CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. (2015)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.3507

SPECIAL ISSUE PAPER

Communication-hiding programming for clusters with multi-coprocessor nodes

Xinnan Dong[1], Mei Wen[1], Jun Chai[1], Xing Cai[2,3,*,†], Mandan Zhao[4] and Chunyuan Zhang[1]

[1]*School of Computer Science, National University of Defense Technology, Changsha, Hunan 410073, China*
[2]*Simula Research Laboratory, P.O. Box 134, 1325 Lyakser, Norway*
[3]*Department of Informatics, University of Oslo, P.O. Box 1080 Blindern, 0316 Oslo, Norway*
[4]*College of Navigation and Aerospace Engineering, Information Engineering University, Zhengzhou, Henan, China*

SUMMARY

Future exascale systems are expected to adopt compute nodes that incorporate many accelerators. To shed some light on the upcoming software challenge, this paper investigates the particular topic of programming clusters that have multiple Xeon Phi coprocessors in each compute node. A new offload approach is considered for intra-node communication, which combines Intel's APIs of *coprocessor offload infrastructure* (COI) and *symmetric communication interface* (SCIF) for achieving low latency. While the conventional pragma-based offload 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, which belongs to the symmetric usage mode. Moreover, a hybird programming strategy based on COI-SCIF is presented for joining the computational force of all CPUs and coprocessors, while realizing communication hiding. All the programming approaches are tested by a real-world 3D application, for which the COI-SCIF-based approach shows a performance advantage on Tianhe-2. Copyright © 2015 John Wiley & Sons, Ltd.

Received 3 March 2015; Accepted 15 March 2015

KEY WORDS: Intel Xeon Phi coprocessor; offload model; SCIF; hybrid programming; Tianhe-2

1. INTRODUCTION

Due to energy-efficiency considerations, future extreme-scale systems will not only be enhanced by hardware accelerators, such as general purpose graphic processing units and many-integrated-core (MIC) coprocessors, but are also projected to have multiple accelerators per compute node. This is exemplified by 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 coprocessor, there arises a new challenge of joining the force of all heterogeneous computing resources, both within one compute node and across many nodes. Here are two important issues. First, the various inter-node and intra-node communication tasks must be executed at the highest speed simultaneously, while the computation proceeds. Second, the CPUs should also carry out an appropriate portion of the entire computation, to avoid wasting the unused computational capacity.

*Correspondence to: Xing Cai, Simula Research Laboratory, P.O. Box 134, 1325 Lyakser, Norway.
†E-mail: xingca@simula.no

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 on 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 because of 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, in order to effectively program multiple Xeon Phi coprocessors within one compute node, we adopt a new offload programming approach that 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) [7] and *symmetric communication interface* (SCIF) [8]. Furthermore, we present a hybrid programming strategy combining techniques such as MPI, OpenMP, COI and SCIF, thus extending our work to cover clusters with multi-coprocessor nodes. This hybrid strategy is motivated by performance. Besides intra-node communication efficiency, results also show that the impact of inter-node communication can be decreased by a proper overlap with the computation. The total workload is partitioned between the CPUs and coprocessors in a way that ensures no waste of the CPUs' computational capacity while overlapping various communication tasks with the computation.

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 programming approaches, using a simple example of 3D stencil computation. Section 5 quantifies the performance advantages of the intra-node and inter-node programming approaches based on low-level COI-SCIF, in terms of both bandwidth benchmark measurements and time usages of a real-world 3D application. All the experiments have been done on Tianhe-2.

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 [9], the CPU host controls the entire execution of a code. Blocks of the code can be delegated to the coprocessors for execution. Because 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. The following is an example of the directive that combines code offload with host-coprocessor data transfers.

DOI: 10.1002/cpe

```
#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 be 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. Coprocessor offload infrastructure and symmetric communication interface

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 [10]. They provide the programmer with a finer control of code offloading and data transfers.

Two of COI's key abstractions are *COIEngine* and *COIProcess*. The first abstraction represents a COI-capable device, for example, 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 because of 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 a 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

DOI: 10.1002/cpe

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], that is, 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 workload 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 an MIC and the host or between two MICs; see [11, 12].

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 [13] is an MPI library implementation for direct inter-node InfiniBand communication between MIC coprocessors. MPICH2-1.5 [14] 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 [15]. They proposed MVAPICH-PRISM [12], an MPI implementation that is a proxy-based communication framework using InfiniBand and SCIF for MIC clusters. All the aforementioned 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 [16].

Some researchers have studied the use of COI and SCIF APIs. COSMIC [17] 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.* [18] 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.* [19] ported existing scientific applications and micro-kernels to a single MIC coprocessor. Chen *et al.* [20] from NUDT sped up several important arithmetics on the MIC coprocessor and achieved the automatic translation and optimization from OpenACC to Intel offload. Pennycook *et al.* [21] explored SIMD for molecular dynamics applications on an MIC coprocessor. Rosales [11] summarized the critical skills for pursuing high performance on Xeon Phi. By offloading the Linpack benchmark to MIC coprocessors, Heinecke *et al.* [22] achieved over 76% efficiency on a 100-node cluster with two MIC coprocessors per node.

Considering hybrid programming, Meng *et al.* [23] presented a computing framework for the Uintah software on an MIC-enhanced cluster. The CPU and coprocessors run the MPI processes dependently, and Pthreads are used for intra-CPU or intra-coprocessor parallelism. However, the programming framework only targets clusters that have one coprocessor in each compute node, so

DOI: 10.1002/cpe

the problem of load balancing has not been considered in the case of having multiple coprocessors within each compute 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, and the extended hybrid programming strategy provides a good starting point for achieving high communication efficiency on a cluster of multi-coprocessor nodes.

4. EXAMPLE: IMPLEMENTING A SIMPLE 3D STENCIL

This section serves to explain the hybrid programming strategy, as well as how to handle the involved intra-node and inter-node data transfers. We will start with the COI-SCIF programming approach for one multi-coprocessor node. This will be done through parallelizing a very simple example of 3D stencil computation. The standard pragma-based offload programming approach and the MPI-based symmetric programming approach are conventional, thus not discussed here.

4.1. Problem description

The 3D stencil example assumes 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 the `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.

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

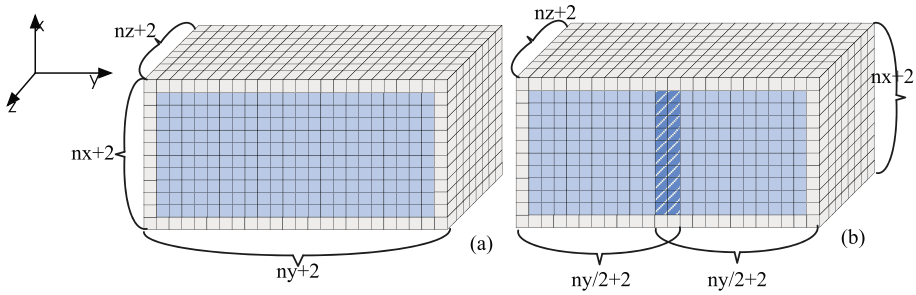

Figure 1. An example of 1D decomposition (in the y-direction) of a 3D uniform grid into two subdomains. (a) The original 3D grid and (b) two subdomains after the decomposition.

DOI: 10.1002/cpe

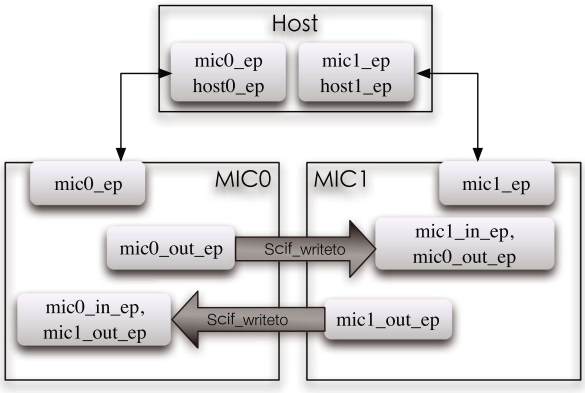

Figure 2. The direct coupling between two coprocessors, achieved by a coprocessor offload infrastructure–symmetric communication interface implementation.

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.

4.2. Coprocessor offload infrastructure–symmetric communication interface implementation for a two-coprocessor node

The conventional pragma-based offload programming approach always relays coprocessor-coprocessor data exchange through the CPU host. Another disadvantage for this stencil example is the unavoidable overhead of repeatedly offloading work from the host to the two coprocessors (once per iteration). In comparison, a COI-SCIF-based implementation uses an independent sub-program per coprocessor. The host main program is also quite different from that of the pragma-based implementation. More specifically, a pair of `COIEngine` and `COIProcess` will be created by the host and connected to each coprocessor. Thereafter, the host can choose not to disturb the two coprocessors, which will carry out *all* the 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, and the pseudo code on the host side is shown as follows.

```
// start MIC0 to run sub-programm 0
COIEngineGetHandle (COI_ISA_KNC,0,&coi_engine0);
COIProcessCreateFromFile (coi_engine0,mic0_main,\
                          &mic0_arg,&coi_proc0);
// start MIC1 to run sub-programm 1
COIEngineGetHandle (COI_ISA_KNC,1,&coi_engine1);
COIProcessCreateFromFile (coi_engine1,mic1_main,\
                          &mic1_arg,&coi_proc1);
// establish SCIF connection of Host-MIC0
host0_ep = scif_open();
scif_bind (host0_ep,host0_portNum);
scif_listen (host0_ep);
scif_accept(host0_ep, host0_portNum, &mic0_ep);
C0_0_reg = scif_register(mic0_ep,C0_0,C0_0_array_size,\
                         C0_0_reg_addr);
// establish SCIF connection of Host-MIC1
...
// wait for MIC0 and MIC1 to finish all iterations
```


DOI: 10.1002/cpe

```
// collect results from MIC0 and MIC1 via SCIF
...
// free registered windows
scif_unregister(mic0_ep,C0_0_reg,C0_0_array_size);
scif_unregister(mic1_ep,C0_1_reg,C0_1_array_size);
// close SCIF endpoints
scif_close (mic0_ep);
scif_close (host0_ep);
scif_close (mic1_ep);
scif_close (host1_ep);
// stop process on MIC0 and MIC1
COIProcessDestroy (coi_proc0);
COIProcessDestroy (coi_proc1);
```

The sub-programs (not shown here) for the two coprocesssors involve some elaborate programming details, but the advantages are three-fold. First, the repeated cost of offload start-ups of the pragma-based implementation is avoided. 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`, makes it easier to overlap computation with communication. This possibility of overlapping is illustrated in Figure 3 (assuming three coprocessors).

4.3. A hybrid programming strategy

4.3.1. Overall design. For MIC-enhanced clusters, where each node has multiple coprocessors, we propose a hybrid programming approach to take advantage of the various resources while also overlapping computation with communication. The proposed programming model is based on MPI+OpenMP+COI+SCIF, as shown in Figure 4. The CPUs serve as a communication proxy. For each coprocessor, the host program creates a COI process to establish a connection with it. As usual, MPI is used to deal with the communication between the compute nodes. For intra-node communication, we program with SCIF, which provides an efficient bi-directional RDMA

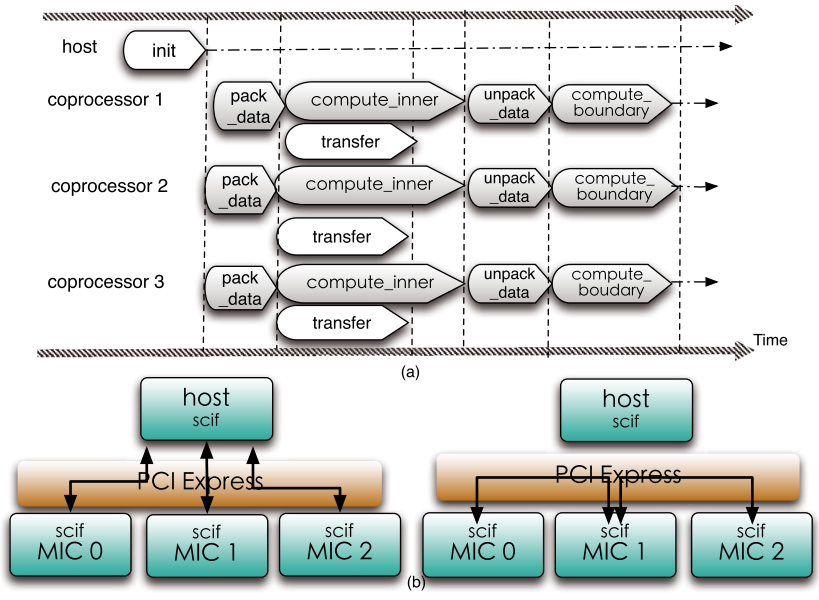

Figure 3. (a) Overlapping computation and coprocessor-coprocessor data transfers and (b) data transfers between multiple coprocessors with (left) or without host (right) relay.

DOI: 10.1002/cpe

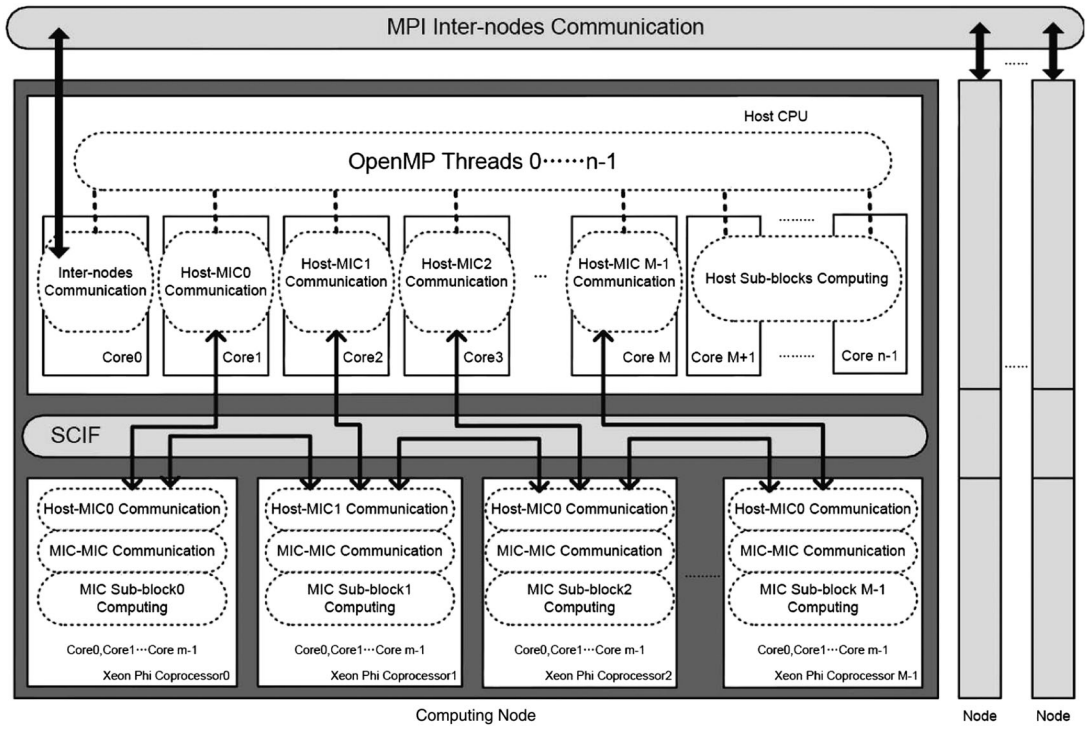

Figure 4. The hybrid programming approach for a many integrated core-enhanced cluster.

communication channel. In addition, the CPUs are also in charge of parts of the computational workload to utilize their unused computational capability. Specifically, we create one MPI process per node, which then spawns as many OpenMP threads as the number of CPU cores. One of the threads is reserved for launching non-blocking MPI routines to handle the inter-node communication. Then, for each coprocessor, one OpenMP thread takes charge and performs (if necessary) host-coprocessor communication through SCIF. The remaining OpenMP threads compute the workload assigned to the CPU host. OpenMP threads are also adopted within each coprocessor. This programming model can ensure the highest speed of inter-node and intra-node data transfers while overlapping these communication tasks with the computation (which is shared between the coprocessors and CPUs).

4.3.2. Division of workload. Because both the CPU host and coprocessors are used in the computation, the workload assigned to different devices must be balanced to achieve high performance. The entire computational mesh is first divided into sub-grids, each assigned to one compute node. Workload division between the CPU host and coprocessors on each node requires some care, not only because of the disparity in computational capability but also because the CPU host has an additional responsibility of inter-node communication and intra-node host-coprocessor data transfer. In this paper, we consider three ways of partitioning the intra-node workload, as shown in Figure 5. Through experiments, we found that the third way shown in Figure 5(c) is an appropriate tradeoff between the first and second alternatives. On one hand, it can reduce the host's workload compared with the second alternative, while still keeping the host busy. On the other hand, the coprocessors can help to compute the content of some of the outgoing MPI messages, so that these can be transferred to the host early enough for securing a good overlap between inter-node communication and computation.

Figure 6 displays an ideal execution flow of the proposed hybrid programming model, which realizes hierarchical pipelining and achieves a complete overlap between computation and communication by non-blocking asynchronous data transfers. The workloads of the coprocessors and host

DOI: 10.1002/cpe

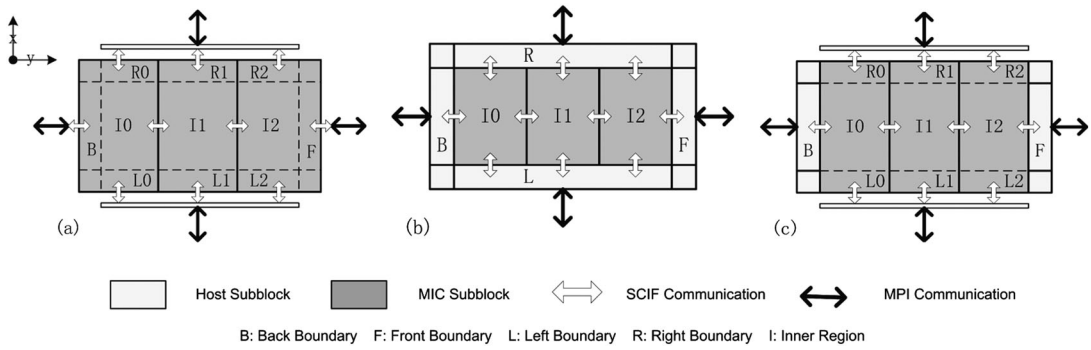

Figure 5. 2D view of three ways of dividing the workload within a single node. (a) The CPU host does no computation. (b) All boundaries are assigned to the host. (c) Some of the boundaries are assigned to the host.

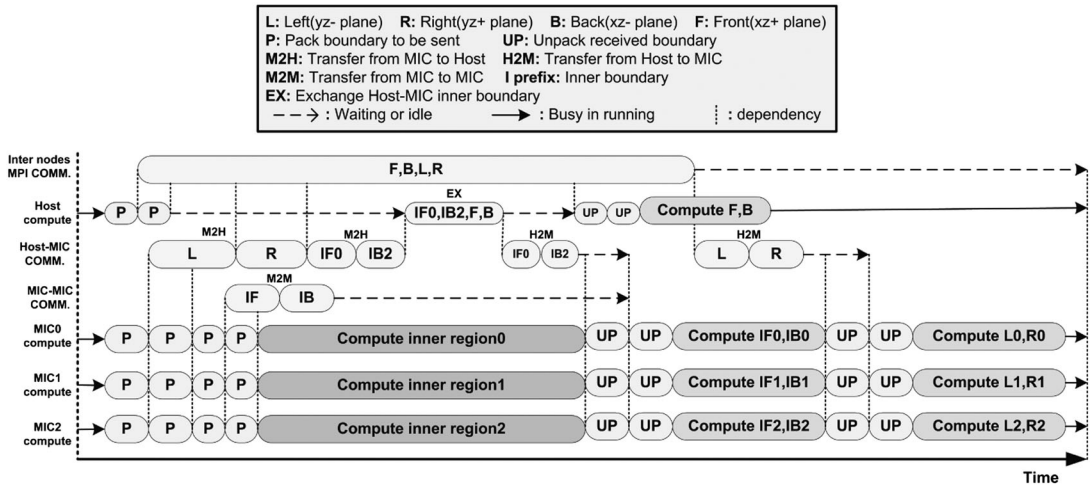

Figure 6. The ideal execution flow for a cluster with three-coprocessor nodes.

are divided into smaller blocks: the boundaries needed for communication and the inner region. In addition, the processing order of the sub-blocks should be scheduled carefully to perfectly overlap inter-node and intra-node communications with the computation of inner regions.

5. EXPERIMENTS AND RESULTS

We will report in this section, measurements of a set of experiments. 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 of the hybrid programming strategy in connection with solving a real-world 3D reaction-diffusion problem [24] that consists of several seven-point stencil computations and additional numerical operations.

5.1. Hardware platform

Each Tianhe-2 compute node is equipped with three Intel Xeon Phi 31S1P coprocessors and two Intel Ivy Bridge 12-core E5-2692 CPUs. Every 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. For the entire cluster, a fat tree interconnect network is used, with custom network interface and switch chips (THNI), and each channel has 160 Gbps bi-directional bandwidth. MPICH2 v1.4.1p1 is used for inter-node communication.

DOI: 10.1002/cpe

5.2. *Bandwidth tests*

Figure 7(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;
- 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 7(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.

Figure 7(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 8(a);
- MIC-MIC: data transfer between two coprocessors, for which each coprocessor independently initiates `scif_writeto`, as illustrated in Figure 8(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 8(c);

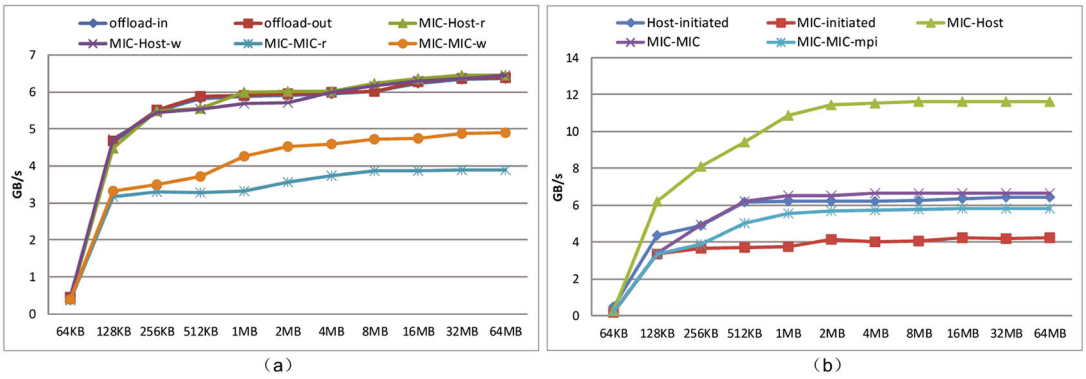

Figure 7. Measured bandwidths, as functions of the transferred data size, (a) for six scenarios of uni-directional data transfers and (b) for five scenarios of bi-directional data transfers. Details can be found in Section 5.2.

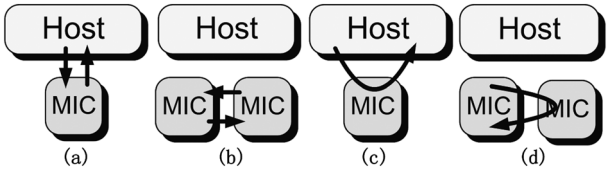

Figure 8. Four scenarios of bi-directional data transfers: (a) both independently initiate data transfer between MIC and host, (b) both independently initiate data transfer between MIC and MIC, (c) only host initiates data transfer between MIC and host, and (d) only one MIC initiates data transfer between MIC and MIC.

DOI: 10.1002/cpe

- 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 8(d);
- MIC-MIC-mpi: data transfer between two coprocessors, for which two MPI processes call `MPI_Isend` and `MPI_Irecv`.

In the case of direct data exchange between two coprocessors, it is always better to let both coprocessors simultaneously initiate `scif_writeto` towards each other, instead of letting one coprocessor initiate both `scif_readfrom` and `scif_writeto`. Relaying the data through the CPU host incurs extra overhead. More specifically, when the SCIF API is used properly, there is no loss in the obtained bandwidth but with increased latency (not shown in Figure 7(b)). When MPI routines are used to communicate the data, however, there is additional loss in the obtained bandwidth, as shown in Figure 7(b). This means that if optimal speed is the goal, MPI should not be chosen for programming data transfers between multiple coprocessors on the same compute node.

5.3. Performance of a real-world 3d application

We used a real-world 3D application [24] to test the two implementations of offloading. 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 seven-point stencil operator. All calculations were done using double precision.

5.3.1. Single-node performance. Table I shows the time usages associated with two ways of 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.

Table II summarizes the time usages associated with employing two or three Xeon Phi coprocessors. Unlike Table I, 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 in Table I. The second reason is due to the

Table I. Time usage (in seconds), by a single coprocessor, of two implementations of a real-world 3D application.

Programming mode	Total time
Pragma-based	30.12
COI-SCIF	26.66

Total number of time steps is 1000. Mesh size: $112 \times 1200 \times 142$.

Table II. Time usage (in seconds) of four implementations of a real-world 3D application.

		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	1.09	0.76
	Total	12.63	10.22	9.82	9.43

The version of 'COI-SCIF*' refers to relaying data transfers via the host. Number of time steps: 1000 and global mesh size: $112 \times 1200 \times 142$.

DOI: 10.1002/cpe

less efficient data transfers of the pragma-based implementation, demonstrated by the 'Data trans' row in Table II.

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 leads to a lower performance than the low-level COI-SCIF implementation. For comparison purposes, Table II also includes another implementation based on using the COI and SCIF APIs. This special 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.

5.3.2. Multi-node performance. Table III shows the execution time of the 3D application on 16 nodes, using the three ways of host-coprocessors workload partitioning depicted in Figure 5. It can be seen that the second partitioning alternative gets the worst performance. In fact, offloading the whole boundary computation to the host increases the execution time significantly. The time taken to compute the Left-Right boundary is far more than that for the Front-Back boundary, because of both a larger size and poorer data locality. However, if the task of computing Left-Right boundary is given to the powerful coprocessors, whereas the host computes the Front-Back boundary, the execution time returns to normal. Although the third workload partitioning does not achieve faster time than the first partitioning in Table III, the lesson learned is that letting the host compute the Front-Back boundary will not slow down the coprocessor. Actually, the thickness of the Front-Back boundary layer can be increased to better utilize the CPU host's computing capacity. This will in turn reduce the workload assigned to the coprocessors, thereby decreasing the total time usage. In addition, by giving the Front-Back boundary calculation to the host, we can start the respective inter-node MPI communication as soon as the host calculation is done, which will overlap with the computation of the Left-Right boundary on the coprocessors and the subsequent coprocessor-host data transfer.

To find out how much workload should ideally be given to the host, we varied the thickness of the Front-Back boundary layer, in connection with the third workload partitioning scheme. Table IV shows the resulting performance measurements. Note that the workload for each coprocessor was fixed at $112 \times 400 \times 142$, which means that the global mesh became larger when we assigned a thicker Front-Back boundary layer to the host. For this setup, we found that the thickness can be up to 22 without increasing the total time usage.

Table V summarizes the time usages of the three programming approaches: pragma-based, MPI+OpenMP, and the new COI-SCIF-based hybrid programming strategy. In the table, it can be seen that the COI-SCIF-based approach clearly outperforms the other two approaches because of a better overlap between communication and computation, as well as faster intra-node data transfers. It should be noticed that we divided the whole data mesh for hiding the communication overhead. This resulted in poor data locality on the host. That is why the host-compute time of the hybrid implementation is slightly longer than the others.

Table III. Time usage (in seconds) associated with 16 nodes and three ways of host-coprocessor workload partitioning.

Boundaries for host	Host compute time	Total
None	N/A	16
Left-Right+Front-Back	14.7+1.06	25
Front-Back	1.0	16

Sub-mesh size assigned to each coprocessor is always $112 \times 400 \times 142$.
The boundary thickness is 1 and the total number of time steps is 1000.

DOI: 10.1002/cpe

Table IV. Time measurements (in seconds) using 16 nodes, as function of the thickness of the Front-Back boundary that is assigned to the host.

F-B thickness	Host compute time	Total time
2	1.06	16.65
16	8.92	16.90
18	10.24	16.46
20	11.50	16.51
22	12.75	16.49
24	13.92	17.61

The sub-mesh size assigned to each coprocessor is always $112 \times 400 \times 142$ and the total number of time steps is 1000.

Table V. Time measurements (in seconds), using 16 nodes, of three implementations of a real-world 3D application.

	Pragma-based	MPI-OpenMP	Hybrid approach
MPI inter-comm	5.85	5.79	5.82
Host compute time	11.69	11.75	12.66
Total	21.02	19.62	16.54

The thickness of the Front-Back boundary (assigned to the host) is 22; the sub-mesh size assigned to each coprocessor is $112 \times 400 \times 142$, and the total number of time steps is 1000.

6. CONCLUSIONS

In the context of a compute node that has multiple coprocessors, this paper has compared three different programming approaches: MPI-based, pragma-based, and COI-SCIF-based. While the first two are easier to implement, the last one gives better performance, but requires more involved programming effort. On the topic of how to efficiently use a cluster of multi-coprocessor nodes, the paper has proposed a hybrid programming strategy. A variety of programming techniques are used, including MPI, OpenMP, COI, SCIF, and appropriate workload partitioning to realize the overall efficiency. For a real-world 3D application, considering the intra-node communication, 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. Measurements also show that the hybrid programming strategy, using COI and SCIF, naturally extends to clusters of multi-coprocessor nodes.

Programming coprocessor-enhanced clusters with computation and communication efficiency is a hard task. Pre-operations and performance analysis are often needed for a real simulation to obtain the optimal settings. It should be remarked that this COI-SCIF-based programming approach is not limited to stencil computation on regular grids. Our findings not only shed some light on this new topic of using multiple Xeon Phi coprocessors within one compute node but also provide a good starting point for fully utilizing Tianhe-2 in the future.

ACKNOWLEDGEMENTS

This work was supported by the National Natural Science Foundation of China, under NSFC (grants Nos. 61033008 and 61272145), and National 863 Program (grants Nos. 2012AA012706). Support from the Innovation Program of NUDT Graduate School (grants Nos. B100603, B120605, and CJ11-06-01) and the FRINATEK program of the Research Council of Norway (project No. 214113) is also acknowledged. The authors gratefully acknowledge the assistance from the National Supercomputing Centre in GuangZhou and Dr. Huayou Su of NUDT.

DOI: 10.1002/cpe

REFERENCES

1. TOP500. *China's Tianhe-2 Supercomputer takes No.1 ranking on 41st TOP500 list*, 2013. (Available from: http://www.top500.org/blog/lists/2013/06/press-release/) [Accessed on June 2013].
2. Dongarra J. *Visit to the National University for Defense Technology Changsha, China*, 2013. (Available from: http://www.netlib.org/utk/people/JackDongarra/PAPERS/tianhe-2-dongarra-report.pdf) [Accessed on May 2013].
3. Intel Corporation. *Intel Xeon Phi Coprocessor Instruction Set Architecture Reference manual*, 2012. (Available from: https://software.intel.com/sites/default/files/forum/278102/327364001en.pdf) [Accessed on June 2014].
4. Jeffers J, Reinders J. *Intel Xeon Phi Coprocessor High-Performance Programming*. Morgan Kaufmann: Amsterdam, 2013.
5. Chrysos G. *Intel Xeon Phi coprocessor – the architecture*, 2012. (Available from: http://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner) [Accessed on June 2014].
6. Intel Corporation. *Intel Xeon Phi Coprocessor System Software Developers Guide*, 2014. (Available from: http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/xeon-phi-coprocessor-system-software-developers-guide.pdf) [Accessed on June 2014].
7. Intel Corporation. *MIC COI API reference manual 0.65*, 2012.
8. Intel Corporation. *MIC SCIF API reference manual 0.65 for user mode Linux*, 2012.
9. Intel Corporation. *The heterogeneous offload model for Intel many integrated core architecture*, 2013. (Available from: http://software.intel.com/sites/default/files/article/326701/heterogeneous-programming-model.pdf) [Accessed on June 2014].
10. Intel Corporation. *Intel Manycore Platform Software Stack (MPSS)*, 2014. (Available from: http://software.intel.com/en-us/articles/intel-manycore-platform-software-stack-mpss#downloads) [Accessed on October 2014].
11. Rosales C. *Porting to the Intel Xeon Phi: Opportunities and Challenges*, Extreme Scaling Workshop (XSCALE13). IEEE: Colorado, U.S.A, 2013.
12. Potluri S, Bureddy D, Hamidouche K, Venkatesh A, Kandalla K, Subramoni H, Panda DK. MVAPICH-PRISM: A proxy-based communication framework using InfiniBand and SCIF for Intel MIC clusters. *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, ACM, Denver, CO, 2013; 1–11.
13. Si M, Ishikawa Y. Direct MPI library for Intel Xeon Phi co-processors. *IEEE 27th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*, IEEE, Cambridge, MA, 2013; 816–824.
14. *MPICH: High-performance and portable MPI*, 1992. (Available from: http://www.mpich.org/) [Accessed on June 2014].
15. Potluri S, Tomko K, Bureddy D, Panda DK. Intra-MIC MPI communication using MVAPICH2: early experience. *Proceedings of TACC-Intel Highly-Parallel Computing Symposium*, Austin, TX, 2012; 1–6.
16. *OFS for Xeon Phi*, 2013. (Available from: https://www.openfabrics.org/images/docs/2013_Dev_Workshop/Mon_0422/2013_Workshop_Mon_1430_OpenFabrics_OFS_software_for_Xeon_Phi.pdf) [Accessed on June 2014].
17. 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. *Proceedings of the 22nd Int'l Symposium on High-Performance Parallel and Distributed Computing (HPDC '13)*, New York, NY, 2013; 215–226.
18. 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. *2013 International Conference on Computational Science (ICCS 2013)*, Barcelona, Spain, 2013; 2508–2511.
19. Schulz KW, Ulerich R, Malaya N, Bauman PT, Stogner R, Simmons C. Early experiences porting scientific applications to the many integrated core (MIC) platform. *TACC-Intel Highly Parallel Computing Symposium, Tech. Rep.*, Austin, TX, 2012; 1–6.
20. Chen C, Yang C, Tang T, Wu Q, Zhang P. *OpenACC to Intel Offload: Automatic Translation and Optimization*, Computer Engineering and Technology. Springer: Heidelberg, 2013; 111–120.
21. Pennycook SJ, Hughes CJ, Smelyanskiy M, Jarvis SA. Exploring SIMD for molecular dynamics, using Intel Xeon processors and Intel Xeon Phi coprocessors. *IEEE Int'L Parallel & Distributed Processing Symposium*, Boston, MA, 2013; 1085–1097.
22. Heinecke A, Vaidyanathan K, Smelyanskiy M, Kobotov A, Dubtsov R, Henry G, Shet AG, Chrysos G, Dubey P. Design and implementation of the Linpack benchmark for single and multi-node systems based on Intel Xeon Phi coprocessor. *IEEE 27th International Symposium on Parallel & Distributed Processing (IPDPS 2013)*, IEEE, Boston, MA, 2013; 126–137.
23. Meng Q, Humphrey A, Schmidt J, Berzins M. Preliminary experiences with the Uintah framework on Intel Xeon Phi and stampede. *Proceedings of the Conference on Extreme Science and Engineering Discovery Environment: Gateway to Discovery*, XSEDE '13, ACM, New York, NY, USA, 2013; 48:1–48:8. (Available from: http://doi.acm.org/10.1145/2484762.2484779) [Accessed on June 2014].
24. Chai J, Hake J, Wu N, Wen M, Cai X, Lines GT, Yang J, Su H, Zhang C, Liao X. Towards simulation of subcellular calcium dynamics at nanometre resolution. *International Journal of High Performance Computing Applications* 2013; **29**(1):51–63.

DOI: 10.1002/cpe