Concurrent Kernel Execution on Xeon Phi within Parallel Heterogeneous Workloads

Florian Wende[1], Thomas Steinke[1], and Frank Cordes[2]

[1] Zuse Institute Berlin, Takustraße 7, D-14195 Berlin, Germany
{wende,steinke}@zib.de
[2] GETLIG&TAR GbR, Bachstelzenstraße 33A, D-14612 Falkensee, Germany
cordes@getlig.com

Abstract. Computations with a sufficient amount of parallelism and workload size may take advantage of many-core coprocessors. In contrast, small-scale workloads usually suffer from a poor utilization of the coprocessor resources. For parallel applications with small but many computational kernels a concurrent processing on a shared coprocessor may be a viable solution. We evaluate the Xeon Phi offload models Intel LEO and OpenMP4 within multi-threaded and multi-process host applications with concurrent coprocessor offloading. Limitations of OpenMP4 regarding data persistence across function calls, e.g. when used within libraries, can slow down the application. We propose an offload-proxy approach for OpenMP4 to recover the performance in these cases. For concurrent kernel execution, we demonstrate the performance of the different offload models and our offload-proxy by using synthetic kernels and a parallel hybrid CPU/Xeon Phi molecular simulation application.

1 Introduction

Throughout the different kinds of applications from science and economy performance gains by up to one order of magnitude are demonstrated by using coprocessors like GPGPUs (General Purpose Graphics Processing Units) or Intel's Xeon Phi instead of traditional multi-core CPUs when the problem is large-scale and highly regular [1,2]. In contrast, small-scale computations usually suffer from a poor utilization of the coprocessor device as a whole. A usual means to achieve acceptable utilization in these cases is executing many such computations in a concurrent manner. This can be done either by merging multiple small compute kernels into a larger "super kernel," or by offloading multiple small kernels for a "concurrent kernel execution" on the coprocessor.

Our work addresses application scenarios of the said type with offloads to the Xeon Phi ("Phi" for short hereafter) from within multi-threaded and multi-process workloads. Our contributions are:

1. A performance evaluation of concurrent offloading to Xeon Phi using Intel's Language Extension for Offload (LEO) and OpenMP4.
2. We study the impact of thread placements on Xeon Phi: Multiple concurrent offloads should not perturb each other.

F. Silva et al. (Eds.): Euro-Par 2014, LNCS 8632, pp. 788–799, 2014.

3. We demonstrate how multiple simultaneous offload data transfers between host and Xeon Phi can affect the overall program performance.
4. For OpenMP4, we propose an offload-proxy pattern to enable data persistence across different function scopes.

In Section 2 we discuss related work. Section 3 is on the Xeon Phi coprocessor and it briefly introduces the Intel LEO and OpenMP4 offload programming model. In Section 4 we use synthetic kernels to get information about the achievable performance in the case of compute and memory bound computations. Section 5 focuses on a real-world application implementing a simulation of a small molecule solvated within a nanodroplet. The application serves as a representative of a parallel heterogeneous workload. Section 6 concludes.

2 Related Work

The offload model and runtime system for the Intel Xeon Phi coprocessor is detailed by Newburn et al. [3].

Johnson et al. [4] explore the support for, what the authors call, Many-Task Computing (MTC) on the Xeon Phi platform. The authors' framework GeMTC is interfaced to Intel's SCIF communication API. It is based on a client server architecture with persistent threads or processes on the Phi. The authors investigate the overhead associated with the task offload itself. With 90% efficiency their approach outperforms OpenMP's offload mechanism.

Somehow related to our real-world application, Pennycook et al. [5] analyze the miniMD benchmark (Sandia) on Xeon Phi. The authors present a variety of optimizations, e.g. taking advantage of the Phi's SIMD units. They achieve performance improvements of about a factor 4 – 5 depending on the problem size and the cut-off value. With their minimal size of 32,000 atoms, the authors consider problem sizes that are more than one order of magnitude above what is addressed by our real-world application.

Prior to Xeon Phi, concurrent kernel execution [6] has been known from Nvidia GPGPUs of the Fermi architecture and later. A major drawback of Fermi is false-serialization of concurrent kernels as a result of the GPU is fed by just one task queue [7]. Current Nvidia GPGPUs provide 32 hardware queues (Hyper-Q) to improve concurrent kernel execution. Investigations on using Hyper-Q from within parallel workloads on the host can be found in [8]. However, a comparison of Xeon Phi offloading with its GPGPU counterpart is not part of this work.

3 Intel Xeon Phi Offload Programming

The Xeon Phi coprocessor is based on Intel's Many Integrated Core (MIC) architecture. It presently holds up to 61 64-bit compute cores [9], each of which with fully-coherent L1 and L2 cache, a 512-bit SIMD vector unit, and 4-way hardware multi-threading. Current Xeon Phis are used as coprocessors to a distinguished host system, where communication with the host is over PCIexpress

(PCIe). The Phi runs its own Linux OS, enabling for a flexible integration into cluster- and supercomputer setups.

From the programmers point of view there are two approaches to involve the Phi into computations: (i) native program execution with support for message passing, e.g. via MPI, and (ii) offload execution with the Phi as a coprocessor to the CPU. While native execution on Xeon Phi requires the entire application be parallelizable, the offload model is the common means to involve the Phi into codes with both serial and parallel sections. In this work, we therefore focus on the offload model and compare against native executions only where meaningful.

Intel LEO and OpenMP4. The Intel Language Extension for Offload (LEO) is a non-shared memory offload model for the Intel Xeon Phi coprocessor [9]. It provides a set of directives to the programmer that allow to mark code regions within a host program to be executed on the coprocessor if present. Since host and coprocessor are physically separate compute devices, memory transfers between the two are necessary in order to provide data for and get results of the computation(s). The models are appropriate for dealing with flat data structures that can be moved bitwise between host and coprocessor. For array-based data structures the copy direction, and the amount of elements to be moved need to be specified in the offload clauses – the actual copy process is implicit.

Figure 1 gives a code snippet that adds two vectors `a` and `b` into `c` using LEO and OpenMP4 – Xeon Phi device 0 is used. `a` and `b` are moved from the host to the coprocessor, and `c` is copied back after the computation. Except for different directives and clauses OpenMP4 is compatible with LEO.

Persistent Data on the Coprocessor. The execution of the offload regions in Fig. 1 go along with the (de)allocation of memory buffers on the coprocessor and the actual data transfers into/from these buffers before and after the offload computation. Repeated offloading with intensive data transfers thus can result in non-negligible overhead and hence reduced overall performance.

Both LEO and OpenMP4 allow for the allocation of memory on the coprocessor, retaining and reusing it across multiple offload regions within the same thread (process) context, and releasing it after the computation [9,10]. Enabling data persistence in LEO is done via the `alloc_if(cond)` and the `free_if(cond)` clause – memory is allocated or freed only if `cond` is `true` respectively `1`. In the OpenMP4 model, keeping data on the coprocessor across multiple offloads is possible within `omp target data` regions only. Offload regions that are enclosed by

```
float a[size],b[size],c[size];
// Offload using Intel LEO:
#pragma offload target(mic:0)\
   in(a[0:size]) in(b[0:size])\
   out(c[0:size])
   { c[0:size]=a[0:size]+b[0:size]; }
```

```
// Offload using OpenMP4:
#pragma omp target device(0)\
   map(to:a[0:size]) map(to:b[0:size])\
   map(from:c[0:size])
   { c[0:size]=a[0:size]+b[0:size]; }
```

Fig. 1. Vector addition using the LEO and the OpenMP4 offload model, respectively

a `target data` region inherit memory allocations associated with variables listed in the surrounding `target data` directive clauses.

Figure 2 illustrates the use of persistent memory on the coprocessor: Ⓐ Allocate memory and transfer data from host to coprocessor without freeing it. Ⓑ Reuse data for computation and copy content of `b` to the host. Ⓒ same as Ⓑ, but memory is freed eventually. For OpenMP4 the regions marked Ⓧ, Ⓨ, Ⓩ correspond to Ⓐ, Ⓑ, Ⓒ. Note the `target update` construct in Ⓨ, where data is moved from the coprocessor to the host within the `target data` region.

Although both models allow for persistent data on the coprocessor, LEO is more flexible since memory allocated via `alloc_if(1)` can be used anywhere in the same thread (process) context. As OpenMP4's `target data` region cannot extend across different function scopes, function calls need to be enclosed by it and variables representing persistent data have to be explicitly passed through. Using OpenMP4 offload e.g. within libraries thus requires the user of the library to create the `target data` region within its code. Contrary to design principles, the user gets involved into the library's memory management on the coprocessor.

OpenMP4 Offload within Libraries Using an Offload-Proxy. One solution to the `target data` problem when using OpenMP4 offload within libraries is using an offload-proxy that is instantiated by the library itself. The proxy creates a `target data` region, enters it, and remains within that region. Library calls create tasks and use a signaling mechanism to wake up the proxy and make it execute the tasks. When finished a task the proxy signals back to the caller.

A similar offload-proxy approach has been already evaluated by the authors in the context of concurrent kernel execution on Nvidia Fermi GPGPUs [7]. Although using the proxy pattern requires code modifications – when not included into the library design from the first – the following benefits can be noted: (i) it implements asynchronicity regarding coprocessor offloads, and (ii) for OpenMP4 it enables data persistence across different function scopes. The latter is also relevant for the integration of OpenMP4 offloading into C++ class designs.

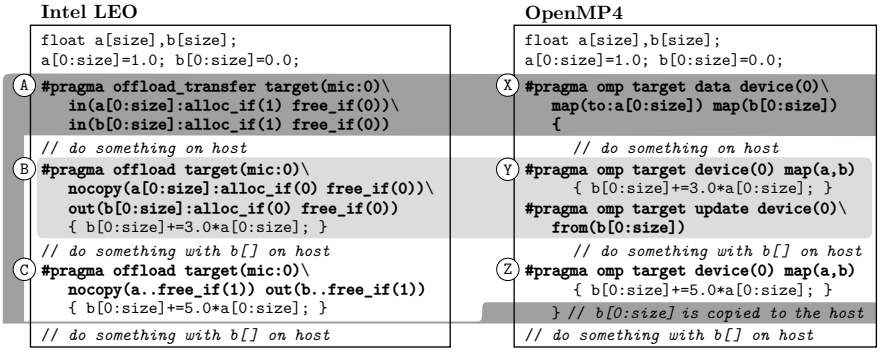

Fig. 2. Persistent data on the coprocessor using LEO and OpenMP4. White, gray- and light-gray-shaded regions have the same meaning in the two models.

4 Synthetic Benchmarks

In this section we assess the performance that can be achieved with LEO and OpenMP4 (+proxy), where multiple concurrent host threads (processes) offload (i) a compute bound, and (ii) a memory bound kernel to the coprocessor each. On the coprocessor itself OpenMP is used within the kernels. As representatives for (i) and (ii) we decided for the Intel MKL SGEMM and the STREAM Copy and Triad benchmark. Our intention is for multiple concurrent "small-scale" setups to determine the fraction of the performance achievable compared to "large-scale" setups, and to find out meaningful thread placements on the coprocessor.

Hardware and Software Setup. We use a compute node hosting two Xeon E5-2670 octa-core CPUs (Hyper-Threading enabled), 64 GB RAM, and two Intel Xeon Phi 7120P connected to the host via PCIe x16. Each Phi has 61 physical (244 logical) cores, and 16 GB ECC RAM – for benchmarking we use 60 physical cores (one core is reserved for the Phi's OS) and have ECC enabled. The host runs a CentOS 6.3 Linux with kernel 2.6.32-279. We use the Intel MPSS 2.1.6720-19, Intel compilers 14.0.3 (C++) and 14.0.1 (Fortran), and Intel MPI 4.1.1.036.

Benchmarking Setup and Methodology. For both SGEMM and STREAM we vary the number of OpenMP threads and MPI processes on the host between $p = 1, \ldots, 60$. Each host thread (process) offloads a set of SGEMM kernels – we call SGEMM directly on the Phi – or a STREAM kernels to the coprocessor by means of LEO respectively OpenMP4. Each offload uses $x = 1, \ldots, 4$ OpenMP threads on the Phi for computation. The benchmarks are written as libraries to allow for portability and ease of integration.

A single benchmark run consists of $N = 50$ successive offloads per thread (process) using respective library calls. To determine the performance of a single benchmark run, we measure the execution time of all offloads and use this value to estimate the compute performance in case (i), and the bandwidth in case (ii). The benchmark runs are repeated 10 times for each setup.

For each offload we take start and end times $\{t^{\mathrm{s}}_{i,k}\}$ and $\{t^{\mathrm{e}}_{i,k}\}$ ($i = 1, \ldots, N$ and $k = 1, \ldots, p$) of the offload (including all overheads and data transfers), and $\{\tau^{\mathrm{s}}_{i,k}\}$ and $\{\tau^{\mathrm{e}}_{i,k}\}$ for kernel execution on Xeon Phi – time stamps are taken with `clock_gettime(CLOCK_REALTIME,..)`. We approximate the degree of concurrency $\mathcal{C}_t$ across all p host threads (processes) as follows: Let $t^{\mathrm{s}} = \max\{t^{\mathrm{s}}_{1,k}\}$, $t^{\mathrm{e}} = \min\{t^{\mathrm{e}}_{N,k}\}$, $\Delta t = t^{\mathrm{e}} - t^{\mathrm{s}}$, and $\Delta t_{i,k} = t^{\mathrm{e}}_{i,k} - t^{\mathrm{s}}_{i,k}$. With $W_t = \{\Delta t_{i,k} : t^{\mathrm{s}}_{i,k} \geq t^{\mathrm{s}} \wedge t^{\mathrm{e}}_{i,k} \leq t^{\mathrm{e}}\}$, we have $1 \leq \frac{1}{\Delta t} \sum_{\omega \in W_t} \omega \leq p$. Hence, $\mathcal{C}_t \approx \frac{1}{p-1}\big(\frac{1}{\Delta t} \sum_{\omega \in W_t} \omega - 1\big) \in [0, 1]$. A similar expression holds for the thread concurrency $\mathcal{C}_\tau$ on the Phi.

If computations perfectly overlap, and if the offloading overhead is negligible then $\mathcal{C}_\tau \leq \mathcal{C}_t \approx 1$. If memory transfers before and/or after the actual computation take place, $\mathcal{C}_t$ can be significantly larger than $\mathcal{C}_\tau$.

Thread Affinity on the Coprocessor. Since the Xeon Phi runs a Linux operating system, assigning threads to specific compute cores can be done by means

of cpu-set masks directly within the offload kernel using the Linux scheduler interface. For our setups we use up to 60 threads (processes) on the host, each with a thread group of size up to 4 on Xeon Phi. We establish a "scatter-compact" thread pinning with "scatter" on the level of the groups and "compact" within the groups. The creation of the per-thread cpu-set mask is illustrated in Fig. 3 – on Xeon Phi the 0th and the last 3 logical cores are reserved for the Phi's OS.

```
#pragma omp parallel num_threads(4) // groupId=0..59
{
  cpu_set_t cpuMask; CPU_ZERO(&cpuMask);
  CPU_SET(1+(4*groupId+omp_get_thread_num())%(4*60),&cpuMask);
  sched_setaffinity(0,sizeof(cpu_set_t),&cpuMask);
}
```

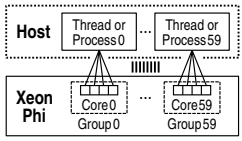

Fig. 3. Setting the thread affinity via `sched_setaffinity()` on Xeon Phi 7-series devices. The pinning model considered is "scatter-compact" (see text).

In many cases this low-level approach can be replaced by setting specific environment variables: e.g. `OMP_PLACES=threads|cores` results in successive logical respectively entire physical cores are assigned to OpenMP threads within multi-threaded offload kernels in the order the threads are created. We found using `KMP_AFFINITY` in multi-offload setups places OpenMP threads within different offload kernels on the same cores, resulting in oversubscription, potentially causing performance degradation.

Intel MKL SGEMM Benchmark. We consider two different benchmarking modes: all data is copied to Xeon Phi at the beginning, and is reused throughout all computations (M1) without any additional data transfers between host and Phi, and (M2) with data transfers containing 25%, 50%, and 100% of the problem size – 100% means two matrices are copied to the Phi, and one is copied back to the host. Benchmark results using matrices of size 1024^2 are illustrated in Fig. 4. Selected results for matrices of size 256^2, 512^2, and 2048^2 are given in Tab. 1.

With `OMP_PLACES=threads` and 1 thread per offload only 15 cores of the Phi are used if $p = 60$, whereas with `OMP_PLACES=cores` one thread resides on every physical core. The difference in the performance can be seen in sub-plots a) and

Table 1. Selected SGEMM performance results for runs using 60 host threads (processes) with 4 threads on the Xeon Phi each. Matrices have size $256^2, 512^2, 2048^2$.

	OpenMP on Host			MPI on Host		
Intel MKL SGEMM	256^2	512^2	2048^2	256^2	512^2	2048^2
(M1) Performance [GFlops/s]	396±2	1226±4	1577±2	328±5	1157±5	1564±2
Concurrency $\mathcal{C}_t \vert \mathcal{C}_\tau$	0.92|0.92	0.93|0.93	0.93|0.93	0.90|0.88	0.91|0.90	0.95|0.94
(M2) Performance [GFlops/s]	94±2	440±3	798±7	234±1	677±2	1452±10
Concurrency $\mathcal{C}_t \vert \mathcal{C}_\tau$	0.94|0.10	0.93|0.29	0.94|0.48	0.85|0.34	0.89|0.46	0.92|0.88

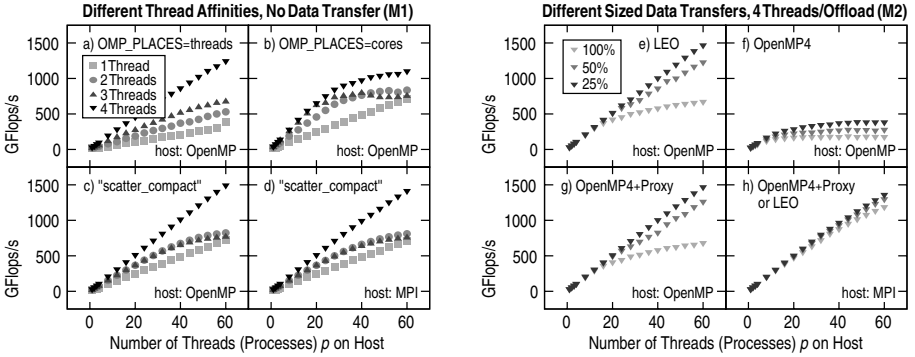

Fig. 4. Performance for p concurrent SGEMM offload computations on Xeon Phi. Left-hand side: Different thread affinities and numbers of threads used per offload. Right-hand side: Impact of data transfers between host and coprocessor. Threads (processes) are created either by means of OpenMP or MPI (right bottom corner of the sub-plots).

b). Increasing the number of threads per offload from 1 to 2, 3, and 4 results in significant performance gains, supporting the point that at least 2 threads per physical Xeon Phi core should be used [9]. However, in b) the 4-thread performance is behind that in a). We found using `OMP_PLACES=threads|cores` for concurrent offloads does not guarantee for a particular host thread (process) that its OpenMP threads on Xeon Phi are assigned cores with contiguous logical core IDs. We observed that it is more likely for b) to have all threads of the same group on different cores than it is for a). We assume the performance discrepancy between a) and b) in Fig. 4 is caused by unfortunate thread placements.

Sub-plots c) and d) show the performance obtained with by-hand thread pinning using the "scatter-compact" scheme. The 1-thread performance is identical to b). In the 4-thread case, the performance is measurably larger than in a), since all 4 threads within the same group execute on the same physical core. Cache-optimized kernels can benefit from sharing the L1-cache in this case. A comparison of c) and d) shows that MPI- and OpenMP-based executions perform almost equivalent for large matrices. We achieve $\approx$65% efficiency in these cases – the Xeon Phi 7120P provides about 2.4 TFlops/s single precision peak performance. Native Xeon Phi executions of SGEMM with larger matrices achieve about 86% efficiency [2]. The performance shown in sub-plots a) – d) is independent of whether LEO or OpenMP4 is used for the offload.

The right-hand side sub-plots display the performance impact of data transfers between successive offload computations. If the entire problem size is transferred, for all executions with multi-threading on the host the performance breaks down significantly. Although each host thread has a corresponding Xeon Phi thread linked by a COIPipeline (Coprocessor Offload Infrastructure) for kernel invocations and data transfers [3], concurrency across multiple pipelines suffers from the current COI implementation uses just one DMA channel. As a consequence, data transfers are serialized, possibly causing kernel executions be serialized too

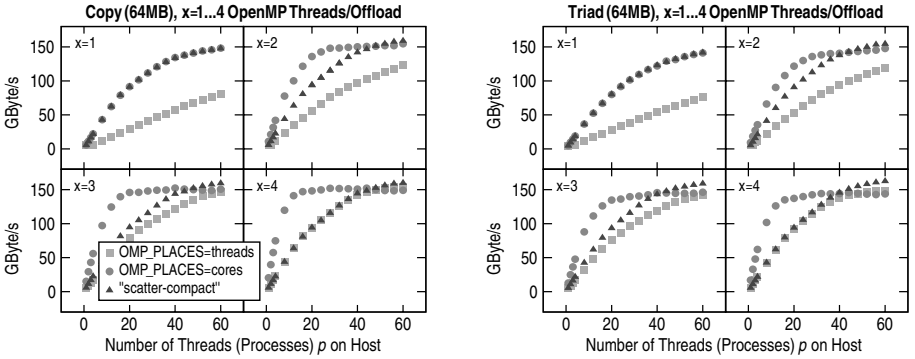

Fig. 5. STREAM copy and triad benchmark for different numbers p of concurrent host threads (processes) and $x = 1, \ldots, 4$ OpenMP threads on the Xeon Phi

(see the decrease of the in-kernel concurrency $\mathcal{C}_\tau$ for (M2) in Tab. 1). When using MPI the number of DMA channels equals the number of MPI ranks. The performance compared to no data transfers thus decreases only a little.

Sub-plot f) shows reduced performance when using OpenMP4 within libraries with frequent data transfers between host and Phi. Our offload-proxy approach recovers the performance achievable with LEO to almost 100% (Fig. 4, g).

STREAM Benchmark. The STREAM copy benchmark refers to `b[0:size]=a[0:size]`, while the STREAM triad is `c[0:size]=a[0:size]+q*b[0:size]`. We aim to measure the streaming performance that can be achieved when accessing main memory from within concurrent memory bound kernels. The streaming performance for different thread affinities and different numbers of OpenMP threads is shown in Fig. 5 – array size: 64 MB. Host threads were created using OpenMP. For kernel offloading Intel LEO was used. Performance results for OpenMP4 and MPI are almost identical as the host just initiates the offloads.

Using `OMP_PLACES=cores` the Phi's physical cores are populated faster than with `OMP_PLACES=threads`. Hence, the streaming performance is higher for both copy and triad. The "scatter-compact" thread pinning scheme gives the same performance as `OMP_PLACES=cores` if a single thread is used per offload. With 60 host threads (processes) and two OpenMP threads per offload, the streaming performance starts to saturate at about 162 GB/s, which is close to the value of 174 GB/s (ECC enabled) for native Phi execution of STREAM triad by 93% [2].

5 Strong Scaling for Simulations of Small Molecules

The program package GLAT (Global Local Adaptive Thermodynamics) overcomes the problem of critical slowing down of conventional thermodynamical simulations by decomposing the conformational space into metastable subregions, which can be investigated almost independently. The current paper

addresses a typical question of pharmaceutical or biochemical applications: The prediction of solvation for a conformational ensemble.

Even small drug-like molecules with < 50 atoms can exhibit more than 100 metastable states. The sampling of such molecules in water environment requires the explicit modeling of a solvation shell, containing at least one order of magnitude more atoms than the "internal molecule." To achieve strong scaling for simulations on these small molecules, GLAT performs almost independent Hybrid Monte Carlo (HMC) samplings of the water solvation for many metastable states concurrently. HMC is a combination of short term Molecular Dynamics (MD) followed by a Monte Carlo (MC) weighting of the generated conformations with respect to the total energy. The calculation of the contributions of the solvent to energy and forces, is transferred to Xeon Phi. The data for the water environment remains on the Phi, whereas the forces of the water on the internal molecule, as well as the potential/kinetic energy of the water are sent back to the host for the HMC step.

Figure 6 illustrates the workflow of a simulation within one metastable state: First the simulation is initialized with coordinates, velocities, and force-field parameters of the internal molecule in a given metastable conformation. Followed by an automatic modeling and minimization of the water environment, the result is a water droplet containing the molecule of interest. Then the water data as well as the positions of the internal molecule are transferred to the Phi where the calculation of the covalent contributions and the forces of the water on the internal molecule is started. Meanwhile the host carries out the force calculation of the internal molecule with itself. At the following barrier the host receives the forces on the internal molecule, completes the MD step, and copies the updated coordinates of the internal molecule to the Phi, whereas the coprocessor calculates the water-water interactions and performs the MD step for the water. Since the whole simulation is embedded into an HMC scheme, the host performs some statistical weightings after a sequence of ≈ 10 MD steps. The HMC sampling is repeated for a given water environment several times until about 10^3 MD steps are reached. The final convergence check will either finish the simulation or restart it with another randomly created water environment.

- The GLAT core is written in Fortran, whereas the coprocessor portion of the code is encapsulated into a C++ library. We introduced the possibility to fall back to the CPU when calling the library. For both Xeon Phi and CPU, kernels have been optimized using SIMD intrinsics.

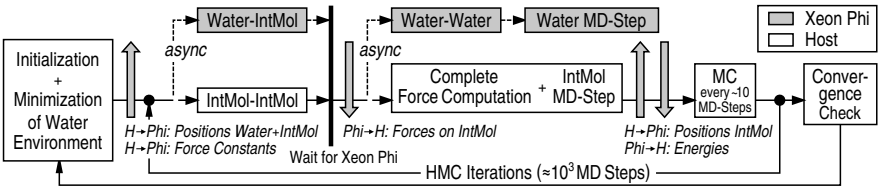

Fig. 6. Workflow of GLAT. The schematic displays an entire simulation cycle.

Benchmarking Setup and Methodology. We consider three different sized problems: An internal molecule consisting of 27 atoms embedded into a water droplet containing 101 (P1), 302 (P2), and 505 (P3) molecules. As a performance measure we determine the number of particle-particle interactions per second for (A) the MD loop only, and (B) an entire simulation cycle including the water minimization, the HMC step, and the final convergence check. Runtimes for 2000 iteration steps are measured using `clock_gettime(CLOCK_REALTIME,..)`.

For each setup we use OpenMP4, with and without our offload-proxy approach (Sec. 3), and Intel LEO for coprocessor offloading. Concurrency on the host is achieved by means of multiple OpenMP threads and/or MPI processes, each of which creating a Markov chain throughout the HMC sampling, and offloading kernels to the Phi to speed up force computations.

The system used for benchmarking is described in Sec. 4. It provides 16 physical respectively 32 logical CPU cores. When using a single Xeon Phi, we create $1 \ldots 16$ concurrent host threads (processes). On Xeon Phi we use 15 OpenMP threads per offload computation for a total of up to 240 threads – we use 60 out of 61 physical cores (see Sec. 4). Computations with two Xeon Phis use either $2 \ldots 32$ MPI ranks on the host, or two multi-threaded MPI ranks with up to 16 OpenMP threads per rank. When redirecting the offload to the host – CPU-only computation –, two OpenMP threads are used for kernel execution.

Benchmarking Results. The benchmarking results are displayed in Fig. 7. For each sub-plot the left hand side graphics are for the MD loop only, whereas the right hand side ones are for an entire simulation cycle – the performance on the right thus is lower. In all cases larger values are better.

Throughout all sub-plots the OpenMP4 performance, when not using our offload-proxy approach, is significantly behind the others due to data transfers (Sec. 3). The performance loss can be compensated to a certain extent with our proxy approach. However, it is below the one obtained with LEO as our offload-proxy performs busy-waiting during the OpenMP4 offloads, and hence consumes CPU resources on the host. Since best performance values can be achieved with LEO, the implementation of the entire simulation uses LEO. The right hand side sub-plots in Fig. 7 thus do not contain data for OpenMP4.

Using MPI on the host can result in measurable performance gains over using OpenMP if problems are small, e.g. (P1) and (P2). With OpenMP concurrent data transfers suffer from just one DMA channel is used by the current COI implementation, causing serialization of data movements between Xeon Phi and host. In case of small problem sizes, where kernel execution times are of the same order as the associated data transfer times and the offload overhead, serialization of data transfers implicitly serializes kernel executions (Tab. 1).

Executions using two Xeon Phis achieve almost twice the overall performance compared to single-Phi executions if the setup becomes large – e.g. (P3). Best performance in these cases can be obtained with a hybrid MPI/OpenMP approach on the host. However, with significantly more than 16 host threads (processes) concurrency on the host, and hence on the Phi, suffers from contention due to oversubscription of CPU resources. It thus would be meaningful to extend

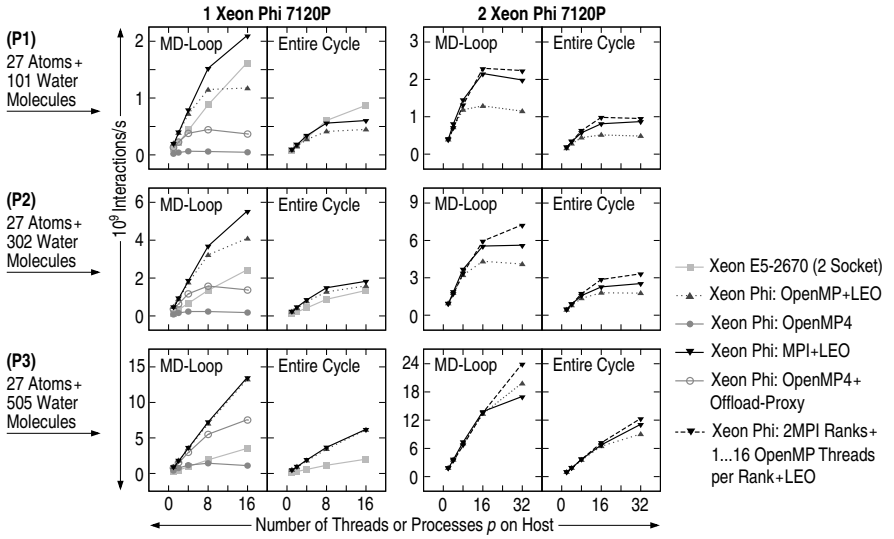

Fig. 7. Particle-particle interaction rates obtained with GLAT for three different sized problems (P1) – (P3) (see the text). Each host thread (process) offloads computations to Xeon Phi via OpenMP4 or LEO. Note the different scales. Larger values are better.

the computation across more than one compute node. Since the current offload models can use coprocessors within the same node only, hybrid approaches like MPI+X are necessary in this case. For GLAT the offloads to Xeon Phi are independent of each other and thus not affected by MPI traffic.

Since GLAT draws on a legacy Fortran code base containing a non-negligible amount of sections that are not highly parallel, comparing our results against native Xeon Phi execution of GLAT (as a whole) would suffer from insufficient performance of its serial parts and the low performance of the internal-molecule kernels.

6 Summary and Conclusion

In this work we investigated the performance of multi-threaded/-process applications with concurrent offloading of many small-scale computational kernels to Xeon Phi. We evaluated the two offload models Intel LEO and OpenMP4 including our offload-proxy approach. For a small synthetic compute bound kernel performing an SGEMM computation, we achieved a high degree of concurrency with up to 60 host threads (processes) offloading to the Phi. For scenarios without data transfers LEO and OpenMP4 perform equally. We observed deficiencies in the OpenMP4 offload model regarding data persistence across different function scopes, limiting its usability within libraries offloading computations to a coprocessor. To partly compensate for this issue, we proposed and evaluated an offload-proxy approach. For a real-world application implementing a simulation

of drug-like molecules solvated within a nanodroplet, we demonstrated its viability. By using OpenMP4 respectively LEO offloading to Xeon Phi speedups of about a factor 2 – 3 over an optimized and parallelized CPU implementation could be achieved.

Acknowledgments. The Intel Xeon Phi nodes are kindly donated by Intel. The authors would like to thank Michael Klemm and Chris J. Newburn (both Intel Corp.) for in-depth discussions of Xeon Phi specifics. This work was partly supported by the Deutsche Forschungsgemeinschaft (DFG), Priority Program "Software for Exascale Computing" (SPP-EXA), DFG-SPP 1648, project FFMK (Fast Fault-tolerant Microkernel), and by Intel Corp. within the "Intel Parallel Computing Centers" initiative.

References

1. Hwu, W.M.W.: GPU Computing Gems Jade Edition, 1st edn. Morgan Kaufmann Publishers Inc., San Francisco (2011)
2. Intel Corporation: Intel Xeon Phi Product Family Performance, rev. 1.0. (December 2012), `http://www.intel.com/performance`
3. Newburn, C.J., Dmitriev, S., Narayanaswamy, R., Wiegert, J., Murty, R., Chinchilla, F., Deodhar, R., McGuire, R.: Offload Compiler Runtime for the Intel Xeon Phi Coprocessor. In: IPDPS Workshops, pp. 1213–1225. IEEE Computer Society (2013)
4. Johnson, J., Krieder, S.J., Grimmer, B., Wozniak, J.M., Wilde, M., Raicu, I.: Understanding the Costs of Many-Task Computing Workloads on Intel Xeon Phi Coprocessors. In: 2nd Greater Chicago Area System Research Workshop (GCASR). Northwestern University, Evanston (2013)
5. Pennycook, S.J., Hughes, C.J., Smelyanskiy, M., Jarvis, S.A.: Exploring SIMD for Molecular Dynamics Using Intel Xeon Processors and Intel Xeon Phi Coprocessors. In: IEEE International Parallel & Distributed Processing Symposium, pp. 1085–1097. IEEE Computer Society, Los Alamitos (2013)
6. Wang, L., Huang, M., El-Ghazawi, T.: Towards Efficient GPU Sharing on Multicore Processors. In: Proceedings of the 2nd International Workshop on Performance Modeling, Benchmarking and Simulation of HPC Systems, PMBS 2011, pp. 23–24. ACM, New York (2011)
7. Wende, F., Cordes, F., Steinke, T.: On Improving the Performance of Multi-threaded CUDA Applications with Concurrent Kernel Execution by Kernel Reordering. In: Proceedings of the 2012 Symposium on Application Accelerators in High Performance Computing, SAAHPC 2012, pp. 74–83. IEEE Computer Society, Washington, DC (2012)
8. Wende, F., Cordes, F., Steinke, T.: Multi-threaded Kernel Offloading to GPGPU using Hyper-Q on Kepler Architecture. Technical Report 14-19, ZIB, Takustr. 7, 14195 Berlin (June 2014)
9. Jeffers, J., Reinders, J.: Intel Xeon Phi Coprocessor High Performance Programming, 1st edn. Morgan Kaufmann Publishers Inc., San Francisco (2013)
10. OpenMP Architecture Review Board: OpenMP Application Program Interface, Version 4.0. 4.0 edn. (July 2013), `http://www.openmp.org`