a two level parallel algorithm: at the upper level, we use the PLASMA algorithm; at the lower level the POTRF task is parallelized using the same parallel algorithm as at upper level by decomposing one tile in sub-tiles of size 32×32. We have not used auto-tuning to select the sizes of the tile and sub-tile, but an empirical approach: after a few experiments showing their average good performances, we have decided to use theses values.

2) Scalability: Figure 4 reports the GFlop/s rate obtained for a matrix of size 8192×8192 with tiles of size 256×256. The overall best performance was obtained by XKaapi for both architectures. On SandyBridge, OpenMP and CilkPlus had a similar level of performances. The XKaapi block and block-recursive versions, described in the previous section, showed a small gain with respect to the recursive version because of a higher level of parallelism. On the Intel Xeon Phi, the behavior is the same than on SandyBridge, except the XKaapi block-recursive version showed an important performance improvement. Results on bigger matrices follow the same behavior (Figure 5) on the two architectures.

On the Intel Xeon Phi architecture, the cores are very efficient on regular vector operations, which is the case for tasks TRSM, GEMM and SYRK of Figure 3, but not for the POTRF task. On the Cholesky factorization, the POTRF tasks on diagonal block `A(ri,ri)` are on the critical path of the execution. Any reduction in the completion of POTRF tasks allows other cores to resume their execution, thus reducing idle time. The same phenomenum was observed on multi-CPUs/multi-GPUs factorization [2], [15], [28] where POTRF tasks are inefficient on GPU, thus to decrease execution time, tasks belonging to the critical path have to be parallelized. On multi-CPUs/multi-GPUs this was done by executing PORTF tasks on CPUs. On the Intel Xeon Phi, the same performance improvement can be obtained by parallelizing POTRF tasks as

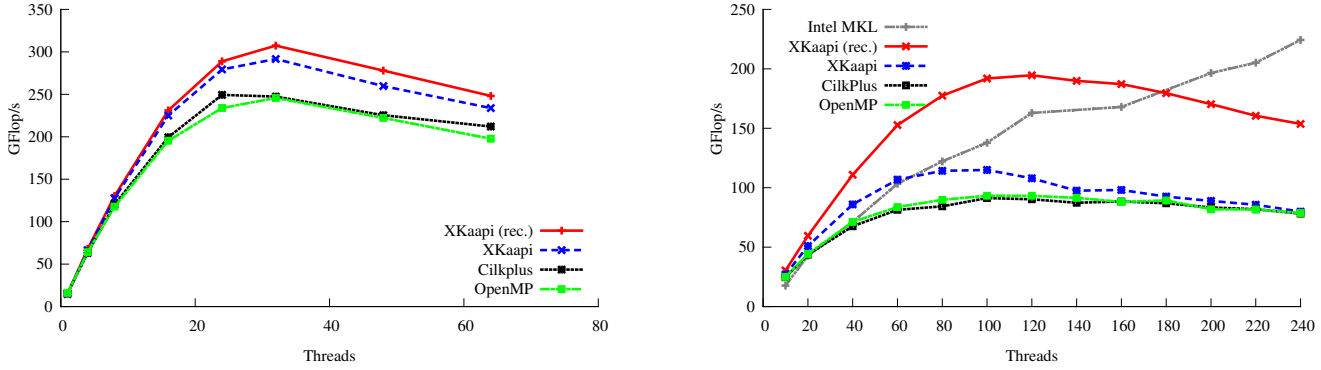
```
/* XKaapi */
for( k=0; k < NB; k++ ) {
  ka::Spawn<TaskPOTRF>()( A(k,k) );
  for( m=k+1; m < NB; m++ )
    ka::Spawn<TaskTRSM>()( A(k,k), A(m,k) );

  for( m=k+1; m < NB; m++ ) {
    ka::Spawn<TaskSYRK>()( A(m,k), A(m,m) );
    for( n=k+1; n < m; n++ )
      ka::Spawn<TaskGEMM>()( A(m,k), A(n,k), A(m,n) );

  }
}
```

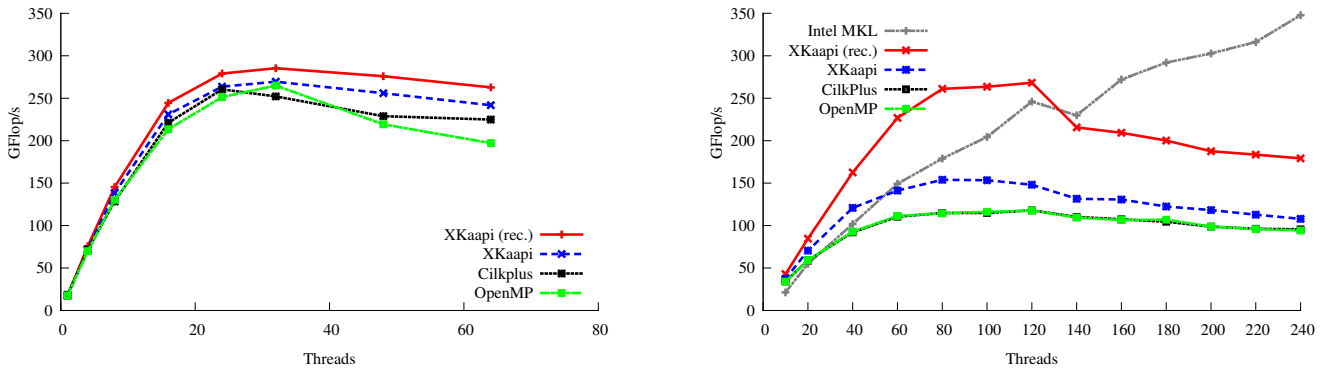
```
/* CilkPlus */
for( k=0; k < NB; k++ ) {
  POTRF( A(k,k) );
  for( m=k+1; m < NB; m++ )
    cilk_spawn TRSM( A(k,k), A(m,k) );
  cilk_sync;
  for( m=k+1; m < NB; m++ ) {
    cilk_spawn SYRK( A(m,k), A(m,m) );
    for( n=k+1; n < m; n++ )
      cilk_spawn GEMM( A(m,k), A(n,k), A(m,n) );
  cilk_sync;
  }
}
```

Fig. 3. Example of a left-looking Cholesky factorization with XKaapi and CilkPlus. PORTF, TRSM, SYRK, GEMM are classical BLAS or LAPACK subroutines. The OpenMP version (not shown) is very close to the CilkPlus version (replacement of `cilk_spawn` by `#pragma omp task` and `cilk_sync` by `#pragma omp taskwait`).



(a) Intel SandyBridge. (b) Intel Xeon Phi.

Fig. 4. Results of Cholesky benchmarks for XKaapi, OpenMP, and CilkPlus for matrix size 8192×8192 and tile size 256×256. MKL was only measured on the Intel Xeon Phi and omitted on the Intel SandyBridge.



(a) Intel SandyBridge. (b) Intel Xeon Phi.

Fig. 5. Results of Cholesky benchmarks for XKaapi, OpenMP, and CilkPlus for a matrix of size 16384×16384 and tile size 512×512. MKL was only measured on the Intel Xeon Phi and omitted on the Intel SandyBridge.

performed by the XKaapi block-recursive version described in our previous multi-GPU implementation [2].

3) Comparison with MKL: Figures 4 and 5 show an interesting behavior on the Intel Xeon Phi. If we compare MKL performance with the two XKaapi codes (block-recursive and block versions), XKaapi versions were above the performance of MKL in most cases. The block-recursive version was up to 47% faster than MKL (for 60 threads) and faster than MKL up to 180 threads. Moreover, for a matrix size of 8192×8192, the XKaapi version reached 152.67 GFlop/s with 60 threads, 191.9 GFlop/s with 100 threads and only 194.57 GFlop/s with 120 threads. The MKL reached 103.35 GFlop/s with 60 threads, 137.8624 with 100 threads and 224.28 GFlop/s with all the 240 threads.

Without any description of how the MKL POTRF routine is implemented, the XKaapi version was able to provide a higher

ratio GFLops/thread than MKL up to 180 threads. We will investigate further why the XKaapi performance drops after 120 threads. These findings led us to believe that our initial port on Intel Xeon Phi may not take care of affinity between tasks and data, and our randomized work stealing may have an important memory footprint. Techniques we have developed for multi-GPUs [2] would be tested to improve the scalability of our runtime system.

4) Comparing both Architectures: The XKaapi block-recursive Cholesky factorization obtained at most 81.15 GFlop/s on one SandyBridge socket for a matrix size of 8192×8192. The same code on Intel Xeon Phi performed at 194.57 GFlop/s, and the MKL was at most 224.28 GFlop/s. Therefore, one Intel Xeon Phi was $2.4x$ more powerful than one SandyBridge socket on this benchmark.

IV. Related Work

Several runtime tools and languages were based on a data-flow paradigm, such as Athapascan [23] used for sparse Cholesky factorizations [30]. QUARK [25] is the data-flow runtime system of the PLASMA dense linear algebra library [29]. StarPU [19] is a runtime system for scheduling a DAG of tasks on multi-CPU and multi-GPU architectures. The StarSS programming model with its current implementation called OmpSs [18] provides a set of OpenMP-like pragmas and a runtime system to schedule tasks. Except OmpSs in the technical report [5] from the EU FP7 project DEEP, none of these softwares report experiments on the Intel Xeon Phi architecture.

In [5] preliminary experiments on Cholesky factorization were reported on pre-release prototypes of the Intel Xeon Phi (Intel Knights Corner and Intel Knights Ferry) for matrix of size 8192×8192, such as in Figure 4. Due to the lack of details in this report, comparisons are difficult: the authors reported about 98 GFlop/s with MKL. The maximal reached performance of OmpSs with "task nesting" [5] was around 78 GFlop/s. In comparison, for the same matrix size, we obtained about 224 GFlop/s with the MKL's Cholesky factorization (on Intel Xeon Phi 5110P), and our preliminary experiment with XKaapi recursive code reached 194 GFlop/s.

Pennycook et al. [7] reported the use of the SIMD instruction sets of Intel Xeon processors and Intel Xeon Phi coprocessors to accelerate molecular dynamics (MD) simulations. The use of an Intel Xeon Phi coprocessor with SIMD was $5.2x$ faster than scalar execution.

The Offload compiler and runtime of Intel Xeon Phi software [8] offers language pragma directives `#pragma offload` to offload computations from a host processor to a coprocessor. The Offload compiler hides parallelism from programmer and manages data transfers and code execution, which rely on its runtime. Cramer et al. [6] evaluated the overhead of the offload directives and compared with a 16-socket 128-core system composed of Intel Xeon X7550 processors. It concluded that the overhead was low compared to large multi-core systems.

The authors in [4] reported comparisons on a dense linear QR factorization in CilkPlus and OpenMP on a dual Intel Knights Ferry coprocessors connected through PCIe to a dual 6-cores Intel Westmere X5680. Their conclusions were similar to ours on Cholesky factorization: OpenMP and CilkPlus showed similar performance. They also observed some degradation of CilkPlus when the number of threads was higher than the hardware cores. We did not observe this behavior on Cholesky factorization but the runtime or the hardware used are different.

MAGMA [31] implements static scheduling for linear algebra algorithms on heterogeneous systems composed of GPUs. Recently it has included some hybrid methods that uses a Sandy Bridge processor and an Intel Xeon Phi coprocessor to performance operations [32]. Heinecke et al. [9] designed three versions of the Linpack benchmark for nodes with Intel Xeon Phi coprocessors: native, single-node hybrid, and multi-node hybrid version.

To the best of our knowledge, no other paper or technical report made experiments on non-numerical benchmarks such as Fibonacci or NQueens benchmarks on the Intel Xeon Phi, which behaves similarly to one SandyBridge socket with 8 physical cores on our experiments.

V. Concluding Remarks

In this paper, we presented the performance results of the XKaapi data-flow programming model on the Intel Xeon Phi coprocessor in native execution. We designed and evaluated three benchmarks (Fibonacci, NQueens, and Cholesky) and compared to OpenMP and CilkPlus, native Xeon Phi parallel programming environments provided by Intel. We conducted experiments with a 60-core Intel Xeon Phi and an Intel Sandy Bridge with 32 physical cores and 64 hardware threads.

Results on non-standard benchmarks (Fibonacci and NQueens) showed that one Intel Xeon Phi chip with 60 cores can be a competitive architecture almost outperforming one socket of the complex superscalar and out-of-order 8 cores Intel Xeon E5-4620 processor, if, and only if, (a) the application exhibits enough parallelism, even irregular and dynamic, for the 240 available threads; (b) the runtime is able to schedule fine grain tasks with low overhead.

Our Cholesky benchmark showed that using finer synchronizations between tasks (data flow dependencies) is more efficient than only relying on the fork-join model as OpenMP-3.1 and CilkPlus. Fine grain parallelism may be increasingly essential as the number of cores grow faster than memory capabilities. Although the 60 cores of our Intel Xeon Phi shared only 8 GB of memory, the design of parallel applications under these constraints require finer tasks that may hopefully take advantage of finer data flow dependencies for better performance.

This paper presented preliminary and promising performance results of XKaapi on the Intel Xeon Phi coprocessor. Future works include extended evaluations on different benchmarks, as well as energy consumption measures. We will also try to take into account specific details of the memory organization to reduce data transfers, using locality heuristics (HEFT scheduler or work stealing with affinity considerations).

Finally, pursuing our previous works [24], [33], we will focus on providing a highly optimized OpenMP-4.0 runtime support for Intel Xeon Phi; and we will also study the

performance of PCIe interconnected multi-Intel Xeon Phi architectures following our research on multi-GPUs [2]

ACKNOWLEDGMENTS

This work has been partially supported by the ANR-11-BS02-013 HPAC Project, the ANR 09-COSI-011-05 Project Repdyn, CAPES/Brazil, CNPq/Brazil, and FAPERGS/Brazil.

REFERENCES

[1] J. V. F. Lima, T. Gautier, N. Maillard, and V. Danjean, "Exploiting Concurrent GPU Operations for Efficient Work Stealing on Multi-GPUs," in *Proc. of the 24th SBAC-PAD*. New York, USA: IEEE, 2012, pp. 75–82.

[2] T. Gautier, J. V. F. Lima, N. Maillard, and B. Raffin, "XKaapi: A Runtime System for Data-Flow Task Programming on Heterogeneous Architectures," in *Proc. of the 27th IEEE IPDPS*, may 2013.

[3] S. Lee and J. S. Vetter, "Early evaluation of directive-based gpu programming models for productive exascale computing," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 23:1–23:11.

[4] J. Eisenlohr, D. E. Hudak, K. Tomko, and T. C. Prince, "Dense Linear Algebra Factorization in OpenMP and Cilk Plus on Intel's MIC Architecture: Development Experiences and Performance Analysis," Austin, Texas, US, april.

[5] V. B. J. Labarta, "Prototype programming environment in Booster Node, delivrable D5.1, EU DEEP project Dynamical Exascale Entry Platform," Tech. Rep., 02, fP7-ICT-2011-7.

[6] T. Cramer, D. Schmidl, M. Klemm, and D. an Mey, "OpenMP Programming on Intel Xeon Phi Coprocessors: An Early Performance Comparison," in *Proceedings of the Many-core Applications Research Community (MARC) Symposium at RWTH Aachen University*, November 2012, pp. 38–44.

[7] S. J. Pennycook, C. J. Hughes, M. Smelyanskiy, and S. A. Jarvis, "Exploring simd for molecular dynamics, using intel xeon processors and intel xeon phi coprocessors," in *Proc. of the 27th IEEE IPDPS*, 2013.

[8] C. J. Newburn, S. Dmitriev, R. Narayanaswamy, J. Wiegert, R. Murty, F. Chinchilla, R. Deodhar, and R. McGuire, "Offload Compiler Runtime for the Intel Xeon Phi Coprocessor," in *Proc. of the 27th IEEE IPDPS Workshops and PhD Forum*, 2013.

[9] A. Heinecke, K. Vaidyanathan, M. Smelyanskiy, A. Kobotov, R. Dubtsov, G. Henry, G. Chrysos, A. G Shet, and P. Dubey, "Design and Implementation of the Linpack Benchmark for Single and Multi-Node Systems Based on Intel(R) Xeon Phi(TM) Coprocessor," in *Proc. of the 27th IEEE IPDPS*. Boston, USA: IEEE, May 2013.

[10] E. A. Lee, "The problem with threads," *Computer*, vol. 39, pp. 33–42, 2006.

[11] R. D. Blumofe and C. E. Leiserson, "Space-efficient scheduling of multithreaded computations," *SIAM J. Comput.*, vol. 27, pp. 202–229, 1998.

[12] OpenMP Architecture Review Board, "http://www.openmp.org," 1997-2013.

[13] I. of Electrical and I. Electronic Engineers, "Information Technology — Portable Operating Systems Interface (POSIX) — Part: System Application Program Interface (API) — Amendment 2: Threads Extension [C Language]," IEEE, New York, NY, IEEE Standard 1003.1c–1995, 1995.

[14] A. Robison, M. Voss, and A. Kukanov, "Optimization via reflection on work stealing in TBB," in *Proc. of the IEEE IPDPS*, 2008, pp. 1–8.

[15] J. Kurzak, H. Ltaief, J. Dongarra, and R. M. Badia, "Scheduling dense linear algebra operations on multicore processors," *Concurr. Comput. : Pract. Exper.*, vol. 22, pp. 15–44, 2010.

[16] A. Duran, R. Ferrer, E. Ayguadé, R. M. Badia, and J. Labarta, "A proposal to extend the openmp tasking model with dependent tasks," *Int. J. Parallel Program.*, vol. 37, pp. 292–305, June 2009.

[17] E. Hermann, B. Raffin, F. c. Faure, T. Gautier, and J. Allard, "Multi-GPU and Multi-CPU Parallelization for Interactive Physics Simulations," in *Proc. of Euro-Par*, vol. 6272. Springer, 2010, pp. 235–246.

[18] J. Bueno, J. Planas, A. Duran, R. M. Badia, X. Martorell, E. Ayguadé, and J. Labarta, "Productive Programming of GPU Clusters with OmpSs," in *Proc. of the IEEE IPDPS*, 2012.

[19] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011.

[20] T. Gautier, X. Besseron, and L. Pigeon, "KAAPI: A thread scheduling runtime system for data flow computations on cluster of multiprocessors," in *Proc. of PASCO'07*. London, Canada: ACM, 2007.

[21] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multithreaded language," in *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, ser. PLDI '98. New York, NY, USA: ACM, 1998, pp. 212–223.

[22] E. Ayguadé, R. Badia, F. Igual, J. Labarta, R. Mayo, and E. Quintana-Ortí, "An Extension of the StarSs Programming Model for Platforms with Multiple GPUs," in *Proc. of Euro-Par*, vol. 5704. Springer, 2009, pp. 851–862.

[23] F. Galilée, J.-L. Roch, G. G. H. Cavalheiro, and M. Doreille, "Athapascan-1: On-line building data flow graph in a parallel language," in *Proc. of PACT'98*. Washington, DC, USA: IEEE Computer Society, 1998, pp. 88–95.

[24] F. Broquedis, T. Gautier, and V. Danjean, "Libkomp, an efficient openmp runtime system for both fork-join and data flow paradigms," in *Proceedings of the 8th international conference on OpenMP in a Heterogeneous World*, ser. IWOMP'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 102–115.

[25] A. YarKhan, J. Kurzak, and J. Dongarra, "QUARK Users' Guide: QUeueing And Runtime for Kernels," University of Tennessee, Tech. Rep. ICL-UT-11-02, 2011.

[26] M. Tchiboukdjian, N. Gast, and D. Trystram, "Decentralized list scheduling," *Annals of Operations Research*, pp. 1–23, 2012.

[27] Takaken, "Source code for n queens problem."

[28] E. Agullo, C. Augonnet, J. Dongarra, H. Ltaief, R. Namyst, S. Thibault, and S. Tomov, "Faster, Cheaper, Better – a Hybridization Methodology to Develop Linear Algebra Software for GPUs," in *GPU Computing Gems*, W. mei W. Hwu, Ed. Morgan Kaufmann, Sep 2010, vol. 2.

[29] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, "A class of parallel tiled linear algebra algorithms for multicore architectures," *Parallel Computing*, vol. 35, no. 1, pp. 38–53, 2009.

[30] B. Dumitrescu, M. Doreille, J.-L. Roch, and D. Trystram, "Two-dimensional block partitionings for the parallel sparse cholesky factorization," *Numerical Algorithms*, vol. 16, pp. 17–38, 1997.

[31] S. Tomov, J. Dongarra, and M. Baboulin, "Towards dense linear algebra for hybrid GPU accelerated manycore systems," *Parallel Computing*, vol. 36, no. 5-6, pp. 232–240, 2010.

[32] J. Dongarra, M. Gates, A. Haidar, Y. Jia, K. Kabir, P. Luszczek, and S. Tomov, "MAGMA MIC 1.0: Linear Algebra Library for Intel Xeon Phi Coprocessors," 2013.

[33] M. Durand, F. Broquedis, T. Gautier, and B. Raffin, "An Efficient OpenMP Loop Scheduler for Irregular Applications on Large-Scale NUMA Machines," in *IWOMP*, A. R. et al., Ed., no. 8122. Berlin, Heidelberg: Springer-Verlag, sep 2013, pp. 141–155.