Utilizing Multiple Xeon Phi Coprocessors on One Compute Node

Xinnan Dong[1], Jun Chai[1], Jing Yang[1], Mei Wen[1], Nan Wu[1], Xing Cai[2,3], Chunyuan Zhang[1], and Zhaoyun Chen[1]

[1] School of Computer Science, National University of Defense Technology
Changsha, Hunan 410073, China
xinnandong@126.com,
{chaijun200306,estella,meiwen,nanwu,cyzhang,chenzhaoyun}@nudt.edu.cn
[2] Simula Research Laboratory
P.O. Box 134, 1325 Lyakser, Norway
xingca@simula.no
[3] Department of Informatics, University of Oslo.
P.O. Box 1080 Blindern, 0316 Oslo, Norway

Abstract. Future exascale systems are expected to adopt compute nodes that incorporate many accelerators. This paper thus investigates the topic of programming multiple Xeon Phi coprocessors that lie inside one compute node. Besides a standard MPI-OpenMP programming approach, which belongs to the symmetric usage mode, two offload-mode programming approaches are considered. The first offload approach is conventional and uses compiler pragmas, whereas the second one is new and combines Intel's APIs of *coprocessor offload infrastructure* (COI) and *symmetric communication interface* (SCIF) for low-latency communication. While the pragma-based approach allows simpler programming, the COI-SCIF approach has three advantages in (1) lower overhead associated with launching offloaded code, (2) higher data transfer bandwidths, and (3) more advanced asynchrony between computation and data movement. The low-level COI-SCIF approach is also shown to have benefits over the MPI-OpenMP counterpart. All the programming approaches are tested by a real-world 3D application, for which the COI-SCIF approach shows a performance upper hand on a Tianhe-2 compute node with three Xeon Phi coprocessors.

1 Introduction

For the field of high-performance computing, energy efficiency considerations have prompted modern supercomputers to adopt accelerators, such as general-purpose GPUs and many-integrated-core (MIC) coprocessors. A good example is Tianhe-2, which is currently ranked No. 1 on the TOP500 List [1]. Three Intel Xeon Phi coprocessors can be found in each of Tianhe-2's 16,000 compute nodes [2]. However, with this unconventional multi-coprocessor-per-node setup come challenges of programming. Apart from ensuring the performance of each

X.-h. Sun et al. (Eds.): ICA3PP 2014, Part II, LNCS 8631, pp. 68–81, 2014.

coprocessor, there arises a new challenge of joining the force of several coprocessors within one compute node. The most important issue in the latter subject concerns implementing data transfers between the coprocessors, to achieve high performance with acceptable coding difficulty.

The Xeon Phi coprocessors from Intel adopt the MIC architecture and support a modified x86 instruction set, thereby providing the programmability of a full-fledged multicore CPU [3–5]. A coprocessor-enhanced compute node has always a CPU *host* consisting of one or more multicore CPU sockets that share a memory address space. There can be one or more coprocessor cards, each connected to the host as a *device* via a PCIe bus. The cores on each coprocessor have access to a shared device memory space that is disjoint from both the host and the other coprocessors.

For a multi-coprocessor compute node, two usage modes can be adopted: *offload* and *symmetric* [6]. In the offload mode, the code is first started on the CPU host, whereas compute-intensive blocks of the code are offloaded to the coprocessors. In the symmetric mode, the coprocessors are considered as independent nodes of a mini-supercomputer. For example, MPI can be used to start the code simultaneously on the coprocessors, and possibly also the CPU host. This MPI approach in the symmetric mode is simple and has the best code portability. However, one major disadvantage with a pure MPI approach is the excessive overhead in memory footprint due to the large number of MPI processes. A remedy is to use one MPI process per coprocessor while adopting OpenMP threads for intra-coprocessor parallelism.

Due to the possible shortcoming of the MPI-based symmetric usage mode, we also want to consider the offload usage mode. The usual approach is to insert an `offload` pragma in front of each code block that is to be offloaded. The resulting coprocessor-coprocessor data transfers are actually relayed through the host. In this paper, we present a new offload programming approach, which allows each coprocessor to run an independent sub-program, while bi-directional and asynchronous coprocessor-coprocessor data transfers are directly enabled by Intel's low-level APIs of *coprocessor offload infrastructure* (COI) [20] and *symmetric communication interface* (SCIF) [21]. The choice of this offload programming approach is motived by performance. We believe this paper is a first effort in studying how to efficiently program multiple Xeon Phi coprocessors within one compute node, by comparing the two offload programming approaches against the MPI-OpenMP counterpart.

The remainder of the paper is organized as follows. Some background information is presented in Section 2, and the related work is surveyed in Section 3. Section 4 explains the two offload programming approaches, using a simple example of 3D stencil computation. Section 5 quantifies the performance advantages of the low-level COI-SCIF approach, in terms of both bandwidth measurements and time usages of a real-world 3D application. All the experiments have been done on a compute node of Tianhe-2, with three Xeon Phi coprocessors.

2 Background

2.1 Xeon Phi Coprocessor

Intel's Xeon Phi coprocessor has up to 61 x86-based Intel CPU cores on a single chip. Each core supports 512-bit SIMD vector computing, and has 32 KB private L1 data cache and 512 KB shared L2 cache. Four hardware threads can be enabled on each core to give up to 244 threads per chip. Each coprocessor has its own device memory and is connected to the CPU host via PCIe bus.

2.2 Pragma-Based Offloading

In this pragma-based programming approach [18], the CPU host controls the entire execution of a code. Blocks of the code can be delegated to the coprocessors for execution. Since memory is not shared between the host and any of the coprocessors, variables and arrays needed in the offloaded code block also have to be allocated on the target coprocessors. The content of the coprocessor data can be transferred back to the host if desired. Below is an example of the directive that combines code offload with host-coprocessor data transfers.

```
#pragma offload target(mic:id) \
        in(input_msg: length(N)) out(output_msg: length(N))
```

Here, `id` is an integer specifying the target coprocessor. The content of array `input_msg` (of length `N`), which is marked by the `in` specifier, is copied from the host at the start of offload. Similarly, the content of array `output_msg` is copied back to the host at the end of offload. A third possible data specifier is `inout`, which marks a variable or array as both input and output. A fourth possible data specifier is `nocopy`, which only marks variables that will be used on the target coprocessor, but without any host-coprocessor data movements (by assuming that these variables persist on the coprocessor). For a code block that is offloaded iteratively, to save the cost of repeatedly allocating/deallocating the same data storage, the modifiers `alloc_if(arg)` and `free_if(arg)` can be used.

To initiate asynchronous host-coprocessor data transfers, such that computations have the possibility of being simultaneously carried out, the `signal` clause can used together with the `offload` pragma or another pragma named `offload_transfer`. The compiler directive only initiates an asynchronous data transfer without offloading any computation to the target coprocessor. A matching `offload_wait` pragma should be used to complete the asynchronous data transfer. An example is as follows:

```
#pragma offload_transfer target(mic:id) \
        out(output_msg: length(N)) signal(output_msg)
                              ...
#pragma offload_wait target(mic:id) wait(output_msg)
```

Although asynchronous data transfers are achievable with pragma-based programming, one major disadvantage is that data transfers between two coprocessors always have to be relayed through the host. The second disadvantage is the offload start-up cost, especially for a code block that is offloaded iteratively.

2.3 COI and SCIF

To realize direct coprocessor-coprocessor data transfers in connection with offload programming, while also avoiding the overhead related to repeated offload start-ups, we use two low-level APIs: COI and SCIF, provided by Intel's MPSS software stack [19]. They provide the programmer with a finer control of code offloading and data transfers.

Two of COI's key abstractions, namely *COIEngine* and *COIProcess*, are important for the following implementations. The first abstraction represents a COI-capable device, e.g., the host or a coprocessor, whereas the second one encapsulates a process created by COI on a remote engine. These two abstractions can be used together to offload computations to multiple coprocessors within one compute node.

SCIF is a low-level API that provides a low-latency communication channel between *clients*, which can be either the host or coprocessors. Efficiency of SCIF is due to direct use of the PCIe bus for bi-directional data transfers between two coprocessors (or between the host and a coprocessor). The following is a list of abstractions used by SCIF:

- *Node*: It is a physical node in SCIF network. Both the host and an MIC card can be seen as a node.
- *Port*: An SCIF port on a node is represented as a 16-bit integer, which is a logical endpoint on the SCIF node similar to an IP port.
- *Endpoint*: The port for a connection is defined as an endpoint, which is similar to a socket.
- *Registered memory*: This is a registered memory driven by SCIF, and is held for the connected endpoints.

For small-amount data transfers (<4KB) between two SCIF clients, the `scif_send` and `scif_recv` functions should be employed, which can also be used for synchronizing the two clients. SCIF also provides remote direct memory access (RDMA) semantics. More specifically, the `scif_register` function exposes local memory on a device for remote access by another device. Then, either function `scif_readfrom` or function `scif_writeto` can be used to initiate asynchronous and zero-copy data transfers (≥4KB) between two devices. Finally, the `scif_fence_signal` function can ensure the completion of an asynchronous RDMA-based data transfer.

2.4 Coprocessor-Only Usage Mode

Strictly speaking, the symmetric usage mode means that the CPU host is used simultaneously with the coprocessors [6], i.e., a form of hybrid computing. We will however loosen the definition of symmetric usage to also include the scenario of only using the coprocessors. This is because if the CPU host is not involved, an existing MPI code can be readily run on multiple coprocessors without the worry of sophisticated load balancing. As mentioned in Section 1,

OpenMP threads can be used to exploit the intra-coprocessor parallelism, giving rise to an MPI-OpenMP programming approach. This is for avoiding the pure MPI approach's excessive overhead in memory footprint, due to the large number of MPI processes.

3 Related Work

Many researchers have focused on single-MIC programming. There are, however, not many publications on programming multiple MIC coprocessors or MIC clusters. As introduced in Section 2, pragma-based offload mode (combined with OpenMP) and MPI-based native/symmetric mode are two existing programming approaches. For the default MPI version included in MPSS, there have been reported bandwidth bottlenecks in intra-node and inter-node MPI communication between a MIC and the host or between two MICs, see [15, 16].

Due to the Intel MPI bandwidth problem in MIC clusters, some researchers proposed alternative MPI implementations for improving the communication performance for the native/symmetric mode. DCFA-MPI [8] is an MPI library implementation for direct inter-node InfiniBand communication between MIC coprocessors. MPICH2-1.5 [9] is an MPI implementation that uses shared memory, TCP/IP, and SCIF-based communication for MIC clusters. The research group of D. K. Panda at The Ohio State University has investigated the communication within a node that consists of a CPU host and one MIC coprocessor [17]. They proposed MVAPICH-PRISM [16], an MPI implementation that is a proxy-based communication framework using InfiniBand and SCIF for MIC clusters. All the above MPI implementations targeted MIC clusters with only one MIC coprocessor per node.

In addition, to solve the MPI bandwidth problem in its early version, Intel MPI has also implemented a proxy-based design that allows hybrid utilization of InfiniBand and SCIF, depending on the actual communication scenario [10].

Some researchers have studied the use of COI and SCIF APIs. COSMIC [11] is a user-level middleware for automatically managing MIC coprocessor resources by scheduling COI processes and their offloads, which can improve both performance and reliability of multiprocessing on MIC coprocessors. Dokulila et al. [12] created a library that supports hybrid execution in C++ applications using MIC coprocessors, where SCIF is used for synchronization and data transfers.

High performance has been achieved on coprocessors for many kernels and some applications. Schulz et al. [13] ported existing scientific applications and micro-kernels to a single MIC coprocessor. Pennycook et al. [14] explored SIMD for molecular dynamics applications on a MIC coprocessor. Rosales [15] has summarized the critical skills for pursuing high performance on Xeon Phi. By offloading the Linpack benchmark to MIC coprocessors, Heinecke et al. [7] achieved over 76% efficiency on a 100-node cluster with two MIC coprocessors per node.

Although COI and SCIF are two established APIs, we believe that our work represents a first effort in combining COI and SCIF for programming multiple MIC coprocessors within one compute node.

4 Two Implementations of a Simple 3D Stencil

This section serves to demonstrate the two offload-based programming approaches and their related data transfers. This will be done through parallelizing a very simple example of 3D stencil computation, to make use of multiple coprocessors within one compute node. The MPI-based programming approach is the same as for the scenario of a CPU cluster, thus not discussed here.

The stencil example involves a box-shaped computational grid that has in total $(n_x+2)\times(n_y+2)\times(n_z+2)$ mesh points. The entire computation is assumed as an iterative loop (over time). During each iteration a 3D array named `C1` is computed by applying a 7-point stencil operator over another 3D array named `C0`. Values of `C1` are prescribed on the entire boundary, so the actual computation per iteration computes the $n_x \times n_y \times n_z$ inner points of `C1` as follows:

```
for (k=1; k<=nz; k++)
   for (j=1; j<=ny; j++)
     for (i=1; i<=nx; i++)
         C1[k][j][i]=a*C0[k][j][i]
            +b*(C0[k][j][i-1]+C0[k][j][i+1]
               +C0[k][j-1][i]+C0[k][j+1][i]
               +C0[k-1][j][i]+C0[k+1][j][i]);
```

Parallelism between the coprocessors can be enforced by dividing the 3D computational grid (and `C0`/`C1` arrays) into subdomains, each being assigned to one coprocessor. Between two neighboring subdomains, values on each other's respective internal boundary layer have to be exchanged through data transfers. It is also customary that the subdomain grid is extended with a layer of ghost points towards each neighbor. An example of 1D grid decomposition can be found in Figure 1.

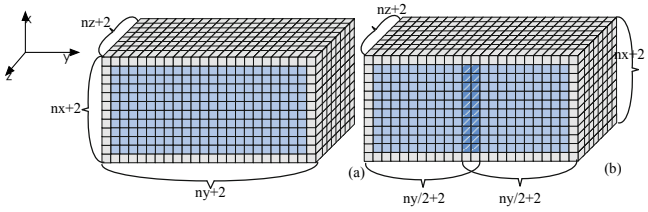

Fig. 1. An example of 1D decomposition (in y-direction) of a 3D grid into two subdomains. (a) Original 3D grid, (b) two subdomains after the decomposition.

The work on each subdomain consists of at least the following tasks per iteration. For each of its neighbors, first pack an "outgoing" buffer (1D array) by copying from respective (possibly non-contiguous) entries of the subdomain 3D array `C0` and then unpack an "incoming" buffer (1D array) by copying its

content to respective (possibly non-contiguous) entries of C0; compute all the entries of the subdomain 3D array C1 (except its boundary entries), by applying a 7-point stencil over the entries of C0; swap the subdomain array pointers C0 and C1 before proceeding to the next iteration. The actual coprocessor-coprocessor data transfers may be mediated by the host, or asynchronously initiated by the coprocessors themselves, depending on the chosen approach of programming.

For simplicity, let us only consider the case of two coprocessors. In the beginning of both implementations, four 3D arrays C00, C10, C01, C11 are allocated on the host side, such that the first two are duplicated on coprocessor 0, and the latter two duplicated on coprocessor 1. It should be obvious from the names that C00 and C01 together constitute the global 3D array C0, which no longer needs a physical storage. The same idea applies to C10, C11 and C1. It is only after all the iterations are done that values of C00, C10, C01, C11 are copied from the coprocessors back to the host.

4.1 Implementation Based on Pragmas

In this implementation, the host also needs to allocate two 1D arrays, in_buffer0 and out_buffer0, on coprocessor 0. Similiarly, in_buffer1 and out_buffer1 are on allocated coprocessor 1. The following code segment shows the actions that happen during each iteration:

```
#pragma omp parallel num_threads(2) {
  int id = omp_get_thread_num();
    if (id==0) {
     #pragma offload target(mic:0) nocopy(C00,C01) \
        in(in_buffer0) out(out_buffer0)
      { // work offloaded to coprocessor0
        ...
      }
    }
    else if (id==1) {
      #pragma offload target(mic:1) nocopy(C10,C11) \
         in(in_buffer1) out(out_buffer1)
       { // work offloaded to coprocessor1
         ...
       }
    }
} // end of OpenMP parallel region
swap_pointers(out_buffer0,in_buffer1);
swap_pointers(out_buffer1,in_buffer0);
```

It should be noted that we have omitted some programming details in the offload pragmas, and details of the offloaded work tasks are also skipped. Coding for coprocessor 0 is identical with that for coprocessor 1, except for the slightly different variable names and the different locations of the respective ghost boundary points.

It can be seen from the above code segment that two OpenMP threads on the host *simultaneously* offload work to the two coprocessors. All data transfers are relayed through the host. In particular, the two swappings of the buffer array pointers ensure the needed coprocessor-coprocessor data exchanges. Another important remark is that although overlapping computation with data movement is theoretically possible, we have chosen a non-overlapping approach above. It otherwise will require each coprocessor to split the offload into several parts. These will be initiated by `offload` or `offload_transfer` pragmas together with the `signal` clause, for the purpose of asynchrony. Some extensive modifications are also needed for the offloaded code blocks.

4.2 Implementation Based on COI and SCIF

The COI-SCIF implementation uses an independent sub-program per coprocessor. At the same time, the host main program is quite different from the previous implementation, i.e., a pair of `COIEngine` and `COIProcess` will be created and connected to each coprocessor. Thereafter, the host can choose not to disturb the two coprocessors, which will carry out the needed computation iterations, interleaved with bi-directional and asynchronous data transfers directly between themselves. That is, data transfers do not pass through the host. As shown in Figure 2, each coprocessor can independently initiate `scif_writeto` towards the other. By paying some extra effort in coding the coprocessor sub-programs, we can obtain several advantages. First, the repeated cost of offload start-ups of the pragma-based implementation is avoided. Instead, using COI and SCIF APIs can make the single-time device code loading and launching more efficient. Second, bi-directional and asynchronous coprocessor-coprocessor data transfers result in higher bandwidths than the host-mediated data transfer approach. Third, the more advanced asynchrony, due to RDMA data accesses such as `scif_readfrom` and `scif_writeto`, make it easier to overlap computation with communication. This possibility of overlapping is illustrated in Figure 3.

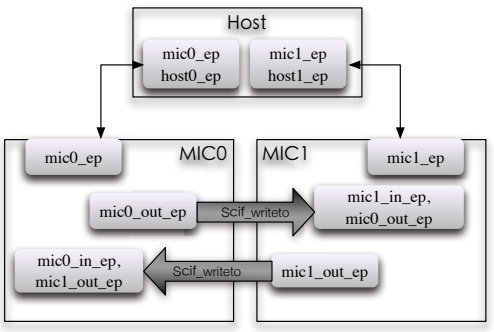

Fig. 2. The coupling between two coprocessors, with a COI-SCIF implementation

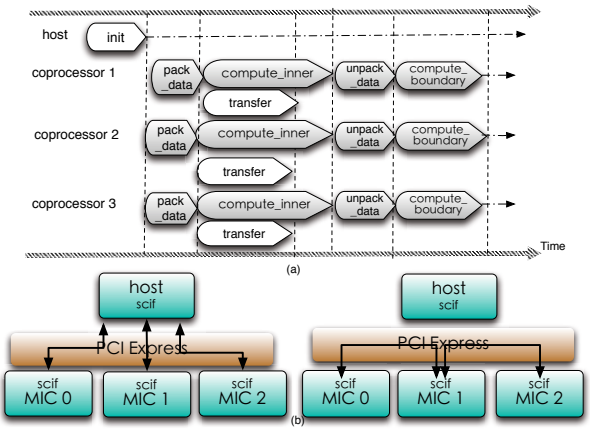

Fig. 3. (a) Overlapping computation and coprocessor-coprocessor data transfers. (b) Data transfers between multiple coprocessors with (left) or without host (right) relay.

5 Experiments and Results

We will report in this section measurements of a set of experiments involving data transfers between multiple Xeon Phi coprocessors. The purpose is to demonstrate the advantages of the COI-SCIF approach, which provides both higher bandwidths and lower overhead related to offload start-ups. Moreover, we want to quantify the resulting performance benefits in connection with solving a real-world 3D reaction-diffusion problem [22] that consists of 7-point stencil computations and additional numerical operations.

5.1 Hardware Platform

One compute node of Tianhe-2 was used as the test hardware platform, having three Intel Xeon Phi 31S1P coprocessors and two Intel Ivy Bridge 12-core E5-2692 CPUs. It should be mentioned that each 31S1P coprocessor has 57 cores, where 56 of them can be used in the offload mode. The PCIe 2.0 bus with 16 lanes between the CPU host and the coprocessors can theoretically offer a bi-directional bandwidth of 16 GB/s in total.

5.2 Bandwidth Tests

Figure 4(a) compares the bandwidth between the following six scenarios of uni-directional data transfer:

- offload-in: data transfer from host to coprocessor by `offload_transfer`;
- offload-out: data transfer from coprocessor to host by `offload_transfer`;
- MIC-Host-r: host-initiated data transfer from coprocessor to host, using the `scif_readfrom` function;

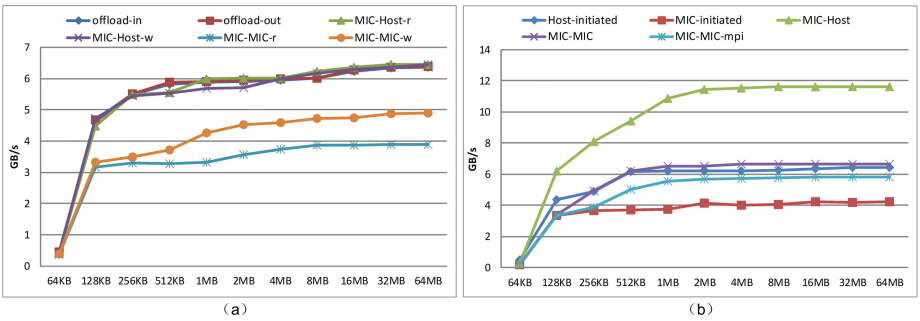

Fig. 4. Measured bandwidths, as functions of the transferred data size, (a) for six scenarios of uni-directional data transfers, (b) for five scenarios of bi-directional data transfers. Details can be found in Section 5.2.

- MIC-Host-w: host-initiated data transfer from host to coprocessor, using the `scif_writeto` function;
- MIC-MIC-r: data transfer from one coprocessor to another (without host involvement), using the `scif_readfrom` function;
- MIC-MIC-w: data transfer from one coprocessor to another (without host involvement), using the `scif_writeto` function.

It can be seen from Figure 4(a) that the first four scenarios enjoy roughly the same bandwidth, which is higher than that of the latter two. Nevertheless, if data need to be transferred from one coprocessor to another, it is still beneficial to use the MIC-MIC-w approach, because otherwise data have to first travel from one coprocessor to the host, then from the host to the other coprocessor.

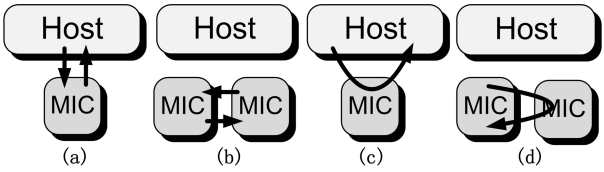

Fig. 5. Four scenarios of bi-directional data transfers: (a) both independently initiate data transfer between MIC and Host, (b) both independently initiate data transfer between MIC and MIC, (c) only host initiates data transfer between MIC and Host, (d) only one MIC initiates data transfer between MIC and MIC

Figure 4(b) shows the bandwidth differences between the following five scenarios of bi-directional data transfer:

- MIC-Host: data transfer between host and coprocessor, for which host and coprocessor independently initiate `scif_writeto`, as illustrated in Figure 5(a);

Table 1. Time usage (in seconds), by a single coprocessor, of three implementations of a real-world 3D application. The total number of time steps is 1000.

Mesh size	Programming mode	Total
$112 \times 1200 \times 142$	Pragma-based	30.12
	COI-SCIF	26.66
	MPI-OpenMP	26.52

- MIC-MIC: data transfer between two coprocessors, for which each coprocessor independently initiates `scif_writeto`, as illustrated in Figure 5(b);
- Host-initiated: data transfer between host and coprocessor, for which both `scif_readfrom` and `scif_writeto` are initiated on the host side, as illustrated in Figure 5(c);
- MIC-initated: data transfer between two coprocessors, for which both the `scif_readfrom` and `scif_writeto` are initiated on the same coprocessor, as illustrated in Figure 5(d);
- MIC-MIC-mpi: data transfer between two coprocessors, for which utilizing `MPI_Isend` and `MPI_Irecv`.

In the case of two coprocessors, it is always better to let both coprocessors simultaneously initiate `scif_readfrom`, instead of letting one coprocessor initiate both `scif_readfrom` and `scif_writeto`.

5.3 Performance of a Real-World 3D Application

We used a real-world 3D application [22] to test the two implementations of offloading, as described in Sections 4.1 and 4.2. Both implementations used OpenMP threads for intra-coprocessor parallelism. The performance of an MPI-OpenMP implementation is also included for comparison. More specifically, the real-world application involved five reaction-diffusion equations. Each equation was numerically split into a reaction part and a diffusion part, where the latter was solved by applying the 7-point stencil operator. In total, each time iteration for solving all the five equations needed 150 floating-point operations per mesh point. All calculations were done using double precision.

Table 1 shows the time usages associated with offloading the computational work to a single Xeon Phi coprocessor. The performance difference is due to the fact that the pragma-based offloading approach induced repeated start-up costs, once every time iteration. Note that no data transfers were needed for this single-coprocessor scenario, therefore no performance difference between the COI-SCIF programming approach and the MPI-OpenMP counterpart.

Table 2 summarizes the time usages associated with employing two or three Xeon Phi coprocessors. Unlike Table 1, the costs of data transfers and packing/unpacking data buffers are now present. The pragma-based offload implementation was considerably slower than the COI-SCIF implementation. There are two reasons for this performance difference. The first reason is due to the repeated offload start-up costs, as we have already experienced for Table 1. The

Table 2. Time usage (in seconds) of four implementations of a real-world 3D application. The version of "COI-SCIF*" refers to relaying data transfers via the host. Number of time steps: 1000, global mesh size: $112 \times 1200 \times 142$.

		Pragma-based	COI-SCIF*	MPI-OpenMP	COI-SCIF
2 Coprocessors	Pack/unpack	0.41	0.41	0.40	0.40
	Data trans	1.27	1.26	0.98	0.80
	Total	19.34	15.08	14.91	14.62
3 Coprocessors	Pack/unpack	0.40	0.40	0.40	0.40
	Data trans	1.21	1.31	0.99	0.76
	Total	12.63	10.22	9.72	9.43

second reason is due to the less efficient data transfers of the pragma-based implementation, demonstrated by the "Data trans" row in Table 2.

We recall that the COI-SCIF implementation adopts bi-directional and asynchronous coprocessor-coprocessor data transfers, thereby capable of hiding (a part of) the data transfer costs. The MPI-based symmetric implemetation also has the advantages in asynchronous data transfers between coprocessors, but the extra overhead of MPI communication leds to a lower performance than the low-level COI-SCIF implementation. For comparison purposes, Table 2 also includes another implementation based on using the COI and SCIF APIs. This third implementation, denoted as COI-SCIF*, relayed data transfers through the host. It thereby closely resembled the pragma-based implementation with respect to data transfers, and also that no overlap happened between data transfer and computation.

6 Conclusions

This paper has focused on two offload programming approaches that can be used for a single compute node with multiple coprocessors. An MPI-based symmetric programming approach is included for comparison purposes. The three approaches, MPI-based, pragma-based and COI-SCIF-based, have rather different characteristics. While the first two are easier to use, the latter one gives better performance but requires more involved programming. For a real-world 3D application, the best performance was achieved by the COI-SCIF approach, where bi-directional and asynchronous data transfers were enabled directly between the coprocessors. The low-level COI-SCIF approach also resulted in lower communication overhead, in comparison with the MPI-based approach. It should be remarked that this programming approach is not limited to stencil computation on regular meshes. Our findings not only shed some light on this new topic of using multiple Xeon Phi coprocessors within one compute node, but provide a good starting point for fully utilizing Tianhe-2 in future.

References

1. Top500, China's Tianhe-2 Supercomputer Takes No.1 Ranking on 41st TOP500 List, http://www.top500.org/blog/lists/2013/06/press-release/
2. Dongarra, J.: Visit to the National University for Defense Technology Changsha, http://www.netlib.org/utk/people/JackDongarra/PAPERS/tianhe-2-dongarra-report.pdf
3. Intel Corporation, Intel Xeon Phi Coprocessor Instruction Set Architecture Reference Manual. Reference number 327364-001 (2012)
4. Jeffers, J., Reinders, J.C.: Intel Xeon Phi Coprocessor High-Performance Programming. Morgan Kaufmann, Walthman (2013)
5. Intel MIC Architecture, http://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner
6. Intel Corporation, Intel Xeon Phi System Software Developer's Guide. Reference number 328207-001EN (2012)
7. Heinecke, A., Vaidyanathan, K., Smelyanskiy, M., Kobotov, A., Dubtsov, R., Henry, G., Chrysos, G., Dubey, P.: Design and implementation of the Linpack benchmark for single and multi-node systems based on Intel Xeon Phi coprocessor. In: IPDPS (2013), doi:10.1109/IPDPS.2013.113
8. Si, M., Ishikawa, Y., Direct, M.P.I.: library for Intel Xeon Phi Co-Processors. In: 27th IEEE International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), Boston, MA, USA (2013), doi:10.1109/IPDPSW.2013.179
9. MPICH: High-performance and Portable MPI, http://www.mpich.org/
10. OFS for Xeon Phi, https://www.openfabrics.org/images/docs/2013Dev_WorkshopnewlineMon_0422/2013_Workshop_Mon_1430_OpenFabrics_OFS_software_for_Xeon_Phi.pdf
11. Cadambi, S., Coviello, G., Li, C., Phull, R., Rao, K., Sankaradass, M., Chakradhar, S.: COSMIC: Middleware for high performance and reliable multiprocessing on Xeon Phi coprocessors. In: Proceedings of the 22nd Int'l Symposium on High-Performance Parallel and Distributed Computing, HPDC 2013 (2013), doi:10.1145/2462902.2462921
12. Dokulila, J., Bajrovica, E., Benknera, S., Pllanaa, S., Sandriesera, M., Bachmayerb, B.: High-level support for hybrid parallel execution of C++ applications targeting Intel Xeon Phi coprocessors. In: 2013 International Conference on Computational Science, ICCS 2013 (2013), doi:10.1016/j.procs.2013.05.430
13. Schulz, W., Ulerich, K., Malaya, R., Bauman, N., Stogner, T.P., Simmons, R., Early, C.: experiences porting scientific applications to the many integrated core (MIC) platform. In: TACC-Intel Highly Parallel Computing Symposium, Tech. Rep. (2012), doi:10.1145/2016741.2016764
14. Pennycook, J., Hughes, S., Smelyanskiy, J.C., Jarvis, M., Exploring, A.S.: SIMD for molecular dynamics, using Intel Xeon processors and Intel Xeon Phi coprocessors. In: IEEE Int'l Parallel & Distributed Processing Symposium (2013), doi:10.1109/IPDPS.2013.44
15. Rosales, C.: Porting to the Intel Xeon Phi: Opportunities and challenges. In: Extreme Scaling Workshop, XSCALE 2013 (2013)
16. Potluri, S., Bureddy, D., Hamidouche, K., Venkatesh, A., Kandalla, K., Subramoni, H., Panda, D.K.: MVAPICH-PRISM: A Proxy-based Communication Framework using InfiniBand and SCIF for Intel MIC Clusters. In: Int'l Conference on Supercomputing (2013)

17. Potluri, S., Venkatesh, A., Bureddy, D., Kandalla, K., Panda, K.: D., Efficient intra-node communication on Intel-MIC clusters. In: 13th IEEE Int'l Symposium on Cluster Computing and the Grid, CCGrid 2013 (2013), doi:10.1109/CCGrid.2013.86
18. The Heterogeneous Offload Model for Intel Many Integrated Core Architecture, http://software.intel.com/sites/default/files/article/326701/heterogeneous-programming-model.pdf
19. Intel Manycore Platform Software Stack (MPSS), http://software.intel.com/en-us/articles/intel-manycore-platform-software-stack-mpss#downloads
20. Intel Corporation, MIC COI API Reference Manual 0.65. Monday December 17 12:12:33 (2012)
21. Intel Corporation, MIC SCIF API Reference Manual 0.65 for User Mode Linux. Mon Dec17 12:05:03 (2012)
22. Chai, Jun, Hake, Johan, Wu, Nan, Wen, Mei, Cai, Xing, Lines, T., Glenn, Yang, Jing, Su, Huayou, Zhang, Chunyuan, Liao, Xiangke, S.: Towards simulation of subcellular calcium dynamics at nanometre resolution. International Journal of High Performance Computing Applications (2013)