Performance Evaluation of Intel Xeon Phi Coprocessor using XKaapi

João V. F. Lima, Nicolas Maillard
Instituto de Informática – UFRGS
Grupo de Processamento Paralelo e Distribuído (GPPD)
{*jvlima, nicolas*}*@inf.ufrgs.br*

François Broquedis, Thierry Gautier, Bruno Raffin
Laboratoire d'Informatique de Grenoble
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. Our benchmark suite is composed of two computing kernels: a Fibonacci computation that allows to study the overhead and the scalability of the runtime system, and a NQueens application generating irregular and dynamic tasks. 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.

1. 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.

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, 6, 11, 13, 15, 16], trying to position the Intel Xeon Phi as a good candidate for executing efficient high-performance parallel applications.

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.

Section 2 overviews the XKaapi's parallel programming model and the difficulties encountered during the port of XKaapi to the Intel Xeon Phi coprocessor. Section 3 reports experimental evaluations compared to OpenMP and CilkPlus, the native parallel programming environments proposed by Intel to develop Xeon Phi's applications. Section 4 concludes the paper and suggests future works.

2. XKaapi on Intel Xeon Phi

2.1. Overview of XKaapi

The XKaapi[1] task model [9], as in Cilk [7], Intel TBB [17], OpenMP-3.0 or StarSs [1, 3], enables non-blocking task creation: the caller creates the task and proceeds with the program execution. The semantic remains sequential such as XKaapi's predecessors Athapascan [8] and KAAPI [9], which was specialized for multi-CPU/multi-GPU iterative applications [12].

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. [14] and Gautier et al. [10].

1 `http://kaapi.gforge.inria.fr`

2.2. 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, 9, 10] have demonstrated the good scalability of XKaapi even at fine grain, so we decided to keep it unmodified for this first evaluation.

3. Experiments

This section presents the experimental results of the XKaapi runtime system on an Intel Xeon Phi coprocessor. All times reported in this section are average of more than 30 executions with a warm-up phase of 2 runs.

3.1. 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 software environment used on the Intel Xeon Phi was the following: firmware version 1.14.4616 that 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 2 of this paper. XKaapi applications were compiled with the same Intel compilers used to compile OpenMP and CilkPlus applications.

3.2. 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.

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

3.2.1. Overall Analysis Table 1 reports experimental results on the Intel Xeon Phi. The results obtained by this benchmark shows XKaapi has the lowest overhead among the three tested environments. As highlighted in [2, 10], XKaapi intrinsic overheads due to the computation of the data-flow dependencies between tasks are reported on steal operations. If the number of steal operations is very small compared to the number of created tasks, such as Fibonacci [7], data-flow related overheads do not impact XKaapi's performance obtained.

3.2.2. 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 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.

3.2.3. OpenMP Performance Issues On the contrary, Intel OpenMP exhibited poor performance for this micro-benchmark with fine grain recursive tasks. 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 poor performance on fine grain task-based programs for the GNU/GCC libGOMP runtime system [2], and in a less significant way for Intel OpenMP as well. 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. [2].

3.3. NQueens

The NQueens benchmark is based on the Takaken [18] optimized sequential code to compute the number of solutions for the NQueens problems. It has been parallelized using XKaapi since 2007 [9] 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.

3.3.1. 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 [18]. Each configuration is explored by an independent task. On final recursion, possible solutions are cumulated 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 cumulated. 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.

3.3.2. Accumulation in XKaapi The XKaapi version creates tasks with access to the global variable declared as "*cumulative write access*" [8, 9], which allows to accumulate arbitrary data with an user defined associative or cumulative 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 cumulated to the victim thread's data. At the end, the global variable contains the final cumulated result. This mechanism enables the XKaapi runtime to reduce the required synchronizations compared to OpenMP and CilkPlus.

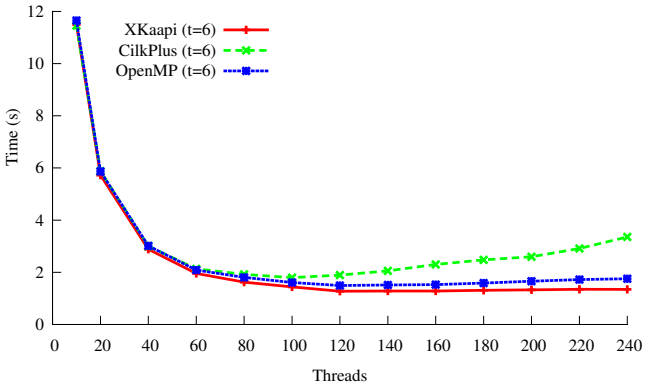

Figure 1. Scalability of the NQueens benchmark (N=17) for XKaapi, CilkPlus and OpenMP. Speedups were computed against the sequential execution time of $114.95s$.

3.3.3. Scalability Figure 1 reports the speedup $S = T_p/T_{seq}$ for NQueens ($N = 17$) on Intel Xeon Phi. 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.

4. 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 two benchmarks (Fibonacci and NQueens) 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 coprocessor.

Our results showed that one Intel Xeon Phi chip with 60 cores can be a competitive architecture, 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.

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

Finally, pursuing our previous works [2, 5], 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 [10]

References

[1] 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*, volume 5704, pages 851–862. Springer, 2009.

[2] 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*, IWOMP'12, pages 102–115, Berlin, Heidelberg, 2012. Springer-Verlag.

[3] 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.

[4] 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*, pages 38–44, November 2012.

[5] M. Durand, F. Broquedis, T. Gautier, and B. Raffin. An Efficient OpenMP Loop Scheduler for Irregular Applications on Large-Scale NUMA Machines. In A. R. et al., editor, *IWOMP*, number 8122, pages 141–155, Berlin, Heidelberg, sep 2013. Springer-Verlag.

[6] 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, april.

[7] 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*, PLDI '98, pages 212–223, New York, NY, USA, 1998. ACM.

[8] 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*, pages 88–95, Washington, DC, USA, 1998. IEEE Computer Society.

[9] T. Gautier, X. Besseron, and L. Pigeon. KAAPI: A thread scheduling runtime system for data flow computations on cluster of multi-processors. In *Proc. of PASCO'07*, London, Canada, 2007. ACM.

[10] 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.

[11] 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, May 2013. IEEE.

[12] 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*, volume 6272, pages 235–246. Springer, 2010.

[13] V. B. J. Labarta. Prototype programming environment in booster node, delivrable d5.1, eu deep project dynamical exascale entry platform. Technical report, 02. FP7-ICT-2011-7.

[14] 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*, pages 75–82, New York, USA, 2012. IEEE.

[15] 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.

[16] 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.

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

[18] Takaken. Source code for n queens problem.