The Open Community Runtime on the Intel Knights Landing Architecture

Jiri Dokulil[1], Siegfried Benkner[1] and Jakub Yaghob[2]

[1] Faculty of Computer Science, University of Vienna, Vienna, Austria
jiri.dokulil, siegfried.benkner@univie.ac.at
[2] Department of Software Engineering, Charles University, Prague, Czech Republic
yaghob@ksi.mff.cuni.cz

Abstract. The Intel Xeon Phi Knights Landing manycore processor comes with new interesting features: on-chip high-bandwidth memory and several user-selectable NUMA configurations. In this paper, we look into how these affect applications that target the Open Community Runtime (OCR), an asynchronous tasked-based runtime system for future parallel architectures. We have extended our OCR runtime to make it NUMA aware and to allow it to use the high-bandwidth memory. We have conducted a range of experiments, comparing OpenMP, TBB, our OCR implementation, and the reference OCR implementation on different machine configurations using a memory intensive seismic simulation.

Keywords: Open Community Runtime, Knights Landing, Intel Xeon Phi, high-bandwidth memory, parallel runtime systems, NUMA

1 Introduction

Given the ever increasing complexity and architectural variety of parallel systems, asynchronous task-based programming systems have gained a lot of momentum since they facilitate decoupling the specification of parallelism from its actual implementation. With a task-based programming model the user exposes the potential parallelism by decomposing a problem into tasks with well-defined inputs and outputs and delegates to the runtime system how tasks are scheduled for parallel execution to the available hardware resources. Such an approach not only allows a program to be dynamically adapted to a specific architecture, but also to reorganize parallel execution in case of changing workloads or varying hardware performance characteristics.

The Open Community Runtime (OCR) [11] is an open specification of a low-level runtime system for future extreme-scale architectures. OCR has been developed in the context of the US X-Stack program in order to provide a common foundation for research on runtime systems and higher-level parallel programming models. OCR relies on an event-driven, asynchronous task-based model where the potential parallelism available in an application is expressed by a large number of tasks (referred to as event driven tasks) with explicit dependences. All data is organized in data blocks, which are relocatable, contiguous

chunks of memory managed by the OCR runtime. Coordination and synchronization is expressed by means of events, which are used to establish control and data dependences.

A task can only access data in a data block if the data block was explicitly passed to it or if it created the data block. As a result, the runtime is aware of all the data a task can possibly access. Once all data blocks a task depends on are available and all events are satisfied, a task becomes *runnable* and will eventually be executed on some execution unit and run to completion regardless of the behavior of other tasks.

At runtime an OCR program forms a dynamically created directed acyclic graph (DAG) of tasks, with explicit information about which data blocks are used (consumed and produced) by which tasks. This gives the runtime system the ability to relocate tasks and data blocks to achieve a better load balance, to optimize memory and energy consumption, and to deal with failures.

In this paper, we report on the extensions of OCR-Vx (our implementation of the OCR specification [6]) for the new Intel Xeon Phi Knights Landing processor (KNL). Our contributions are: First, a NUMA-aware task scheduler for OCR-Vx, which is also beneficial on other NUMA machines, not just the KNL. Second, enabling OCR-Vx to use the on-chip high-bandwidth MCDRAM of the KNL. Finally, we have evaluated the performance of the runtime on six different configurations of the KNL, using a seismic simulation code. To provide a wider context, the same code was tested with the reference OCR implementation [10] and rewritten in OpenMP. We also compare our OCR and OpenMP codes to the original native TBB application. In total, five application variants were tested.

This paper is organized as follows: Section 2 briefly presents the new features of the KNL architecture. Section 3 describes relevant components of our OCR implementation and the extensions to support NUMA systems and high-bandwidth memory. Experimental results are provided in Section 4. Section 5 discusses related work followed by a conclusion and discussion of future work.

2 Knights Landing architecture

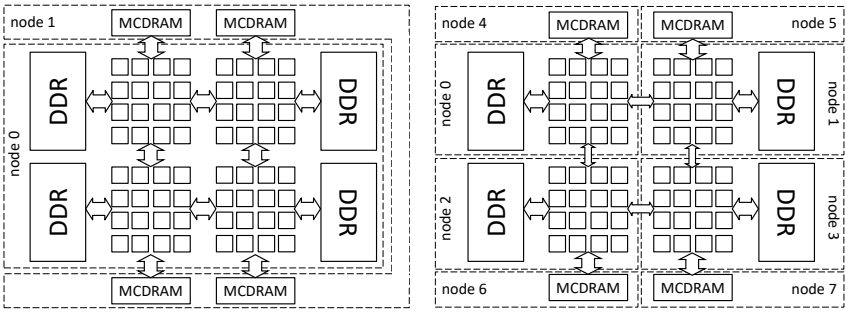
Knights Landing is the latest release in the Intel Xeon Phi product range. Formerly known as MIC (Many Integrated Cores), Xeon Phi is a family of high-performance many-core architectures. Unlike the previous architecture, the Knights Corner (KNC), which was available as a coprocessor PCIe card, the KNL is a "full" CPU, which is used as the only processor of a server. In our experiments, we use a machine with one Intel Xeon Phi Processor 7230, which has 64 cores, four way hyperthreading (256 threads), it runs at 1.30 GHz, and it has 16 GB of on-chip high-bandwidth MCDRAM. The main memory (DDR4) is 96 GB. Each core has 32 KB data cache and 32 KB instruction cache and there are two vector processing units with AVX-512 support attached to each core. The cores are arranged into tiles, with each tile containing two cores and 1 MB L2 cache shared by the two cores. The tiles are laid out as a 2D mesh. Since 64 cores occupy 32 tiles, there is 32 MB of L2 cache in total.

The whole processor is fully cache coherent. To maintain cache coherency, it has a distributed tag directory, organized as a set of per-tile tag directories (TDs), which identify the state and the location on the chip of any cache line. For any memory address, the hardware can identify (using a hash function) the TD responsible for that address. If there is a cache miss, the tile where the cache miss occurred must send a message to the tile with the TD corresponding to the accessed memory address. Depending on whether the cache line is cached somewhere (this is determined from the TD), the message is forwarded to the tile with the cached data or to the memory. To support different workloads, the KNL chip can be configured (in BIOS) to use different ways of organizing the tag directories. These are called *clustering modes*.

With the *all-to-all* clustering mode, memory addresses are uniformly distributed across all TDs on the chip. As a result, there is a high probability that when a tag directory needs to be used (e.g., a cache miss), the corresponding tag directory may be far away from the core that needs it, causing a high latency of the cache operation. In the *quadrant* clustering mode, the tiles are divided into four parts called quadrants, which are spatially local to four groups of memory controllers. Memory addresses served by a memory controller in a quadrant are guaranteed to be mapped only to TDs contained in that quadrant. On average, this provides a much lower latency than in the all-to-all mode. In *hemisphere* mode, two groups are used instead of four. The division of the cores is hidden from the operating system and the whole system is presented as a single NUMA node, except for the MCDRAM, which is shown as one separate NUMA node. The *SNC-4* (Sub-NUMA Clustering) mode uses the same tile partitioning as the quadrant mode, but each quadrant is exposed as a separate NUMA node. The MCDRAM is also split into four NUMA nodes. The *SNC-2* mode is the same as SNC-4, except that only two groups of cores are used. If cache traffic crosses the boundary between two NUMA nodes in SNC-2/4, it is more expensive than in the corresponding hemisphere/quadrant mode.

In our experiments, we only use the quadrant and SNC-4 modes. The all-to-all mode should only be used as a failsafe and not in production. We did some tests with the hemisphere and SNC-2 modes, but they did not bring any new interesting results, so we will only present the numbers for quadrant and SNC-4.

Another configuration option is the usage model of the MCDRAM. There are three options, independent of the clustering modes. First, the MCDRAM can be used in the *flat* mode, where it is available as a separate NUMA node (or nodes, depending on the clustering mode), which can be used by the application to allocate memory. MCDRAM and the machine's main memory (DDR) share the same physical address space. Aside from possibly different memory allocation calls, both memory types can be used by the application in exactly the same way. Another mode for the MCDRAM is the *cache* mode. In that case, the MCDRAM is used as a last level cache for the main memory. This cache is completely transparent to software. The last available mode is the *hybrid* mode, where part of the MCDRAM is available in the flat mode and the rest is used as cache. The core organization and NUMA nodes for the flat MCDRAM mode



(a) Quadrant (b) SNC-4

Fig. 1. KNL clustering modes. Displayed are cores, memory (MCDRAM and DDR), and NUMA nodes. Both configurations are shown with MCDRAM switched to flat mode. In cache modes, the NUMA nodes with the MCDRAM (node 1 in quadrant mode and nodes 4–7 in SNC-4 mode) would not be present.

with quadrant and SNC-4 clustering modes are shown in Figure 1. Note that the MCDRAM NUMA nodes don't contain any cores, only memory.

In the flat mode, the MCDRAM can be allocated using the `hbw_malloc` call, which is part of the *memkind* library. Since the MCDRAM is available as one or more separate NUMA nodes (node 1 in quadrant, nodes 4–7 in SNC-4), it is also possible to use `numa_alloc_onnode`, which is part of *libnuma*. In SNC-4 mode, `numa_alloc_onnode` can be used to directly specify which of the four MCDRAM nodes to use, giving the application more control. This is not possible with `hbw_malloc`.

3 OCR-Vx

OCR-Vx is a collection of open source[1] OCR implementations that we created at the University of Vienna [6]. For this paper, the relevant implementation is OCR-Vsm, which is a shared-memory implementation. Originally, it was built on top of the task scheduler from Intel Threading Building Blocks (TBB).The latest version can be configured to use different schedulers (see Section 3.1 for details). OCR-Vsm is similar to OCR-Vdm, the distributed-memory implementation, but it uses several optimizations, which are much easier to do in a shared memory environment and we have yet not implemented them in the distributed runtime.

Eventually, the optimized shared-memory runtime will be used within the distributed runtime, to control a single node. This approach is already being used by the XSOCR runtime [10], which was created by Intel and Rice University as part of the XStack project (hence the "XS" in the name) as a reference implementation of the OCR specification.

[1] Stable releases of the runtime are available at `http://www.univie.ac.at/ocr-vx/`

3.1 NUMA support

Most of today's high performance computing systems can be characterized as Non-Uniform Memory Architectures (NUMA). In NUMA systems, the different CPU cores don't have uniform access to the system memory. Usually, different parts of memory are "closer" to certain cores, giving them faster access compared to the rest of the memory. Overall, a NUMA machine consists of several NUMA nodes. A NUMA node contains zero or more CPU cores and some portion of the total system memory. In a typical four socket NUMA server, there are four NUMA nodes, each containing cores from one of the four CPUs and a quarter of the main memory. The KNL configured as SNC-4 with MCDRAM as cache looks the same way.

In OCR, task and data placement is handled by the runtime. The OCR API provides a way to influence the placement using affinities. An affinity is an OCR abstraction which represents the hardware architecture. The OCR specification leaves the actual organization of affinities to the runtime implementation. It only provides a way to list all affinities, to get the affinity of the current task, and to set an affinity for data blocks and tasks. The runtime is also allowed to ignore any affinity specified by the application. A common solution is to make the affinities correspond to machines in a cluster. This way, if a task has the same affinity as a data block, it should have direct access to the data block's data. If two tasks share an affinity, they should be able to efficiently share data. The data and task placement within the affinity (the machine) is completely handled by the runtime.

An alternative is to use finer grained affinities. We have modified our runtime to provide one affinity per NUMA node, making the application more involved in data and task placement even within a node. Even a single machine may now have multiple affinities, making it look more like a distributed-memory system from the application's point of view. The OCR runtime uses the `hwloc` library to explore the hardware architecture and determine the number of affinities to provide.

However, the TBB task scheduler which we used within our OCR implementation cannot be used to execute tasks on a specified NUMA node. Thus, we have created our own NUMA-aware scheduler, which uses the same task-stealing principles as the TBB scheduler, but the threads are split into several groups corresponding to NUMA nodes. For example, in a four socket server capable of supporting 16 threads per core, the runtime uses 64 threads split into four groups with 16 threads per group. Each group is pinned (using OS thread affinities) to a different NUMA node. Task stealing is only done within a NUMA node, not across the boundaries. Therefore, if a task is spawned to a NUMA node, it is guaranteed to be executed by that NUMA node.

The same approach applies also to the KNL. If 128 worker threads are used on KNL configured as SNC-4, there will be four groups with 32 threads each. Naturally, only the four NUMA nodes that contain the cores are used, not the NUMA nodes with MCDRAM and zero cores.

3.2 High-bandwidth memory support

Another issue is the high-bandwidth MCDRAM memory of the KNL. If it's configured as cache, no changes to the code are necessary. But to use the MCDRAM in flat mode, it is necessary to change the way memory is allocated by the application if the memory needs to be placed in the MCDRAM. Two options are available in that case. First, the `hbw_malloc` call can be used instead of the normal `malloc` function to allocate data in the MCDRAM. The way this data is placed in the MCDRAM (in which of the four MCDRAM nodes) is determined by the OS. The second alternative is to use `numa_alloc_onnode` and specify one of the NUMA nodes that correspond to the MCDRAM. In SNC-4 mode, this allows the application to specify which of the four nodes to use, giving it more control. Note that `numa_alloc_onnode` could also be used as an alternative to `malloc` to allocate data in a specific part of the DDR memory. In SNC-4 mode, allocating data on nodes 0 to 3 results in the data being placed in DDR, while allocating on nodes 4 to 7 places it in MCDRAM.

The modified OCR-Vsm runtime can be configured in four different ways: to use `malloc`, to use `hbw_malloc`, to use `numa_alloc_onnode` to place data in DDR, and to use to use `numa_alloc_onnode` to place data in MCDRAM. If the NUMA allocator is used to target DDR, the data is placed to the NUMA node that corresponds to the affinity provided by the user when the data block is created. If no affinity is given, it's placed in the NUMA node local to the task that created the data block. If the NUMA allocator targets MCDRAM, the placement algorithm is the same, except it places the data in the adjacent MCDRAM. For example, in SNC-4 mode, it uses NUMA node 4 instead of 0, 5 instead of 1, etc.

4 Experimental evaluation

We have used the Seismic application, which was also used in our earlier work evaluating OCR on KNC, the previous Xeon Phi architecture [5]. However, the application was substantially updated, to better reflect the task nature of OCR.

4.1 Seismic application

The original Seismic application is distributed as an example with the TBB library. It simulates propagation of seismic waves through 2D terrain. There are several properties associated with each grid point (stress, velocity, dampening, etc.). All values are stored as double precision floating point numbers and they are always processed in double precision. For our earlier experiments on the KNC, we modified the code to make it more computationally intense. No such modification was used this time. The code performs a comparable number of arithmetic operations and memory (load/store) operations. This makes the performance of the memory subsystem much more important. At the same time, the smaller amount of computation performed by each iteration makes any runtime overhead more pronounced.

```
for(int it = 0; it < iterations; ++it) {
  parallel_for(int y = 0; y < height; ++y) {
    vectorized_for(int x = 0; x < width; ++x) {
      update_stress(x,y);
    }
  }
  parallel_for(int y = 0; y < height; ++y) {
    vectorized_for(int x = 0; x < width; ++x) {
      update_velocity(x,y);
    }
  }
}
```

Listing 1: Pseudo-code showing the common structure of the TBB and OpenMP variants of Seismic.

Seismic runs in several iterations and each iteration comprises two phases. First, horizontal and vertical stress is updated for each grid point based on values of properties (other than the stresses) of the point and its neighbors to the right and below. Second, the seismic wave velocity is updated for each point based on properties (not including the velocity) of the point and its neighbors to the left and above. There are no dependences within a phase, just between phases. These dependences are limited only to dependences between neighbors. We have created three implementation variants of the Seismic code: using TBB, OpenMP, and OCR.

The TBB version is the code distributed as an example with the TBB library, with minimal modifications required to make it usable as a benchmark. The original Seismic is an interactive graphical application, with fixed problem size. We have removed the GUI code and put the simulation iterations into a simple for loop. We also modified the data structures to make the problem size configurable at runtime.

The OpenMP variant is very similar to the TBB code. The two phases of the simulation are performed by a parallel for loop. Although a different OpenMP parallelization strategy might be more efficient on the KNL, especially in the SNC-4 mode, we decided to use parallel for loops, because our goal is to investigate how common parallelization techniques compare to their OCR alternative.

In both codes, there are three levels of nested for loops. The outermost loop advances the iterations of the simulation. The middle loop iterates over the y-dimension. It is repeated two times, one after the other, doing the two phases in sequence. These two middle loops are the ones being parallelized using TBB parallel algorithms and OpenMP parallel for. The innermost loop (present in both middle loops) iterates over the x-dimension. Hints are provided for compiler auto-vectorization, so that the innermost loop is vectorized, i.e., SIMD instructions are used to perform several consecutive iterations of the loop together. The basic pseudo-code outline of the algorithm is shown in Listing 1.

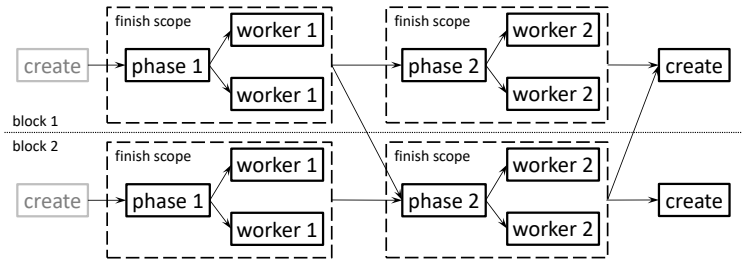


Fig. 2. An example of tasks (boxes) and their dependences (arrows) within one iteration of the Seismic on OCR, with data split into two blocks and with two tasks working on each block in every phase in parallel.

Due to the nature of the OCR programming model, the OCR code needs to be structured differently. To allow for distributed execution, the data is split into smaller blocks, each containing the same number of (consecutive) rows. These are distributed evenly across the available OCR affinities. Initially, one task is created for each block. This task is responsible for generating all tasks that are needed to perform one iteration step on the data block. If it is not the last iteration, another copy of this generator task is added at the end of the iteration. All tasks that process a certain block of data are bound (using OCR affinities) to the compute node where the data is located. The tasks that process neighboring blocks are synchronized using channel events, a new experimental extension of the OCR specification, which simplifies implementation of synchronization where the same pattern is repeated multiple times.

An example of the tasks and their dependences for one iteration is shown in Figure 2. In the example, the data is split into two blocks. The tasks are labeled with their type, which corresponds to the C function they execute as their body. There are two phases per iteration and two worker tasks are used per block in each phase. The number of blocks and the number of tasks per block is configurable.

The `create` task creates all of the following tasks, up to (and including) the next `create`. A task called `phase 1` creates two worker tasks (of the `worker 1` type) to process the data block in parallel. The `phase 1` task is a *finish* task, which means that any further tasks it creates (denoted by the *finish scope* boxes) have to finish before the original finish task considered to be finished. Therefore, the `phase 2` task, which depends on the `phase 1` task, cannot start before both worker tasks finish. For this reason, the dependence is drawn going from the finish scope to the `phase 2` task. The dependences going across the block boundary (the vertical line in the middle of the picture) are used to exchange data and synchronize tasks that process neighboring blocks.

In the TBB and OpenMP codes, there is an (implicit) barrier at the end of each phase. There is no such barrier in the OCR code, except for the very end of the computation. This allows some tasks from the next phase to start

before all of the tasks from the current phase have finished, which is not the case with the more coarsely synchronized TBB and OpenMP codes. Of course, the fine-grain synchronization could also be used with the TBB and OpenMP codes. While this style of programming is a natural fit for the OCR code, the high-level parallel for (`pragma` in OpenMP, parallel algorithm in TBB) is the first choice in OpenMP and TBB. They are also much easier to use. In TBB, the `parallel_for` algorithm is transformed into tasks internally, in a very efficient way. It would be difficult to achieve the same performance using the basic TBB task interface, being comparably difficult to writing the OCR application. Even though code size is far from an accurate code complexity metric, it may be interesting to compare them in this case. The OpenMP code is the smallest, at just around 6kbytes (200 lines), followed by the TBB code at 10kbytes (370 lines). The OCR code is much larger – 36kbytes (1000 lines).

4.2 Application and runtime configuration

The applications provide several configuration options, depending on the implementation variant. The size of the data and number of iterations can be configured in all cases. We have used 8192×8192 as the data size, which translates to memory footprint of around 3 GB. With 1000 iterations, this provides a good reasonably long execution times (tens of seconds), but the data still easily fits into the MCDRAM, which we needed for some of the experiments. All source codes were compiled with GCC 6.3, with AVX-512 instruction set enabled.

The OpenMP runtime used in the OpenMP variant can be configured using environment variables. We have experimented with different thread placement, but the default option turned out to be the most efficient, so only the number of threads is adjusted by setting `OMP_NUM_THREADS`. Using 128 threads provided the fastest execution times. The task-based runtimes (TBB and all OCR variants) also performed best with 128 threads, except for the XSOCR, which performed best with 256 threads.

The TBB variant uses automatic task granularity selection provided by the `parallel_for` construct. The task granularity of the OCR Seismic code can be configured using command line parameters. For OCR-Vsm, the data was split into 128 equally-sized horizontal blocks and there was one task per block in each phase. In total, there are worker 128 tasks per phase. For XSOCR, we used 256 tasks per phase and 256 blocks. The task and block counts were obtained by manual tuning, where the relevant search space was exhaustively searched. The 128 and 256 tasks provided the best performance on the respective runtimes. Other configurations consistently resulted in a significant performance degradation (>30%).

The OpenMP and TBB applications can be configured to either use `malloc` or `hbw_malloc`. `malloc` is used in cache mode and in flat mode to avoid using the MCDRAM. `hbw_malloc` is used only in flat mode, to allocate application data in MCDRAM. The OCR-Vsm runtime can be configured in several ways. It's possible to either use the TBB scheduler or the new NUMA scheduler. The internal structures of the runtime are always allocated using `malloc`, but the

	quadrant			SNC-4		
	none	flat	cache	none	flat	cache
OpenMP	92.44	20.69	17.38	100.01	80.94	25.56
TBB	92.72	17.32	18.53	110.23	78.80	40.24
OCR-Vsm TBB	93.83	20.93	19.33	95.29	34.04	25.48
OCR-Vsm NUMA	93.56	20.76	19.42	92.27	19.53	20.75
XSOCR	98.66	20.68	19.19	99.08	20.66	31.66

Table 1. The execution time in seconds for 1000 iterations of the Seismic application, using different clustering modes (quadrant and SNC-4) and MCDRAM usage models (none, flat, cache). Mode *none* means that the machine was switched to flat (explicit) MCDRAM mode, but the data was allocated in DDR, not MCDRAM.

data blocks (the application data) can be allocated using `malloc`, `hbw_malloc`, or `numa_alloc_onnode`.

The fastest options turned out to be `malloc` and `hbw_malloc` (i.e., the same allocators as the ones used in OpenMP and native TBB) when combined with the TBB scheduler. In quadrant mode, this was also the best option for the NUMA scheduler. The only exception is the NUMA scheduler on SNC-4, where `numa_alloc_onnode` was used with all MCDRAM modes. In flat mode, it can either be used to allocate data in the nearest DDR NUMA node or nearest MCDRAM node. In cache mode, it always targets the nearest DDR NUMA node, but the data is also automatically cached in the nearest MCDRAM.

4.3 Results

An overview of the results obtained by executing 1000 iterations of the different variants of the Seismic application is shown in Table 1. Note that in all cases, only the actual computation is timed. The measurements exclude application setup, where the runtimes are started, memory for application data is allocated and the data is filled with initial values. However, it does include task creation in the task-based variants. This is not only fair, it is also unavoidable since the tasks are created on the fly, not upfront (which was the case with our earlier Seismic OCR code [5]).

As you can see, the fastest version is either the native TBB application on a KNL configured in quadrant cluster mode and explicitly using the MCDRAM to store the data, or the OpenMP code on a KNL in quadrant mode with the MCDRAM serving as an automatic cache. The difference is too small to declare a clear winner.

The results clearly show that it's essential to use the MCDRAM in codes like Seismic, which require high memory bandwidth. This is an expected result, but it's interesting to note the scale of the potential performance benefit. For example, the TBB variant is over 5.3x faster when `hbw_malloc` is used on the quadrant/flat configuration instead of plain `malloc`.

Using MCDRAM either directly or as a cache always provided some performance improvement. In most cases, the cache was more efficient than manual

	quadrant			SNC-4		
	none	flat	cache	none	flat	cache
OpenMP	0.13	2.38	2.09	5.28	2.82	10.46
TBB	0.43	1.35	2.72	3.16	10.56	7.84
OCR-Vsm TBB	0.34	1.08	1.56	2.36	38.02	10.93
OCR-Vsm NUMA	0.37	0.68	2.07	0.34	2.13	2.71
XSOCR	0.51	3.75	4.79	4.78	3.39	7.53

Table 2. The difference (in percent) between the slowest and fastest run. Each configuration was executed 10 times. A value of 0 would mean that every run had exactly the same execution time. A value of 100 would indicate that the longest run took double the time needed by the fastest run.

allocation, although there were several cases where it is the other way round (TBB on quadrant, OCR-Vsm NUMA on SNC-4, and XSOCR on SNC-4). The automatic and explicit MCDRAM management is not the only difference between the two modes. If the MCDRAM is used as cache, all memory accesses are cached, including data used by the runtime, like the memory used to store the tasks. OpenMP, TBB, and both OCR-Vsm variants only store the application data in MCDRAM. To make the XSOCR use the MCDRAM, we've changed the runtime source codes to make all allocations using `hbw_malloc`. This moves both application data and runtime data into the MCDRAM.

All application variants that use the MCDRAM run reasonably fast in the quadrant mode of the KNL. If we switch the KNL to SNC-4 this is no longer the case. It's important to note at this point that in SNC-4, the cost for memory accesses that cross the boundaries between the four core groups is significantly higher than in the quadrant mode. As a result, the codes that are not NUMA aware (all except for OCR-Vsm NUMA) suffer from a significant performance penalty. For example, the native TBB application is 2.2x slower in cache mode and 4.5x slower if `hbw_malloc` is used. OCR-Vsm NUMA provides similar results on quadrant and SNC-4, running even slightly faster on SNC-4 if MCDRAM is not used as cache. It's interesting to compare the performance of native TBB and OCR TBB on SNC-4/flat. The OCR-based code is 2.3x faster, despite using the same task scheduler. This shows that even a small change to the way data is allocated and the way tasks are created and submitted for execution can cause a significant difference in performance.

Each software/hardware configuration was executed 10 times. Table 2 shows the difference (in percent) between the slowest and fastest runs. If you look at the values for the TBB and OCR-Vsm TBB codes on SNC-4/flat, you can see a significant performance fluctuation (10.56% and 38.02%, respectively). This also suggests that a necessary condition for a good result in these cases is that the TBB scheduler and `numad` manage to line up correctly. In quadrant mode, no automatic memory movement is performed (all MCDRAM is just one NUMA node), and the results are stable. The differences are just 1.35% and 1.08%.

Another interesting observation was the effect of the number of threads on long term performance. All variants except XSOCR use 128 threads, since that

proved to be the fastest. The best performance for XSOCR was achieved with 256 threads. Both 128 and 256 threads map nicely to the 64 physical cores of the KNL. The 256 of XSOCR provide similar performance, but according to a temperature monitoring tool, they heat up faster, forcing the power management to throttle the cores down. Therefore, the computation with XSOCR slows down by a few percent after about half a minute of computation. The actual number varies by configuration; the highest was close to 10%. We have not observed similar slowdown with the other application variants.

To provide a comparison with a non-KNL system, we have also executed the experiments on a standard NUMA server. It has four Intel Xeon E-7 4820 (8 cores, 18 MB cache, 2.00 GHz) and 128 GB of memory. There is no MCDRAM and the system can only be viewed as having four NUMA nodes. Unlike the KNL, the CPU in this machine does not have AVX-512, but only SSE 4.2. The performance characteristics are similar to SNC-4. The NUMA-aware OCR is the fastest, native TBB the slowest. Even though the CPUs are not a new model (E-7 4820 was released in 2011), the performance advantage of the KNL is so significant (around 12x if MCDRAM is used, 2.5x without MCDRAM) that even if the CPU is upgraded to the top-end latest Xeon (around 10x increase in theoretical peak performance), the KNL may still be a strong competitor, with a good chance to win in a head-to-head comparison. It is important to note that the KNL server we have used is about half the price of the four socket Xeon server and a single top-range Broadwell price is comparable to the price of the 7230 KNL used in our experiments.

5 Related Work

In the following, we briefly discuss a few related task-based runtimes and programming systems. We are not yet aware of any such system that would be tuned specifically for the KNL. However, of the three main optimization directions (vectorization, NUMA-aware parallelism, and MCDRAM) two were already relevant on existing systems. Vectorization is usually relevant inside of the tasks, so the runtimes are mostly concerned with efficient NUMA-aware task scheduling.

StarPU [1] was among the first task-based runtime systems that specifically targeted single-node heterogeneous architectures comprised of CPUs and GPUs. StarPU uses the `hwloc` library to explore the machine architecture and it creates combined workers based on this topology. The workers are then used by different schedulers provided by StarPU. Since StarPU is already used with architectures that combine different memories (e.g., local memory of a GPU), it should be possible to extend it to also deal with the MCDRAM. The schedulers could then also explicitly move data from DDR to MCDRAM.

OmpSs [4] extends the OpenMP shared-memory programming model with directives for task-parallel programming. The underlying runtime (Nanos++) also relies on the `hwloc` library. The socket-aware scheduler uses this information to keep tasks local (inside a NUMA node). It can be configured not to steal tasks from other NUMA nodes (like our scheduler) or to steal from neighboring nodes.

ParalleX [8] is a parallel and distributed programming model around the concept of message-driven work-queues supporting fine-grained parallel execution through cooperative lightweight threads within a global address space. The HPX runtime system [9] is a C++-based implementation of ParalleX within an active global address space, which supports migration of objects between the nodes of clusters. HPX provides different schedulers and some of the schedulers can be configured to be *NUMA sensitive*, in which case they first try to steal work from the local NUMA node, before going to other nodes.

Legion [2] is a data-centric, task-based programming system that supports dynamic hierarchical data partitioning based on the concept of logical regions. Regions may be partitioned into sub-regions and may overlap. Tasks are bound to regions and may access regions with different privileges and subject to different coherence modes. Legion provides a mapping interface that enables programmers to control the mapping of tasks and data regions to a specific parallel architecture. This mapping allows processors to be combined into *processor groups* and let the whole group serve a single work-queue. Processor groups can be created to mirror the NUMA nodes. An implementation of Legion on top of OCR is being realized within the US X-Stack program.

PaRSEC [3] is a generic framework for task-scheduling on heterogeneous many-core architectures. PaRSEC relies on a symbolic representations of task graphs that can be enhanced with user-provided priorities and data/task mappings that can take into account the NUMA characteristics of an architecture.

The Bobox system [7] is also a parallel and distributed programming model, but it is focused on applications that are close to database query evaluation. Yet it also successfully employs a task-based runtime. Its task scheduler is also NUMA-aware and, like most of the other systems, it first tries to steal tasks locally, before stealing across NUMA node boundaries.

6 Conclusion and Future Work

We have evaluated different implementations of the Seismic application on the KNL. We managed to extract very good performance from the KNL, using all of the programming models and runtimes. Native TBB and OpenMP variants were the fastest, but they were followed closely by the three OCR variants. This is a good result for the OCR, since the Seismic application is almost a textbook example of an application which can be parallelized with OpenMP.

For most variants, we achieved better performance on the non-NUMA quadrant clustering mode. The NUMA-aware OCR-Vsm runtime was slightly faster on SNC-4 (except when MCDRAM was used as cache), but the other variants, which are NUMA-oblivious, suffered a significant performance penalty. In our experience, mapping the four NUMA nodes of the KNL to four OCR affinities and treating it like a distributed system with four nodes is a reasonable solution. Overall, it seems the non-NUMA clustering mode is a better choice, except for very well-tuned NUMA-aware codes.

For a memory intensive application like Seismic, the MCDRAM is critical for achieving good performance. However, that may not always be the case. In our case, the application data is way too large to fit into L1 and L2 caches, but small enough to fit into the MCDRAM. Using the MCDRAM as cache worked very well as a result. However, if the data is very small (and mostly fits into L1 and L2 caches) or larger than the available MCDRAM, the application could suffer from the higher latency of cache misses caused by the extra cache level.

In the future, we plan to extend our NUMA-aware scheduler for the distributed-memory OCR implementation and to investigate hierarchical scheduling approaches and automatic means for utilizing the MCDRAM.

References

1. 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; Euro-Par 2009 23, 187–198 (2011)
2. Bauer, M., Treichler, S., Slaughter, E., Aiken, A.: Legion: Expressing locality and independence with logical regions. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. pp. 66:1–66:11. SC '12, IEEE Computer Society Press, Los Alamitos, CA, USA (2012)
3. Bosilca, G., Bouteiller, A., Danalis, A., Faverge, M., Herault, T., Lemariner, P., Dongarra, J.: PaRSEC: Exploiting heterogeneity to enhance scalability. IEEE Computing in Science and Engineering 15(6), 36–45 (2013)
4. Bueno, J., Planas, J., Duran, A., Badia, R., Martorell, X., Ayguade, E., Labarta, J.: Productive programming of GPU clusters with OmpSs. In: IPDPS 2012 Parallel Distributed Processing Symposium (2012)
5. Dokulil, J., Benkner, S.: Retargeting of the Open Community Runtime to Intel Xeon Phi. In: International Conference On Computational Science, ICCS 2015. pp. 1453–1462. Procedia Computer Science (2015)
6. Dokulil, J., Sandrieser, M., Benkner, S.: OCR-Vx - an alternative implementation of the Open Community Runtime. In: International Workshop on Runtime Systems for Extreme Scale Programming Models and Architectures, in conjunction with SC15. Austin, Texas (2015)
7. Falt, Z., Krulis, M., Bednarek, D., Yaghob, J., Zavoral, F.: Towards efficient locality aware parallel data stream processing. Journal of Universal Computer Science 21(6), 816–841 (2015)
8. Hartmut, K., Brodowicz, M., Sterling, T.: Parallex an advanced parallel execution model for scaling-impaired applications. In: Proceedings of the 2009 International Conference on Parallel Processing Workshops (ICPPW '09). pp. 94–401 (2009)
9. Kaiser, H., Heller, T., Adelstein-Lelbach, B., Serio, A., Fey, D.: HPX - a task based programming model in a global address space. In: The 8th International Conference on Partitioned Global Address Space Programming Models (PGAS) (2014)
10. Mattson, T.G., et al.: The Open Community Runtime: A runtime system for extreme scale computing. In: 2016 IEEE High Performance Extreme Computing Conference (HPEC). pp. 1–7 (2016)
11. Mattson, T., Cledat, R. (eds.): The Open Community Runtime Interface (April 2016), `https://xstack.exascale-tech.com/git/public?p=ocr.git;a=blob;f=ocr/spec/ocr-1.1.0.pdf`