Assessing the Performance of OpenMP Programs on the Intel Xeon Phi*

Dirk Schmidl[1,3], Tim Cramer[1,3], Sandra Wienke[1,3],
Christian Terboven[1,3], and Matthias S. Müller[1,2,3]

[1] Center for Computing and Communication, RWTH Aachen University, D - 52074 Aachen
[2] Chair for High Performance Computing, RWTH Aachen University, D - 52074 Aachen
[3] JARA High-Performance Computing, Schinkelstrae 2, D 52062 Aachen
{schmidl,cramer,wienke,terboven,mueller}@rz.rwth-aachen.de

Abstract. The Intel Xeon Phi has been introduced as a new type of compute accelerator that is capable of executing native x86 applications. It supports programming models that are well-established in the HPC community, namely MPI and OpenMP, thus removing the necessity to refactor codes for using accelerator-specific programming paradigms. Because of its native x86 support, the Xeon Phi may also be used stand-alone, meaning codes can be executed directly on the device without the need for interaction with a host. In this sense, the Xeon Phi resembles a big SMP on a chip if its 240 logical cores are compared to a common Xeon-based compute node offering up to 32 logical cores. In this work, we compare a Xeon-based two-socket compute node with the Xeon Phi stand-alone in scalability and performance using OpenMP codes. Considering both as individual SMP systems, they come at a very similar price and power envelope, but our results show significant differences in absolute application performance and scalability. We also show in how far common programming idioms for the Xeon multi-core architecture are applicable for the Xeon Phi many-core architecture and which challenges the changing ratio of core count to single core performance poses for the application programmer.

1 Introduction

Intel calls the new Intel Xeon Phi a coprocessor instead of using the term accelerator. Indeed it can be both, an accelerator which is used to speed up scientific applications, or a standalone SMP on a single chip. In the first case compute-intensive parts of an application can be executed on the device, as it is known from programming paradigms like CUDA or OpenCL explicitly designed for accelerators. For the Xeon Phi this can be achieved with the Intel Language Extensions for Offload (LEO). However, in this work we will focus on the second scenario and assess the behavior of Xeon Phi as an SMP machine with many cores. The coprocessor itself is able to run x86 code and supports many standard parallel programming paradigms like OpenMP or MPI which is meant to make the rewrite of a kernel or even complete applications unnecessary. The aim is to reach a much better usability and make the Xeon Phi available for a wider

* Parts of this work were funded by the German Federal Ministry of Research and Education (BMBF) under Grant No. 01IH11006.

F. Wolf, B. Mohr, and D. an Mey (Eds.): Euro-Par 2013, LNCS 8097, pp. 547–558, 2013.

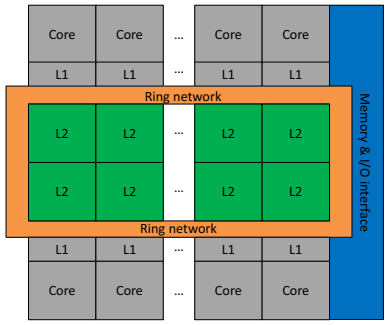
range of applications than it is possible with GPUs today. However, the fact that code optimized for multi-core architectures can run on this new many-core architecture by just recompiling does not mean that the performance for general-purpose applications is as desired. We investigate if OpenMP constructs, such as OpenMP tasks and nested parallel regions, can be used on the Xeon Phi efficiently. This would allow executing more complex programs natively in contrast to just offloading vector parallel loops as it is done for other accelerators. We present a performance evaluation with both basic kernels as well as more complex benchmarks, like a conjugate gradient solver to show that there are no general reasons which prevents an efficient use of the Xeon Phi as a SMP system on a chip. Furthermore, we evaluate the performance of the NAS parallel benchmarks and some real-world application codes, which use different methods to utilize the coprocessor.

The structure of the paper is as follows: First, we give an overview of related work done in this field in Sec. 2 and provide an overview of the Intel Xeon Phi architecture compared to Xeon architecture in Sec. 3. We start the performance evaluation in Sec. 4 benchmarking the memory subsystem, selected OpenMP constructs and CG method. Then we present the NAS parallel benchmarks and some real-world application codes, which use different methods to utilize the coprocessor, in Sec. 5 and Sec. 6, before we conclude this paper in Sec. 7.

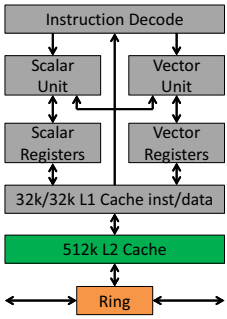
2 Related Work

Previous studies show that throughput-oriented processors like GPUs are one way to fulfill the requirement for more and more compute capabilities. This is not only valid for dense linear algebra kernels [18], but also for memory-bound kernels like sparse matrix vector multiplication [2] (depending on the matrix storage format). In order to benefit from the GPU compute capabilities in general-purpose CUDA applications it is essential to understand the underlying hardware architecture in addition to the programming model [6]. Thus, the effort for porting scientific applications to CUDA or OpenCL can be much higher compared to directive-based programming models like OpenMP [19].

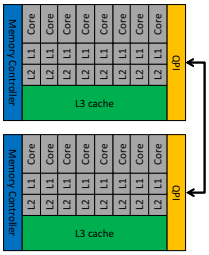
While early experiences on Intel Xeon Phi coprocessors revealed that porting scientific codes can be relatively straightforward [15], other studies also show that this does not necessarily mean that a reasonable performance can be reached without any architecture-specific optimizations like vectorization, software prefetching, SIMD intrinsics, large TLB tables, hardware-supported gather or the correct padding and alignment. [14] showed that the baseline implementation of an compute-intensive radar computation program can be slightly faster on one Xeon Phi compared to a two-socket Sandy Bridge (SNB) system, but that 2x speedup can only reached with architecture-specific optimizations and modifications of the algorithm. [20] gained similar results for multigrid methods which are widely used to accelerate the convergence of iterative solvers. In their study the baseline implementation of the code is even slower on one Xeon Phi compared to two Sandy Bridge processors, but also benefits from the coprocesser after specific optimizations. While [14,20] concentrate on one specific



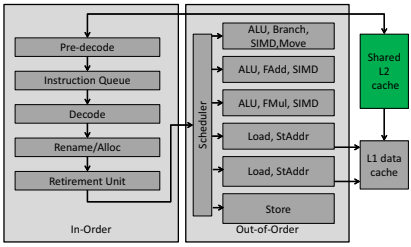
(a) High-level architecture of the Xeon Phi coprocessor.



(b) Functional units of the Xeon Phi coprocessor core.



(c) High-level architecture of the SNB system.



(d) Functional units of the Xeon processor core.

Fig. 1. CPU and core architectures of the Intel Xeon processor and the Intel Xeon Phi coprocessor

application, we focus on a wider range of smaller compute kernels as well as real-world application studies to assess the performance of OpenMP programs.

3 Architecture Comparison

In this section, we compare the architectural differences between the Intel Xeon E5 processors and the Intel Xeon Phi coprocessors.

The Intel Xeon Phi coprocessor provides a shared-memory many-core CPU that is packed on a PCI Express extension card. The specific version used here has 60 cache-coherent cores clocked at 1.053 GHz and 8 GB of coprocessor memory. Each core has 32 KB L1 instruction and data cache and a 512 KB L2 cache. A ring network connects all cores with each other and with memory and I/O devices (see. Fig. 1(a)). Every core in the SNB system has the same amount of L1 and L2 cache as the Xeon Phi cores, but there is also a shared 30 MB L3 cache on the SNB chip. The system used for our experiments consists of two 8-core chips clocked at 2.0 GHz, connected through the Intel Quick Path Interconnect (QPI) (see Fig. 1(c)). Hence, the two-socket SNB-system offers a NUMA architecture while the Xeon Phi has a flat memory model.

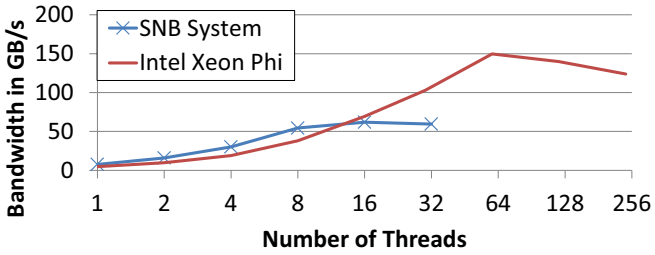


Fig. 2. Memory bandwidth on the SNB and the Intel Xeon Phi system, measured with the stream benchmark

The differences between the micro-architecture of the coprocessor and Xeon processors are more substantial (see Fig. 1(b) and 1(d)). The cores in the Xeon Phi chip are designed to deliver a high compute power per consumed watt. They are clocked comparably slowly at 1.053 GHz and execute instructions in-order. The strength of these cores is the vector unit, which can handle 512 bit vectors.

In contrast the SNB cores use a complex out-of-order engine, which is one contribution to the higher power demand of a single SNB core. However the out-of-order engine can optimize code execution on the core and depending on the executed instruction stream this may speedup execution a lot. The SNB cores are clocked at 2 GHz and support AVX vector operations with 256 bit vectors. Overall the peak performance of one core is nearly the same, the SNB core can deliver 16 GFLOPS and the Xeon Phi core 16.8 GFLOPS.

If we ignore the fact that the Xeon Phi needs a host system and look at it as a SMP system, both investigated systems consume roughly the same amount of power (250 W), space (one blade) and cost roughly the same amount of money, which makes this comparison valuable.

4 Kernel Benchmarks

This section presents some basic performance data for the Xeon Phi and SNB-system. For all benchmarks and applications investigated porting to the Xeon Phi required only to set `-mmic` as a compiler option for the Intel compiler.

4.1 Memory Benchmarks

The memory subsystem on both investigated platforms differs a lot since the SNB system uses DDR3 RAM whereas the Intel Xeon Phi is equipped with GDDR5 RAM. Here, we use simple benchmarks to highlight the differences in memory bandwidth and latency for both systems.

The stream benchmark [12] is a standard package to measure the available memory bandwidth on a system. Fig. 2 shows the results for the Triad vector operation ($\bar{a} = \bar{b} + x * \bar{c}$) for a memory footprint of roughly 2 GB. A good thread to core mapping was ensured with the affinity support of the Intel Compiler. We set `KMP_AFFINITY` to

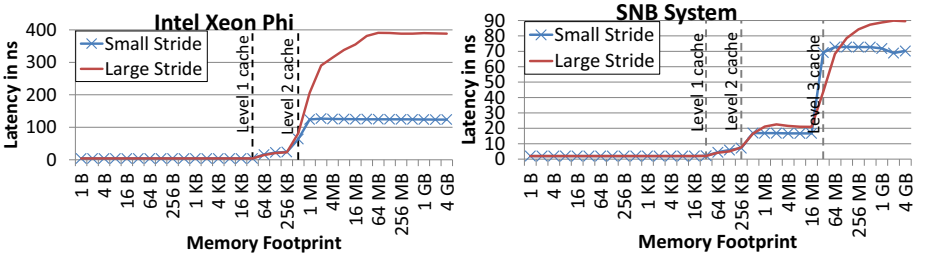


Fig. 3. Memory latency on the SNB system and the Xeon Phi system for different memory footprints. A random stride is used to walk through the memory to prevent prefetching. The small stride most probably hits on the same page whereas the large stride always hits on one of the next pages, causing also a TLB miss.

`balanced` on the Xeon Phi and to `scatter` on the SNB system, since these options delivered best results.

On both systems the bandwidth rises with the number of threads for a low number of threads. When 8 threads on the SNB or 60 threads on the Phi system are used the bandwidth reaches 62 GB/s or 150 GB/s, respectively. After this peak the bandwidth slightly drops down on both systems.

Next, we investigated the latency of the memory. We implemented a pointer chasing benchmark, similar to the latency test in lmbench [13]. We use a stride with a random offset to avoid latency hiding by hardware or software prefetching. Fig. 3 shows the latency for two different strides on both systems. One stride is small enough to hit the same memory page in most cases, whereas the large stride will always hit the next page if the memory footprint is large enough. This may cause a TLB miss if the corresponding TLB entry is not cached.

If the memory fits into the caches, the latency slightly rises with every cache level and the small or large stride does not make any difference since the TLB cache is large enough to keep all entries in the cache on both systems. With a memory footprint which only fits into the main memory the results diverge: On the SNB system, the latency is about 55 ns for the small and about 75 ns for the large stride, whereas it is 130 ns (small stride) and 400 ns (large stride) on the Xeon Phi. For the small stride the ratio of clock tick to memory latency is nearly the same on both systems since the Xeon Phi is clocked at 1 GHz and the SNB at 2 GHz. In contrast for the large stride the ratio is 400 clock ticks/cache miss and 150 clock ticks/cache miss on the Xeon Phi and SNB, respectively.

Concluding, the memory on the Xeon Phi can deliver a very high bandwidth compared to the SNB system, but the latency is worse for large strides between the data accesses.

4.2 OpenMP Constructs

The performance of the OpenMP runtime can be essential for the overall scalability of OpenMP applications. Here, we investigate the overhead of synchronization primitives in the Intel OpenMP runtime. First, we use the `syncbench` benchmark, which is part of the EPCC microbenchmarks [4] to measure the overhead of a `parallel`

`for`, a `barrier` and a `reduction` operation in OpenMP. Second, we use self-implemented extensions (see [16]) of the benchmark to investigate the overhead for OpenMP tasks and for nested parallel regions, the two only ways in OpenMP to express multi-level parallelism. For nested parallel regions we use an outer parallel region with two threads and vary the number of threads in the inner region. For tasks we examine the `single-producer` (one thread creates all tasks) and the `parallelproducer` (all threads create tasks in parallel) patterns for task creation.

Table 1. Overhead in microseconds of OpenMP synchronization constructs without nesting (top), in an inner nested parallel region (middle) and for OpenMP Tasks (bottom) on the Intel Xeon Phi and on the SNB system for a different number of threads

	Intel Xeon Phi			SNB system		
EPCC syncbench						
#threads	Parallel for	barrier	reduction	Parallel for	barrier	reduction
16	13.81	5.83	21.61	3.47	2.05	5.83
32	15.85	8.21	24.80	24.36	31.78	58.90
60	17.71	9.96	29.56			
240	27.56	13.37	48.86			
Nested Parallel Regions						
#threads	Parallel for	barrier	reduction	Parallel for	barrier	reduction
2 * 8	56.67	5.47	57.83	13.89	1.79	16.86
2 * 16	117.17	7.21	130.68	27.61	2.39	32.13
2 * 30	318.93	7.74	336.03			
2 * 120	1774.59	13.14	1824.63			
OpenMP Tasks						
#threads	serial-producer		parallel-producer	serial-producer		parallel-producer
16	81.18		1.67	63.25		0.74
32	165.50		1.78	146.41		4.11
60	294.55		3.54			
240	1355.90		8.39			

The systems differ in the number of cores which makes a one to one comparison of the overhead difficult. On the SNB system our experiences have shown that most applications deliver best performance for 16 (1 thread per core) or 32 threads (1 thread per hyperthread). On the Xeon Phi it takes 60 or 240 Threads, respectively, to utilize all cores or hyperthreads. We ensured the usage of one core per thread by thread binding for 16 and 60 threads on the SNB and Xeon Phi, respectively.

Table 1 shows the measurement results for the SNB system and for the Xeon Phi across different numbers of threads. For the non-nested constructs the overhead of using hyperthreads is much larger on the SNB system. If all cores start one thread, the SNB system is 3-5 times better than the Xeon Phi, whereas the time is nearly identical for the `reduction` and `parallel for`, when all hyperthreads are used.

For nested parallel regions, the overhead is much larger on the Xeon Phi system, whereas it is nearly the same (compared to the non-nested regions) on the SNB system. A reduction operation with 120 threads in the inner nested region takes 1824 microseconds, whereas the worst case on the SNB system (16 inner threads) takes only 32 microseconds. This shows that nested parallel regions introduce more overhead on the Xeon Phi system than on the SNB. For OpenMP tasks on the Xeon Phi, the overhead for the `single-producer` pattern is in the same order as for the nested parallel regions. Creating one task takes about 1355 microseconds, whereas it only takes 146 microseconds on the SNB system. However, creating tasks in parallel with the `parallel-producer` pattern works much better, here one task creation takes about 8 microseconds, which is only 2x more than on the SNB and much less than for the `serial-producer` pattern. The reason is that the tasks can be created in separate task queues with this pattern, whereas the `single-producer` pattern requires locking of the task queue when all threads steal tasks out of this queue.

We conclude that the OpenMP runtime on the Phi as SMP system on a single chip can handle thread and task creation without introducing much more overhead than on the SNB system although the number of threads is much larger on the Xeon Phi. If nested parallelism is needed to utilize the large number of threads available, the `parallel-producer` pattern with tasks seems to be an appropriate way to express this parallelism and it should be preferred over nested parallel regions if possible. Generally, if in a producer-consumer scenario only one thread is responsible for creating tasks to be executed by the other threads, increasing the core count while decreasing the single core performance (i.e. clock speed) as on the Xeon Phi may lead to the creator becoming a bottleneck.

4.3 Sparse-Matrix-Vector-Multiplication in a CG Method

To evaluate the performance of a real-world compute kernel, we use a CG solver [11]. The runtime of the algorithm is dominated by the Sparse-Matrix-Vector-Multiplication (SpMV), so we concentrate our analysis on this operation. Depending on the sparsity pattern of the matrix an adequate load balancing is needed. In the case of the CG-method the optimal load balancing can be reached by using a pre-calculated number of rows for each thread depending on the number of nonzero values per row, instead of using a static schedule of an OpenMP work sharing construct. If this is not possible for some problem class, OpenMP also offers some means for load balancing: The first is to use a dynamic schedule with an appropriated chunk size for work sharing construct, the second is to use OpenMP Tasks.

On ccNUMA machines, correct data and thread placement is essential [5]. For that reason we initialize the data in parallel in the pre-calculated version to distribute the pages over the sockets and bind the threads to the cores to avoid thread migration. For the two alternative implementations, an optimal data placement is not possible because of the dynamic scheduling, so that we use a static schedule for the data initialization. However, in [16] and [17] we have shown that at least for the tasking approach the Intel OpenMP runtime works quite well for the `parallel-producer multiple-executors` pattern. Since the two test systems differ in amount and efficient usability of hardware threads, we use different binding strategies, which are

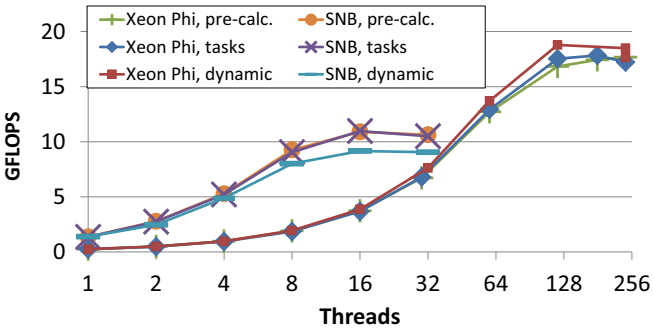


Fig. 4. SpMV performance (1000 CG iterations) on the SNB and the Intel Xeon Phi system

optimal for this kernel and the corresponding hardware. The matrix represents a computational fluid dynamics problem (Fluorem/HV15R) and is taken from the University of Florida Sparse Matrix Collection [8]. The matrix is stored in compressed row storage (CRS) format. The matrix dimension is $N = 2,017,169$ and the number of non-zero elements is $nnz = 283,073,458$, which results in a memory footprint of approximately 3.2 GB. Hence, the data set does not fit into the caches.

Fig. 4 shows the performance results for the three implementations of the SpMV (1000 CG iterations) on both systems. The version using the pre-calculated distribution of the non-zeros reaches the same performance and scalability as the tasking version on the two-socket SNB system because both have been optimized for big shared-memory systems. The measurements show that the corresponding two unmodified, cross-compiled Intel Xeon Phi implementation variants also end up at roughly the same performance. In contrast to the SNB system, the worksharing version using a dynamic schedule works even better on the Intel Xeon Phi. The reason for the different behavior of the two systems is that an optimal data distribution cannot be achieved with a dynamic schedule on ccNUMA machines, which is not an issue on the Xeon Phi. The direct comparison between the two SMP systems shows that the Xeon Phi can profit from the higher memory bandwidth for this kernel and reaches a 1.7x better performance than the two Sandy Bridges. While the peak is reached at about 11 GFLOPS on the SNB system with using only one hardware thread per core, the maximum performance on the Xeon Phi is at about 18.8 GFLOPS with 120 threads (2 hardware threads per core). In [7] we showed with the help of the Roofline Model [21] that this performance is close to the theoretical maximum taking the memory-bound character of this kernel into account.

The results show that for this kernel benchmark OpenMP tasking, as well as OpenMP worksharing with different scheduling strategies, run fine on the Intel Xeon Phi without any special tuning for the architecture.

5 NAS Parallel Benchmarks

The NAS Parallel Benchmarks [1] are a set of benchmarks designed to evaluate the parallel performance of parallel computers. We ran the reference implementation without any code change. On both systems we enabled compiler optimization and used version 13.0 of the Intel Compiler.

Table 2. Runtime (in seconds) and speedup of the NAS parallel benchmarks on the Xeon Phi and the SNB system

	SNB			Xeon Phi		
Benchmark	1 Thread	32 Threads	Speedup	1 Thread	240 Threads	Speedup
IS	23.12	1.38	16.75	192.49	2.46	78.25
EP	186.81	8.11	23.03	1518.42	13.34	113.82
MG	64.04	8.03	7.98	498.94	9.63	51.81
FT	306.11	19.19	15.95	2393.01	53.97	44.34
BT	1241.63	82.61	15.03	9433.52	132.29	71.31
SP	826.25	137.69	6.00	12264.29	164.59	74.51
LU	1109.76	62.23	17.83	9835.09	163.33	60.22

Table 2 shows the runtimes for problem size `C` of all benchmarks on both systems with one thread and with best effort performance, which was when all threads were used. The speedup on the SNB system is between 6 and 24, on the Xeon Phi system between 44 and 114. This shows that the benchmarks scale well on both systems and that the Xeon Phi system can deliver a good scalability for standard kinds of applications. But the serial performance on the Xeon Phi is much lower compared to the SNB system. With one thread the benchmarks run between 7.5 and 15 times slower on the Xeon Phi. Given that the theoretical peak performance of one phyiscal core is nearly the same for both, this is a surprising result. Although the speedup on the Xeon Phi system is quite impressive with up to 114 on 60 cores, the Xeon Phi system is in total slower for every benchmark, when all resources are used.

6 Application Case Studies

After studying the performance of kernels and benchmark codes, we investigated the performance of four real-world codes of the RWTH Aachen University, namely:

iMOOSE: The innovative Modern Object Oriented Solver Environment (iMOOSE) is a finite elements package developed by the Institute of Electrical Machines[1] at RWTH Aachen University. The native compilation of this heavy C++ code ($\sim 300,000$ lines of code) parallelized with OpenMP worked without any problems using the Intel Compiler. In our measurements we investigate a three-dimensional model of permanent-magnet excited synchronous machine. We only look at the solving process which uses a CG-solver and dominates the total serial run time on the host system (up to 90 %).

FIRE: The Flexible Image Retrieval Engine (FIRE) [9], developed at the Human Language Technology and Pattern Recognition Group of RWTH Aachen University, takes a set of query images and for each query image it returns a number of similar images from an image database.

NestedCP: NestedCP [10] is developed from the Virtual Reality Group of the RWTH Aachen University and is used to extract critical points in unsteady flow field datasets.

[1] `http://www.iem.rwth-aachen.de`

Table 3. Elapsed time for the applications on Intel Sandy Bridge (SNB) and Intel Xeon Phi Coprocessor

	SNB			Xeon Phi		
Application	1 Thread	best (#threads)	Speedup	1 Thread	best (#threads)	Speedup
iMOOSE	104.68	12.20 (16)	8.58	1243.54	15.59 (240)	79.74
FIRE	284.60	16.68 (32)	17.06	2672.71	38.25 (234)	98.02
NestedCP Nested	46.99	3.21 (32)	14.62	845.14	35.58 (240)	23.76
NestedCP Tasking	47.34	2.43 (32)	19.47	848.34	11.14 (240)	76.16
NINA	470.06	61.16 (16)	7.68	1381.94	27.29 (177)	50.64

Critical points are essential parts of the velocity field topologies and extracting them helps to interactively visualize the data in virtual environments. Two versions of the code were investigated, first a version parallelized with nested parallel regions and second, a version using OpenMP tasks to express the parallelism on the same levels.

NINA: The software package for the solution of Neuromagnetic INverse lArge-scale problems (NINA) was developed by Bücker, Beucker and Rupp [3] and deals with the reconstruction of focal activity in the human brain. It includes computations of matrix-vector products using a matrix of dimensions 128×512000. Here, we use an established C framework in a simplified version that mimics the original MATLAB approach.

Table 3 shows the runtime of the example applications on the SNB system and the Xeon Phi. Noticeable is that nearly all applications gain a good speedup on the Xeon Phi system of 50 to 80. The only exception is the NestedCP version parallelized with nested parallel regions, the tasking version instead delivers a speedup of 76. This confirms our assumption from Sect. 4.2 that tasking is a more appropriate way to express multi-level parallelism on the Xeon Phi system. However, although the scalability is good for all codes on both systems, the total runtime is higher on the Xeon Phi except for one code. The reason again is the serial runtime of the Xeon Phi cores. iMOOSE, FIRE and NestedCP are slower by a factor of 10 to 18 compared to one SNB core. NINA is only slower by a factor of three with one thread and because of the good scalability on the Xeon Phi, the system outperforms the SNB system by a factor of 2.2. A profile for the NINA code showed that roughly 95 % of the kernel execution time is spent in a dense matrix-vector multiplication which is performed very often. This memory access pattern and floating point operations of this operation is very similar to the stream benchmark, where two vectors are multiplied. Since all matrix elements are needed only once, the operation is memory bandwidth bound and since the accesses are consecutive in the matrix and the vector, latency is not important. The stream benchmark has shown that the Xeon Phi can reach a about 2.5 times higher memory bandwidth compared to the SNB system, this is why the NINA code performs well here. All the other codes do not have one single hotspot and they do not use dense linear algebra. Our assumption is that they profit much more from the out-of-order execution capabilities of the SNB cores and thus the SNB system outperforms the Xeon Phi for all these codes.

7 Conclusion

We investigated the performance of Intel's new Xeon Phi coprocessor, if it is used as a standalone SMP system. There are two basic differences between the Xeon Phi and the Sandy Bridge system we used for comparison. First, the Xeon Phi offers many more cores (60) and hardware threads (240) than the SNB system (16 cores / 32 hardware threads). Second, the design of the Xeon Phi cores is simpler and uses in-order execution, whereas the Sandy Bridge cores can do out-of-order execution. We find that the memory bandwidth of the Xeon Phi is about 2.5x higher than the SNB system, but the memory latency for large strides is much lower on the SNB system. Furthermore, we measured the overhead of OpenMP synchronization constructs for single and multi-level parallelism, as well as the overhead introduced by task regions. Our findings are that the overhead for these constructs are in the same order of magnitude for both systems, although a much larger number of threads is needed on the Xeon Phi to utilize all resources of the chip. Thus, the OpenMP runtime should not prevent applications from scaling to a large number of threads on the new platform. Indeed, the NAS parallel benchmarks and all user applications investigated (iMOOSE, FIRE, NestedCP and NINA) have shown a good scalability on the Xeon Phi system between 50 and 113. However, our results also show that the serial performance of one Xeon Phi core is outperformed by a SNB core by a factor of 8-12 for many applications. This leads to a better overall performance on the SNB system for most of these applications. NINA was the only application that delivered a better overall performance on the Xeon Phi where it was 2.2 times faster than on the SNB system. The code executed dense matrix vector multiplications in 95 % of the compute-intensive parts. The other codes have less predictable memory access patterns. We assume that the high memory latency of the Xeon Phi is an issue here since the in-order engine cannot hide the latency in contrast to the out-of-order engine of the SNB. According to our experience, if the Xeon Phi is used as a stand alone SMP, it does not deliver a performance comparable to a Sandy Bridge system for many applications, because of the poor single core performance. For some applications, like the NINA code, the performance is fine, but for most of our codes the SNB system is the platform of choice. Future work is to investigate which kind of applications can be tuned to perform well on the Intel Xeon Phi system and which tuning steps are necessary.

References

1. Bailey, D.H., Barszcz, E., Barton, J.T., Browning, D.S., Carter, R.L., Fatoohi, R.A., Frederickson, P.O., Lasinski, T.A., Simon, H.D., Venkatakrishnan, V., Weeratunga, S.K.: The nas parallel benchmarks. Technical report, NASA Ames Research Center (1991)
2. Bell, N., Garland, M.: Implementing sparse matrix-vector multiplication on throughput-oriented processors. In: Proc. of the Conference on High Performance Computing Networking, Storage and Analysis, SC 2009, pp. 18:1–18:11. ACM, New York (2009)
3. Bücker, H.M., Beucker, R., Rupp, A.: Parallel Minimum p-Norm Solution of the Neuromagnetic Inverse Problem for Realistic Signals Using Exact Hessian-Vector Products. SIAM J. on Scientific Computing 30(6), 2905–2921 (2008)
4. Bull, J.M.: Measuring Synchronisation and Scheduling Overheads in OpenMP. In: Proc. of First European Workshop on OpenMP, pp. 99–105 (1999)

5. Terboven, C., an Mey, D., Schmidl, D., Jin, H., Wagner, M.: Data and Thread Affinity in OpenMP Programs. In: Proc. of the 2008 Workshop on Memory Access on Future Processors: A Solved Problem?, MAW 2008, pp. 377–384. ACM (2008)
6. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Skadron, K.: A performance study of general-purpose applications on graphics processors using CUDA. J. Parallel Distrib. Comput. 68(10), 1370–1380 (2008)
7. Cramer, T., Schmidl, D., Klemm, M., an Mey, D.: OpenMP Programming on Intel Xeon Phi Coprocessors: An Early Performance Comparison. In: Proc. of the Many-core Applications Research Community Symposium, pp. 38–44 (November 2012)
8. Davis, T.A.: University of Florida Sparse Matrix Collection. NA Digest 92 (1994)
9. Deselaers, T., Keysers, D., Ney, H.: Features for image retrieval: an experimental comparison. Information Retrieval 11(2), 77–107 (2008)
10. Gerndt, A., Sarholz, S., Wolter, M., an Mey, D., Bischof, C., Kuhlen, T.: Nested OpenMP for Efficient Computation of 3D Critical Points in Multi-Block CFD Datasets. In: SC 2006 Conference, Proc. of the ACM/IEEE 2006, p. 46 (November 2006)
11. Hestenes, M.R., Stiefel, E.: Methods of Conjugate Gradients for Solving Linear Systems. J. of Research of the National Bureau of Standards 49(6), 409–436 (1952)
12. McCalpin, J.: STREAM: Sustainable Memory Bandwidth in High Performance Computers
13. McVoy, L., Staelin, C.: lmbench: portable tools for performance analysis. In: Proc. of the 1996 Annual Conference on USENIX, ATEC 1996, p. 23. USENIX Association, Berkeley (1996)
14. Park, J., Tang, P.T.P., Smelyanskiy, M., Kim, D., Benson, T.: Efficient backprojection-based synthetic aperture radar computation with many-core processors. In: Proc. of the Int. Conference on High Performance Computing, Networking, Storage and Analysis, SC 2012, pp. 28:1–28:11. IEEE Computer Society Press, Los Alamitos (2012)
15. Schulz, K.W., Ulerich, R., Malaya, N., Bauman, P.T., Stogner, R., Simmons, C.: Early Experiences Porting Scientific Applications to the Many Integrated Core (MIC) Platform. Technical report, TACC-Intel Highly Parallel Computing Symposium (April 2012)
16. Terboven, C., Schmidl, D., Cramer, T., an Mey, D.: Assessing OpenMP Tasking Implementations on NUMA Architectures. In: Chapman, B.M., Massaioli, F., Müller, M.S., Rorro, M. (eds.) IWOMP 2012. LNCS, vol. 7312, pp. 182–195. Springer, Heidelberg (2012)
17. Terboven, C., Schmidl, D., Cramer, T., an Mey, D.: Task-Parallel Programming on NUMA Architectures. In: Kaklamanis, C., Papatheodorou, T., Spirakis, P.G. (eds.) Euro-Par 2012. LNCS, vol. 7484, pp. 638–649. Springer, Heidelberg (2012)
18. Volkov, V., Demmel, J.W.: Benchmarking GPUs to tune dense linear algebra. In: Proc. of the 2008 ACM/IEEE Conference on Supercomputing, SC 2008 (2008)
19. Wienke, S., Plotnikov, D., an Mey, D., Bischof, C., Hardjosuwito, A., Gorgels, C., Brecher, C.: Simulation of bevel gear cutting with GPGPUs - performance and productivity. Computer Science - Research and Development 26, 165–174 (2011)
20. Williams, S., Kalamkar, D.D., Singh, A., Deshpande, A.M., Van Straalen, B., Smelyanskiy, M., Almgren, A., Dubey, P., Shalf, J., Oliker, L.: Optimization of geometric multigrid for emerging multi- and manycore processors. In: Proc. of the Int. Conference on HPC, Networking, Storage and Analysis, SC 2012 (2012)
21. Williams, S., Waterman, A., Patterson, D.: Roofline: An Insightful Visual Performance Model for Multicore Architectures. Commun. ACM 52(4), 65–76 (2009)