Achieving high-performance with a sparse direct solver on Intel KNL

Emmanuel Agullo, Alfredo Buttari, Mikko Byckling, Abdou Guermouche, Ian Masliah

▶ **To cite this version:**

HAL Id: hal-01473475

https://inria.hal.science/hal-01473475

Submitted on 21 Feb 2017

Achieving high-performance with a sparse direct solver on Intel KNL

Emmanuel Agullo, Alfredo Buttari, Mikko Byckling, Abdou Guermouche, Ian Masliah

Achieving high-performance with a sparse direct solver on Intel KNL

Emmanuel Agullo[*], Alfredo Buttari[†], Mikko Byckling[‡], Abdou
Guermouche[§], Ian Masliah[¶]

Project-Teams HiePACS and Roma

Abstract: The need for energy-efficient high-end systems has led hardware vendors to design new types of chips for general purpose computing. However, designing or porting a code tailored for these new types of processing units is often considered as a major hurdle for their broad adoption. In this paper, we consider a modern Intel Xeon Phi processor, namely the Intel Knights Landing (KNL) and a numerical code initially designed for a classical multi-core system. More precisely, we consider the `qr_mumps` scientific library implementing a sparse direct method on top of the StarPU runtime system. We show that with a portable programming model (task-based programming), a good software support (a robust runtime system coupled with an efficient scheduler) and some well defined hardware and software settings, we are able to transparently run the exact same numerical code. This code not only achieves very high performance (up to 1 TFlop/s) on the KNL but also significantly outperforms a modern Intel Xeon multi-core processor both in terms of time to solution and energy efficiency up to a factor of 2.0.

Key-words: manycore parallelism, Intel KNL, portability, high-performance computing, energy efficiency, sparse direct solver, runtime system

---

[*] Inria - LaBRI, Bordeaux
[†] CNRS - IRIT, Toulouse
[‡] Intel Corporation
[§] Université de Bordeaux - LaBRI, Bordeaux
[¶] Inria, Bordeaux

Solveur creux direct haute-performance sur Intel KNL

Résumé : Le besoin de systèmes haut-de-gamme efficaces d'un point de vue énergétique a poussé les fabricants à mettre au point de nouvelles puces. Cependant, mettre au point ou porter un code adapté pour ces nouveaux types d'unités de calcul est souvent considéré comme une limitation majeure à leur large adoption. Dans ce papier, nous considérons un processeur Intel Xeon Phi moderne, précisément l'Intel Knights Landing (KNL), et un code numérique initialement mis au point pour les machines multi-cœurs. Plus précisément, nous considérons la bibliothèque scientifique `qr_mumps` implémentant une méthode directe creuse au dessus du moteur d'exécution StarPU. Nous montrons qu'avec un modèle de programmation portable (programmation à base de tâches), un bon support logiciel (un moteur d'exécution à base de tâches robuste couplé avec un ordonnanceur efficace) et des paramètres matériel et système bien déterminés, nous sommes capables de tourner exactement le même code numérique de manière transparente. Non seulement ce code atteint une très haute performance (jusqu'à 1 TFlop/s) sur le KNL mais de surcroît il surpasse significativement un processeur multi-cœur standard à la fois en termes de temps de résolution que d'efficacité énergétique jusqu'à un facteur 2.0.

Mots-clés : parallélisme, Intel KNL, portabilité, calcul haute-performance, efficacité énergétique, solveur creux direct, moteur d'exécution

1 Introduction

Modern multicore and manycore architectures are now part of the high-end and mainstream computing scene and can offer impressive performance for many applications. This architecture trend has been driven by the need to reduce power consumption, increase processor utilization, and deal with the memory-processor speed gap. However, the complexity of these new architectures has created several programming challenges, and achieving performance on these systems is often difficult work. On the other hand, many research efforts have been devoted to propose new programming paradigms that provide programmers with portable techniques and tools to exploit such complex hardware. One major trend consists in adding support for task-based programming which allows to design a numerical library as a set of inter-dependent tasks. The large panel of mature task-based runtime systems (Legion [7], PaRSEC [8], StarPU [5] or StarSs [6] to quote a few) has made possible the design of a wide range of new robust task-based, high-performance, scientific libraries. In return, this high performance computing (HPC) ecosystem has motivated the introduction of dependent tasks in the revision 4.0 of the OpenMP standard (with the introduction of the "depend" clause), providing task-based programming to a broader audience than the sole HPC community.

The dense linear algebra community has adopted this modular approach over the past few years [21, 4, 9] and delivered production-quality software following these principles for exploiting multicore and heterogeneous architectures. For example, the `MAGMA` and `Chameleon`[1] libraries [4], provide Linear Algebra algorithms over heterogeneous hardware and can optionally use the StarPU runtime system to perform dynamic scheduling between CPUs and accelerators, illustrating the trend toward delegating scheduling to the underlying runtime system. Moreover, such libraries exhibit state-of-the-art performance, resulting from heavy tuning and strong optimization efforts. This approach was later applied to more irregular algorithms such as sparse direct methods [18, 17, 3].

While most of this research and development effort has been devoted to program multicore and heterogeneous systems, much fewer studies have been conducted on manycore systems. These types of processors have indeed been more recently introduced and aim at bridging the gap between the energy-efficiency ensured by accelerators and the standard usage of an x86 system. Yet, designing or porting a code tailored for these new types of processing units is often considered as a major hurdle for their broad adoption.

In this paper, we consider the second generation of Intel Xeon Phi processors, namely the Knights Landing (KNL), targeted for HPC and supercomputing. Some studies have been carried out recently to study and optimize the performance of existing algorithms/software packages on this platform. For example Malas *et al.* [19], Haidar *et al.* [16] and Rosales *et al.* [23] assess the interest of the KNL system for different classes of applications through experiments showing that the platform is able to outperform classical computing nodes based

---

[1]`https://project.inria.fr/chameleon/`

on general purpose processors. In [13], the authors present a roofline model approach to evaluate and optimize several classes of applications for the KNL device. They provide case-studies and guidelines to improve the behavior of the considered applications. To the best of our knowledge, no study has been carried so far focusing on the behavior of direct methods for sparse linear systems on the KNL platform. Here we consider the `qr_mumps` solver [3] implementing a sparse direct method on top of the StarPU runtime system. `qr_mumps` was originally designed for classical multi-core systems and has been tailored to deliver a maximum level of concurrency between tasks by implementing state-of-the-art tile algorithms [11, 15] (sometimes also referenced as communication-avoiding algorithms). The objective of the present study is to assess the effort required to port such a code on a KNL processor while achieving high performance. Assuming that the numerical code already benefits from a careful design for exploiting the level of concurrency provided by the hardware platform, we can solely focus on the impact of the system and hardware settings on the overall performance. We finally assess whether the KNL allows for an improved energy efficiency in comparison with a standard multicore Intel Xeon processor.

The contributions of this paper are the following. First, we show that task-based programming is an effective paradigm for exploiting the potential of modern manycore processors. Second, if we confirm what previous studies have shown about Intel MIC processors being subtle to tune, our experimental study also emphasizes the impact of each individual important hardware and software setting. We hope this study can be beneficial to the HPC community for porting their own code on such platforms. The third contribution is that, overall, we deliver a sparse direct solver that can attain up to 1 TFlop/s on a single manycore processor while achieving a high energy efficiency with respect to its execution on a regular Intel multicore processor.

The rest of the paper is organized as follows. Section 2 presents the numerical method implemented in the `qr_mumps` solver as well as the Intel KNL processor. Section 3 presents the important hardware and software parameters to set up when porting a numerical code on KNL. Section 4 presents the experimental results and Section 5 concludes this study.

2   Background

2.1   Task-based, parallel multifrontal QR factorization

The multifrontal method, introduced by Duff *et al.* [14] as a method for the factorization of sparse, symmetric linear systems, can be adapted to the QR factorization of a sparse matrix thanks to the fact that the R factor of a matrix A and the Cholesky factor of the normal equation matrix $A^T A$ share the same structure. The multifrontal QR factorization is based on the concept of *elimination tree* which expresses the dependencies between the unknowns of A and, thus, the order in which they have to be eliminated. Each vertex f of the tree is associated with k_f unknowns of A; the unknowns associated with a node can

only be eliminated after those associated with the children of the node. As a result, the multifrontal QR factorization consists in a **topological order** (i.e., bottom-up, typically a post-order) traversal of the elimination tree. When a node is visited the corresponding k_f rows of A and coefficients resulting from the processing of child nodes are assembled together into a dense matrix, called *frontal matrix* or, simply, *front*. Once the frontal matrix is assembled, the k_f unknowns are eliminated through a **complete dense QR factorization** of the front. This produces k_f rows of the global R factor, a number of Householder reflectors that implicitly represent the global Q factor and a *contribution block*, i.e. the set of coefficients which will be assembled into the parent front together with the contribution blocks from all the sibling fronts.

The multifrontal method provides two distinct sources of concurrency: **tree** and **node parallelism**. The first one stems from the fact that fronts in different branches are independent and can thus be processed concurrently; the second one from the fact that, if a front is large enough, multiple processes can be used to assemble and factorize it. These two sources of concurrency are, moreover, complementary because at the bottom of the elimination tree frontal matrices are abundant but of small size, whereas at the top only a few fronts are available but of relatively large size. We refer to [10] and the references therein for further details on the multifrontal QR method.
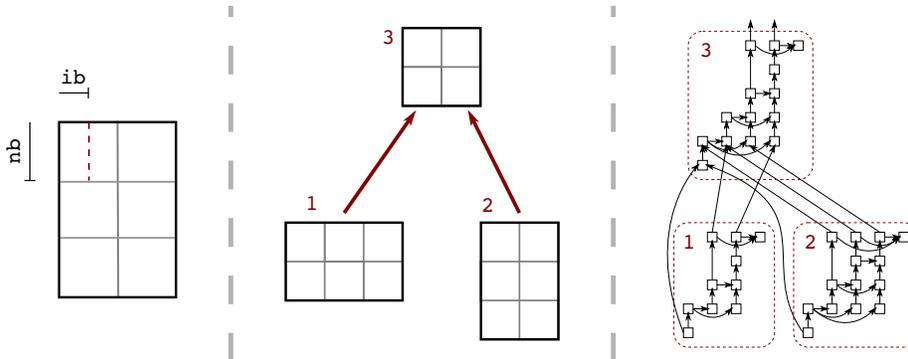

Figure 1: Front partitioning into tiles (*left*). Elimination tree (*center*). DAG of tasks (*right*).

The baseline of this work is the implementation described in [3] and currently available in the `qr_mumps` software, V 2.0. In this approach, frontal matrices are partitioned into square blocks (also referred to as *tiles*) of size `nb` (typically $nb = O(10^2)$); operations on tiles use an internal blocking size *ib* in order to take advantage of the efficiency of BLAS-3 routines. This is shown in Figure 1 (*left*). Because frontal matrices are commonly (strongly) over-determined (i.e., with many more rows than columns) their factorization is achieved through a 2D, Communication Avoiding dense QR algorithm [11, 15]. If a task is defined as an operation on one or few tiles, the whole elimination tree can be transformed into a Directed Acyclic Graph (DAG) where nodes represent tasks and

edges the dependencies between them and thus the order in which they have to be executed. This is illustrated in Figure 1 (*center*) and (*right*). This DAG expresses a high amount of concurrency because it allows for seamlessly handling both tree and node parallelism, allows for pipelining assembly and factorization operations and has the further advantage that it permits to pipeline the processing of a front with those of its children, which provides an additional source of concurrency. It must be noted that in common use cases (such as the problems used in our experimental results in Section 4) the elimination tree can have tens of thousands of nodes and frontal matrices can have up to hundreds of thousands of rows or columns; as a result the DAG included an extremely high number of tasks with complex dependencies.

Modern tools exist for achieving this task based parallelization which are commonly referred to as *task-based runtime systems* or, simply, *runtime systems*. These tools provide a portable programming model which makes abstraction of the underlying architecture and which allows the programmer to express his workload in the form of a DAG of tasks using different paradigms such as the *Sequential Task Flow* (STF) or the *Parametrized Tasks Graph* (PTG); the runtime system takes care of executing the tasks of the DAG on the computing platforms respecting the dependencies and making use of the available processing units (including accelerators such as GPU devices). Modern runtime systems provide a wide set of convenient features which greatly simplify the implementation of complex algorithms on modern computing platforms; these features include the automatic detection of the dependency among tasks or the handling of data (including the transfers of data from different memory modules). For the implementation of the `qr_mumps` package, we have chosen to rely on the StarPU [5] runtime system which uses the STF parallel programming paradigm; in this programming approach, the parallel code resembles the sequential one except that, instead of directly executing operations on data, tasks are submitted to the runtime system along with details on how each task accesses the data. The runtime system can thus automatically build the DAG through a data dependency analysis, and deploy it on the available processing units. We refer the reader to[3] for further details on the approach used in the `qr_mumps` package.

Our previous work [3, 1, 10, 2] demonstrates that this task-based approach delivers much higher performance and scalability with respect to traditional approaches where tree and node parallelism are exploited separately. Moreover, the STF programming model and the modern runtime systems that rely on it, allow for implementing this approach in an efficient, robust and portable way.

2.2   KNL description

The Intel KNL architecture was launched during Summer 2016. It is a manycore system composed by numerous computing CPU cores (at least 64 cores). Each core is an Intel Airmont (Atom) core having four threads each. The cores are organized into tiles each containing two cores sharing a 1MB L2 cache. The tiles are connected to each other with a mesh. From the memory point of view,

in addition to the traditional DDR4 memory, the device is equipped with the Multi-Channel DRAM (MCDRAM) memory which is a high bandwidth ($\sim$ 4x more than DDR4), but low capacity (16GB) memory. This new type of memory can be configured as either a separate NUMA node (*flat memory mode*), in which case the placement of data has to be explicitly handled, as an additional level of cache (*cache memory mode*), or a mixture of the two (*hybrid memory mode*).

Aside from the MCDRAM management, it is possible on the KNL to use several cache clustering modes, which will be referred to as *clustering modes*. The mode selection have a strong impact on the performance of the caches. From the point of view the programmer, these clustering modes impact the memory hierarchy. For instance, in the SNC-2 or SNC-4 clustering modes, the device can be seen as a NUMA system while with the quadrant or hemisphere, it can be considered as a SMP platform. Depending on the chosen clustering mode it us up to the programmer to ensure locality of memory references as well as the placement of data in memory, which may significantly impact the efficiency of data accesses (we refer the reader to [25] for further details).

3   Setting up KNL parameters

Due to the algorithmic properties of sparse direct methods, a large amount of memory may be necessary to solve problems from real-life applications. The peak of memory consumption may be of the order of tens or hundreds of GigaBytes (see our test problems in Section 4.2.1). Thus, most of the tuning parameters we consider are related to memory management features provided on the Intel KNL platform.

3.0.1   Hardware settings

In this work we assume that the memory needed for each experiment exceeds the size of the MCDRAM. As mentioned in Section 2.2, we will use the cache mode for MCDRAM to consider the latter as an additional level of shared cache making the use of MCDRAM transparent to the application; this is coherent with the choice made in previous studies such as [19, 23]. The NUMA nature of the SNC-2 and SNC-4 demands a careful handling of the placement of data and tasks; this, however, imposes constraints on the scheduling policy which ultimately results in a loss of performance when the DAG has complex dependencies, like in our case. For this reason, we have chosen to use the quadrant clustering configuration.

3.0.2   Software settings

KNL is a x86 processor and runs a standard Linux operating system such as RHEL 7.2/7.3 or SUSE, which abstract the hardware memory to the applications via a *virtual memory* scheme. We call *Paging* a software mechanism to manage virtual memory in the Linux kernel, with pages of size 4 kB by default. We call *Translation Lookaside Buffer* (TLB) a hardware cache memory used

by the processors' *Memory Management Unit* (MMU) to accelerate the *Paging* process.

KNL has hardware support for using 4kB, 2MB and 1GB pages and Larger TLBs than conventional processors. The advantages of using *hugepages* is twofold. First, it reduces the number of entries in the TLB, which makes a TLB miss faster to process. Secondly, it reduces the number of TLB misses as a *hugepage* can map a much larger amount of virtual memory (512 times more between 4kB and 2 MB pages). Recent studies [24, 20] have shown the benefits of using a larger page size, or *hugepages* in high performance computing applications.

In the following experiments, we will use two different approaches to use larger virtual memory pages: 1) *Transparent Huge Pages* (THP) provided by the Linux operating system to automatically promoting large allocations to use larger than 4kB page size, 2) Explicit *hugepages* allocations. Finally, we will evaluate the impact of different allocators on behavior of `qr_mumps`. We will rely on either the regular Linux allocator, or an allocator provided by Intel Threading Building Blocks (TBB), see [22]. The aim of the latter is to have a scalable approach for allocating memory from concurrent threads.

In Table 1, we give a description of the test machines and summarize the different parameters described throughout Section 3 with the best settings found.

Test machines :	
System 1 (KNL64)	Intel(R) Xeon Phi(TM) CPU 7210 - 64 cores @1.3 GHz
System 2 (KNL68)	Intel(R) Xeon Phi(TM) CPU 7250 - 68 cores @1.4 GHz
System 3 (BDW)	Intel(R) Xeon(R) E5 2697v5 - 2 sockets, 18 cores @2.3 Ghz
KNL Hardware settings :	
Clustering mode	quadrant
MCDRAM mode	cache
Operating system/memory settings :	
Operating system	RHEL 7.2
Memory allocator	TBB : scalable allocator, Explicit Hugepages (8000)
THP	always active
Hugepage size	2MB
Libraries settings :	
Compiler	Intel Parallel Studio 2017, Update 1
BLAS library	Intel Math Kernel Library, 2017 Update 1
`qr_mumps`	2.0
StarPU/scheduler	trunk (rev. 19630)/ws

Table 1: Experimental hardware and software configuration (Best settings)

4 Experimental results

Finding optimal parameter values is a complex problem as the resulting *configuration space* from software and hardware settings is usually very large. Furthermore, these values can vary greatly depending on the targeted architecture (number of cores, cache sizes, bandwidth...) and the elimination tree resulting from the matrix (Section 2.1). As seen in Section 3, through an advanced understanding of the hardware in the context of the `qr_mumps` software we can decide of the KNL configuration to prune the *configuration space*.

First, in Section 4.1, we will use the configuration described in Table 1 using the `KNL64` platform to focus on tuning `qr_mumps` parameters. Then, we will fix the `qr_mumps` parameters we have obtained to detail the influence of the OS memory settings parameters from our previous analysis done in Section 3. Once we have explained thoroughly how to tune both the `qr_mumps` software and the OS for the `KNL64` machine, we will compare the results with those obtained on the `KNL68` and `BDW` systems and analyze their energy efficiency.

4.1 Tuning `qr_mumps` for the KNL

As our experimental analysis focuses on relatively large problems, most of the execution time is spent in the dense QR factorization of frontal matrices, which consists mostly in BLAS-3 operations. Therefore, we first measure and analyze the effect of block size parameters on the performance of the dense QR factorization.
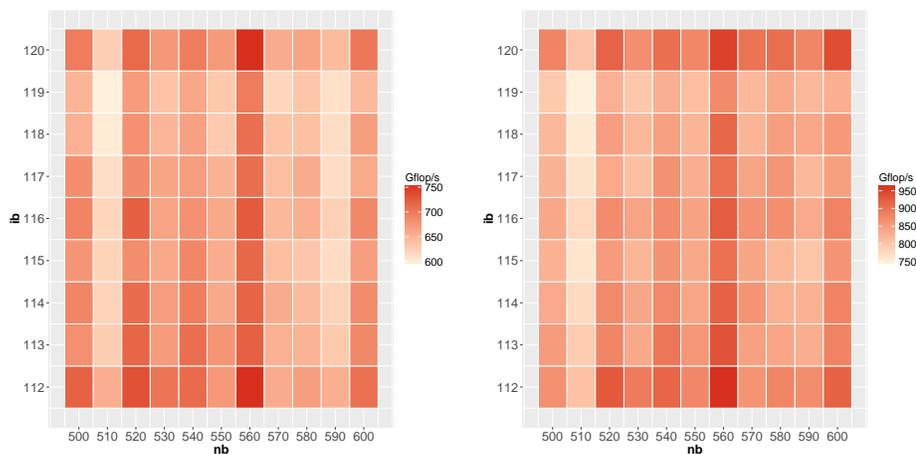

Figure 2: Impact of block size for fronts (`KNL64`) of size 16384×8192 (*left*) and 20480×16384 (*right*).

Figure 2 shows the performance of the dense QR factorization on two different matrices sizes (both over-determined, which is the most common case for

fronts) depending on the block size nb which represents the granularity of the tasks and the internal block size ib, which corresponds to the block size used internally by each task to carry out the computations. Here, we have restricted the values shown for the tile sizes to those around the optimal size for readability purposes. On the KNL larger tile sizes than what is used for conventional multicore systems are necessary due to the wide vector sizes (512-bit SIMD vectors), the limited bandwidth for the number of cores and the important L3 cache resulting from the MCDRAM cache-mode configuration.

Figure 2 outlines the difficulty of tuning block sizes as results can vary as much as 200 Gflop/s by changing nb by 10 or ib by 1. To better understand these results, it is important to note that higher performance arises by choosing values of ib that are multiple of the size of the SIMD vector(112, 120 in Figure 2). As our block sizes are still relatively small, having a block size multiple of the SIMD length (8 for double-precision) greatly improves the performance of the BLAS-3 operations and their stability. This is more-so true with the size of an 512-SIMD vector being equal to the size of an L1 cache line on the KNL.

4.2   Performance analysis for the `qr_mumps` software

4.2.1   Experimental settings

To evaluate the behavior of the `qr_mumps` solver we selected a set of five matrices from the UF Sparse Matrix Collection [12]. These matrices are listed in Table 2 along with their size, number of nonzeros, operation count, memory peak obtained when MeTiS, fill-reducing column permutation is applied and the best performance obtained.

Mat. name	m	n	nz	op. count (Gflop)	peak mem (GB)	Best Perf. (Gflop/s)
spal_004	10203	321696	46168124	27059	23.3	562.21
TF17	38132	48630	586218	38209	12.8	837.55
n4c6-b6	104115	51813	728805	97304	35.6	1001.79
lp_nug30	52260	379350	1567800	171051	83.4	970.23
TF18	95368	123867	1597545	194472	78.1	1018.61

Table 2: Matrix set and relative factorization cost obtained with MeTiS ordering.

4.2.2   Performance analysis of KNL configurations

The results presented in Figure 3 are based on the block sizes obtained from the tuning done on front matrices ($nb = 560$, $ib = 112$).

As explained in Section 3, understanding how to setup the hardware and the OS is key to make the best out of the KNL. Results in Figure 3 shows a difference in performance from simply optimizing `qr_mumps` to making the best out of the

memory system of up to 19%. The results displayed in Figure 3 show the speed of the multifrontal factorization on the `KNL64` system which exceeds 1 Tflop/s on the TF18 and n4c6-b6. We can see that most matrices benefit greatly from *hugepages*. It is also important to note that optimal performance is obtained while using a TBB allocator for *hugepages* with THP active. While *hugepages* allocated with TBB only concerns memory allocated by `qr_mumps`, THP are active system-wide which further improves the TLBs efficiency (see Section 3). The reason for spal_004 providing a relatively low number of Gflop/s is due to the lack of parallelism in the elimination tree.

4.2.3 Performance comparison and energy efficiency

In this section, we compare the performance of the three platforms described in Table 3 (`KNL64`, `KNL68`, `BDW`) and the energy consumption of the `BDW` platform to that of the `KNL68` one.

Table 3 attests that performance obtained on the `KNL64`/`KNL68` is better than two sockets of a recent Broadwell processor for matrices with enough parallelism in the elimination tree. The performance gain from `KNL64` to `KNL68` also shows that increasing the number of cores leads to performance improvements. Using this result, we can infer that `qr_mumps` allows for good scalability on the KNL and can scale further if the number of cores increases.

From the point of view of energy consumption during the factorization, it was measured out-of-band via IPMI interface by polling the total system power draw approximately every 0.1 seconds. To have the total energy consumption (in Joules), we then compute $E = \sum_{i=1}^{K} P_i \Delta t$, where P_i denotes the system power draw in Watts and Δt_i denotes the length of the interval in seconds. The comparison between `BDW` and `KNL68` platforms is then performed using the *Gflop/s/watt* metric which allows to evaluate the energy efficiency. As it is often the case with manycore platforms, we can see that with more cores at a lower
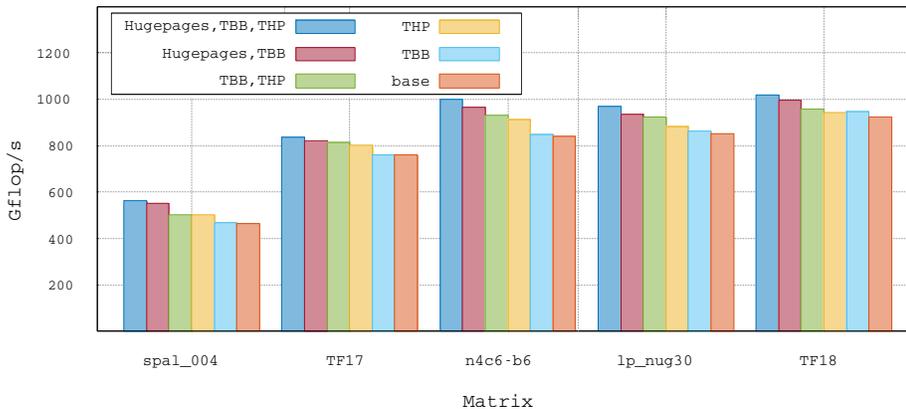

Figure 3: KNL configurations for optimal performance (`KNL64`).

	Gflop/s			Gflop/s/watt	
Matrix	BDW	KNL64	KNL68	BDW	KNL68
spal_004	605.35	562.21	579.43	1.31	1.91
TF17	674.51	837.55	954.50	1.49	2.88
lp_nug30	730.05	970.23	1057.18	1.65	3.13
n4c6-b6	759.01	1001.79	1076.38	1.62	3.12
TF18	761.72	1018.61	1092.40	1.56	3.03

Table 3: Performance and energy consumption.

frequency the KNL provides better energy efficiency than the BDW. If we exclude spal_004 which is not very performance-efficient on KNL but still more energy-efficient, we have a difference in energy efficiency represented in Gflop/s/watt of around 2. This means that for a given problem the KNL platforms are able to use twice less energy than the BDW system for doing the factorization.

5   Conclusion

One important conclusion of this study is that task-based programming can effectively ensure portability. Indeed, we showed that a state-of-the-art numerical code designed according to this paradigm can efficiently be ported from a standard multicore processor to a manycore one. This confirms that modern runtime systems (StarPU in our case) are mature enough to efficiently ensure the support of complex, irregular scientific libraries on such hardware.

We highlighted the important parameters to consider on a KNL processor and carefully assessed their impact on the overall performance with problems from real-life applications. In the best hardware/system configuration (quadrant clustering and cache MCDRAM KNL modes / enabling transparent and explicit Hugepages as well as TBB allocation), the resulting sparse direct solver could achieve up to 1 TFlop/s with an energy efficiency up to twice higher than on a regular Intel Xeon Broadwell processor.
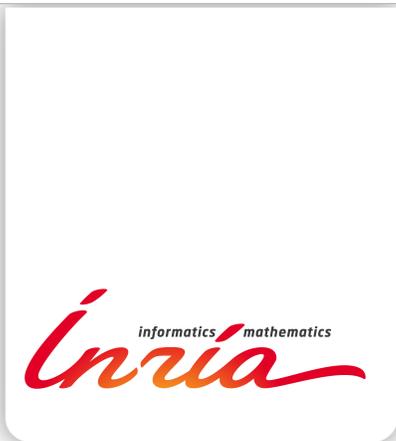
References

[1] Agullo, E., Buttari, A., Guermouche, A., Lopez, F.: Multifrontal QR factorization for multicore architectures over runtime systems. In: Euro-Par 2013 Parallel Processing. pp. 521–532. Springer Berlin Heidelberg (2013)

[2] Agullo, E., Buttari, A., Guermouche, A., Lopez, F.: Task-based multifrontal QR solver for GPU-accelerated multicore architectures. In: HiPC. pp. 54–63. IEEE Computer Society (2015)

[3] Agullo, E., Buttari, A., Guermouche, A., Lopez, F.: Implementing multi-frontal sparse solvers for multicore architectures with sequential task flow runtime systems. ACM Trans. Math. Softw. 43(2), 13:1–13:22 (Aug 2016)

[4] Agullo, E., Demmel, J., Dongarra, J., Hadri, B., Kurzak, J., Langou, J., Ltaief, H., Luszczek, P., Tomov, S.: Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. Journal of Physics: Conference Series 180(1), 012037 (2009)

[5] Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A.: StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009 23, 187–198 (Feb 2011)

[6] Ayguadé, E., Badia, R.M., Igual, F.D., Labarta, J., Mayo, R., Quintana-Ortí, E.S.: An extension of the StarSs programming model for platforms with multiple GPUs. In: Euro-Par. pp. 851–862 (2009)

[7] Bauer, M., Treichler, S., Slaughter, E., Aiken, A.: Legion: expressing locality and independence with logical regions. In: SC Conference on High Performance Computing Networking, Storage and Analysis, SC '12, Salt Lake City, UT, USA - November 11 - 15, 2012. p. 66 (2012)

[8] Bosilca, G., Bouteiller, A., Danalis, A., Faverge, M., Hérault, T., Dongarra, J.J.: Parsec: Exploiting heterogeneity to enhance scalability. Computing in Science and Engineering 15(6), 36–45 (2013)

[9] Bosilca, G., Bouteiller, A., Danalis, A., Herault, T., Luszczek, P., Dongarra, J.: Dense linear algebra on distributed heterogeneous hardware with a symbolic DAG approach. Scalable Computing and Communications: Theory and Practice pp. 699–733 (2013)

[10] Buttari, A.: Fine-grained multithreading for the multifrontal QR factorization of sparse matrices. SIAM Journal on Scientific Computing 35(4), C323–C345 (2013)

[11] Buttari, A., Langou, J., Kurzak, J., Dongarra, J.: A class of parallel tiled linear algebra algorithms for multicore architectures. Parallel Comput. 35, 38–53 (January 2009)

[12] Davis, T.A., Hu, Y.: The university of Florida sparse matrix collection. ACM Trans. Math. Softw. 38(1), 1:1–1:25 (Dec 2011)

[13] Doerfler, D., Deslippe, J., Williams, S., Oliker, L., Cook, B., Kurth, T., Lobet, M., Malas, T.M., Vay, J., Vincenti, H.: Applying the roofline performance model to the intel xeon phi knights landing processor. In: High Performance Computing - ISC High Performance 2016 International Workshops, ExaComm, E-MuCoCoS, HPC-IODC, IXPUG, IWOPH, P^3MA, VHPC, WOPSSS, Frankfurt, Germany, June 19-23, 2016, Revised Selected Papers. pp. 339–353 (2016)

[14] Duff, I.S., Reid, J.K.: The multifrontal solution of indefinite sparse symmetric linear systems. ACM Transactions On Mathematical Software 9, 302–325 (1983)

[15] Hadri, B., Ltaief, H., Agullo, E., Dongarra, J.: Tile QR factorization with parallel panel processing for multicore architectures. In: IPDPS. pp. 1–10. IEEE (2010)

[16] Haidar, A., Tomov, S., Arturov, K., Guney, M., Story, S., Dongarra, J.: LU, QR, and Cholesky factorizations: Programming model, performance analysis and optimization techniques for the Intel Knights Landing Xeon Phi. In: 2016 IEEE High Performance Extreme Computing Conference, HPEC 2016, Waltham, MA, USA, September 13-15, 2016. pp. 1–7 (2016)

[17] Kim, K., Eijkhout, V.: A parallel sparse direct solver via hierarchical DAG scheduling. ACM Trans. Math. Softw. 41(1), 3:1–3:27 (Oct 2014)

[18] Lacoste, X., Faverge, M., Ramet, P., Thibault, S., Bosilca, G.: Taking advantage of hybrid systems for sparse direct solvers via task-based runtimes (05/2014 2014)

[19] Malas, T.M., Kurth, T., Deslippe, J.: Optimization of the sparse matrix-vector products of an IDR Krylov iterative solver in emgeo for the intel KNL manycore processor. In: High Performance Computing - ISC High Performance 2016 International Workshops, ExaComm, E-MuCoCoS, HPC-IODC, IXPUG, IWOPH, P^3MA, VHPC, WOPSSS, Frankfurt, Germany, June 19-23, 2016, Revised Selected Papers. pp. 378–389 (2016)

[20] Morari, A., Gioiosa, R., Wisniewski, R.W., Rosenburg, B.S., Inglett, T.A., Valero, M.: Evaluating the impact of TLB misses on future HPC systems. In: Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International. pp. 1010–1021. IEEE (2012)

[21] Quintana-Ortí, G., Quintana-Ortí, E.S., Geijn, R.A.V.D., Zee, F.G.V., Chan, E.: Programming matrix algorithms-by-blocks for thread-level parallelism. ACM Trans. Math. Softw. 36(3) (2009)

[22] Reinders, J.: Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism. O'Reilly (2007)

[23] Rosales, C., Cazes, J., Milfeld, K., Gómez-Iglesias, A., Koesterke, L., Huang, L., Vienne, J.: A comparative study of application performance and scalability on the intel knights landing processor. In: High Performance Computing - ISC High Performance 2016 International Workshops, ExaComm, E-MuCoCoS, HPC-IODC, IXPUG, IWOPH, P^3MA, VHPC, WOPSSS, Frankfurt, Germany, June 19-23, 2016, Revised Selected Papers. pp. 307–318 (2016)

[24] Shmueli, E., Almasi, G., Brunheroto, J., Castanos, J., Dozsa, G., Kumar, S., Lieber, D.: Evaluating the effect of replacing CNK with Linux on the compute-nodes of Blue Gene/l. In: Proceedings of the 22nd annual international conference on Supercomputing. pp. 165–174. ACM (2008)

[25] Sodani, A.: Knights Landing (KNL): 2nd generation Intel; Xeon Phi processor. In: 2015 IEEE Hot Chips 27 Symposium (HCS). pp. 1–24 (Aug 2015)