Liao XK, Yang CQ, Tang T *et al.* OpenMC: Towards simplifying programming for TianHe supercomputers. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 29(3): 532–546 May 2014. DOI 10.1007/s11390-014-1447-4

OpenMC: Towards Simplifying Programming for TianHe Supercomputers

Xiang-Ke Liao[1] (廖湘科), *Member*, *ACM*, *IEEE*, Can-Qun Yang[1] (杨灿群), Tao Tang[1] (唐 滔)
Hui-Zhan Yi[1] (易会战), Feng Wang[1] (王 锋), *Member*, *CCF*, *ACM*, Qiang Wu[1] (吴 强), *Member*, *IEEE*, and Jingling Xue[2] (薛京灵), *Senior Member*, *IEEE*, *Member*, *ACM*

[1] *School of Computer Science, National University of Defense Technology, Changsha 410073, China*

[2] *School of Computer Science and Engineering, University of New South Wales, Sydney, Australia*

E-mail: {xkliao, canqun, taotang84, huizhanyi, fengwang, qiangwu}@nudt.edu.cn; jingling@cse.unsw.edu.au

Received August 29, 2013; revised January 21, 2014.

Abstract Modern petascale and future exascale systems are massively heterogeneous architectures. Developing productive intra-node programming models is crucial toward addressing their programming challenge. We introduce a directive-based intra-node programming model, OpenMC, and show that this new model can achieve ease of programming, high performance, and the degree of portability desired for heterogeneous nodes, especially those in TianHe supercomputers. While existing models are geared towards offloading computations to accelerators (typically one), OpenMC aims to more uniformly and adequately exploit the potential offered by multiple CPUs and accelerators in a compute node. OpenMC achieves this by providing a unified abstraction of hardware resources as workers and facilitating the exploitation of asynchronous task parallelism on the workers. We present an overview of OpenMC, a prototyping implementation, and results from some initial comparisons with OpenMP and hand-written code in developing six applications on two types of nodes from TianHe supercomputers.

Keywords supercomputer, programming model, heterogeneous, MIC

1 Introduction

Modern petascale and future exascale systems, which can process one quadrillion (10^{15}) FLOPS and one quintillion (10^{18}) FLOPS, respectively, are massively heterogeneous architectures comprising multi-core CPUs and accelerators. As heterogeneous systems achieve higher performance and energy efficiency than CPU-only systems, there are increasingly more accelerator-based supercomputers (using, e.g., GPUs[1] and Intel MIC[2]) appearing in the Top500① and Green500② lists.

The TianHe supercomputers, designed at the National University of Defense Technology (NUDT), are a few of the earliest large-scale systems embracing heterogeneous architectures. Announced in 2009, TianHe-1[3] achieved a peak performance of 1.206 PFLOPS and became China's first supercomputer beyond petascale. TianHe-1 consists of 2 560 compute nodes, each containing two Intel® Xeon® processors and an ATI Radeon™ HD4870×2 GPU. TianHe-1A[4], a subsequently upgraded system of TianHe-1, achieved 2.566 PFLOPS on the Linpack benchmark and ranked the first of Top500 list in November 2010. TianHe-1A, whose architecture is similar to that of its predecessor, consists of 7 168 compute nodes, each featuring two Intel® Xeon® X5670 processors and one Nvidia M2050 GPU. Recently, we have developed TianHe-2③, which consists of 16 000 nodes accelerated by Intel® Xeon® Phi™ co-processors. With a Linpack benchmark performance of 33.9 PFLOPS, TianHe-2 has been the fastest supercomputer in the world since June 2013.

Over the last few years, we have been looking for a suitable programming model for heterogeneous systems. In TianHe supercomputers, we relied on a hybrid model, in which MPI applies to inter-node program-

Regular Paper

This work is supported by the National High Technology Research and Development 863 Program of China under Grant No. 2012AA01A301, and the National Natural Science Foundation of China under Grant No. 61170049.

① http://www.top500.org, Mar. 2014.

② http://www.green500.org, Mar. 2014.

③ www.netlib.org/utk/people/JackDongarra/PAPERS/tianhe-2-dongarra-report.pdf, Mar. 2014.

ming and "OpenMP + X" to intra-node programming, where X is accelerator-specific, e.g., Brook+/CAL for AMD GPUs④, CUDA for NVIDA GPUs[5] or "Offload" for Intel MIC[2]. While MPI may continue to be the primary model for inter-node programming, "OpenMP + X" is becoming less appealing as it makes intra-node programming ad hoc, fairly complex, and quite difficult to obtain the degree of portability desired.

In search for a better intra-node programming model, we prefer a directive-based solution to CUDA[5] and OpenCL[6], because the former provides a higher-level abstraction for programming accelerators and facilitates incremental parallelization[7]. Several directive-based models, including OpenACC[8], PGI Accelerator⑤ and OpenHMPP⑥, have recently been introduced. By providing directive-based interfaces, they enable programmers to offload computations to accelerators and manage parallelism and data communication explicitly or implicitly.

To harness the full potential of TianHe supercomputers, existing directive-based intra-node programming models are inadequate in three aspects. First, a single code region, when offloaded, is usually executed as a solo task in an entire accelerator, resulting in its low utilization when the task exhibits insufficient parallelism. Second, as accelerator-oriented computing is emphasized, the ever-increasing processing power of general-purpose multi-core CPUs is neglected and thus inadequately exploited, especially for some irregular applications. Finally, multiple devices, i.e., multiple multi-core CPUs and multiple many-core accelerators are nowadays found in a single node. We need a little more than just offering a syntax to offload computations to accelerators. The ability to orchestrate the execution of multiple tasks efficiently across these devices becomes essential. We are not aware of any directive-based open-source compiler for TianHe-like heterogeneous systems accelerated by both GPUs and Intel MIC.

In this work, we are motivated to overcome the aforementioned three limitations. By building on recent advantages in the field, we make the following contributions:

• We introduce a directive-based intra-node programming model, OpenMC (Open Many-Core), towards simplifying programming for heterogeneous compute nodes, especially those used in the TianHe supercomputers. OpenMC provides a unified abstraction of the hardware resources in a compute node as workers, where a worker can be a single multi-core device or a subset of its cores (if this is permitted), and facilitates the exploitation of asynchronous task parallelism on the workers. As a result, OpenMC allows different types of devices to be utilized uniformly (without distinguishing CPUs from accelerators) and flexibly (with a device being utilized either wholly or partially).

• We present a prototype for OpenMC, including a basic runtime system and a compiler built on top of GCC.

• We show that OpenMC can achieve ease of programming, high performance, and the degree of portability desired. For the six representative applications running on two types of compute nodes (CPU-GPU and CPU-MIC) used in TianHe supercomputers, OpenMC obtains about 59.60% (56.50%) of hand-tuned performance with 11.20% (15.12%) coding overhead on CPU-GPU (CPU-MIC) on average. In addition, OpenMC outperforms OpenMP running on the CPUs only in two platforms by 1.87X and 2.47X on average.

The rest of this paper is organized as follows. Section 2 introduces OpenMC. Section 3 illustrates it with HPL as a significant example. Section 4 describes a prototype, including a basic runtime system and a compiler. Section 5 evaluates OpenMC on two types of heterogeneous nodes. Section 6 discusses the related work. Section 7 concludes the paper.

2 OpenMC Programming Model

We present an introduction to OpenMC, including its execution model, memory model, API, and runtime library.

2.1 Execution Model

Without loss of generality, we assume that MPI is used as the inter-node programming model. There can be several (MPI) processes running per node. OpenMC represents an intra-node programming model that allows programmers to orchestrate the execution of a single (MPI) process, called an *OpenMC program*, across the multiple devices in a node.

As shown in Fig.1, the OpenMC execution model provides two levels of abstraction for the "software" running on the "hardware". All the hardware resources in a compute node are available to an OpenMC program as a group of workers at the logical level. The tasks in an OpenMC program are organized at the software level in terms of a master thread, asynchronously executing agents that are dynamically spawned by the master (to run in slave threads) and accs (or accelerator regions)

④http://ati.amd.com/technology/streamcomputing, Mar. 2014.

⑤PGI Accelerator Compilers, Portland Group Inc., http://www.pgronp.com/, Apr. 2014.

⑥http://www.caps-entreprise.com/openhmpp-directives/, Mar. 2014.

offloaded to workers from an agent. In this paper, a task is an execution or invocation of an agent/acc. All the tasks (between two consecutive global synchronization points) will run in parallel unless their dependencies are explicitly annotated by programmers.

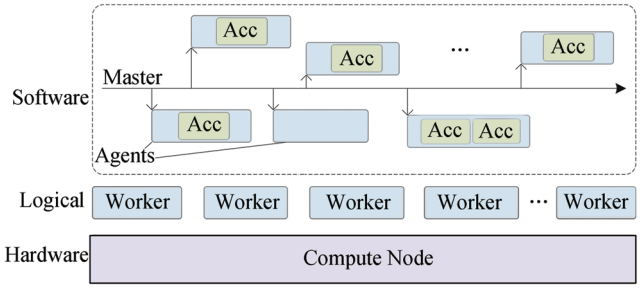


Fig.1. OpenMC execution model.

The four types of entities, worker, master, agent, and acc, are explained in more detail below.

• *Worker*. A worker represents a group of processing cores with the same shared memory space. A worker can be either an entire device (CPU or accelerator) or a subset of its cores. By default, the cores in a worker require hardware coherence, so that programmers can request the conventional multi-threaded codes (e.g., OpenMP or Pthread) to be scheduled and executed correctly on the worker. In the absence of hardware coherence (e.g., on GPUs), the codes must be written to ensure global memory consistency (e.g., in CUDA).

• *Master*. As the main thread of an OpenMC program, the master thread is responsible for managing agents and accs, handling I/O requests, and communicating with (the master threads of) the other OpenMC programs. Like an OpenMP program, the master thread of an OpenMC program is created and terminated at the entry and exit of the OpenMC program respectively. The master thread is executed serially on the host.

• *Agent*. An agent is a code segment that consists of some serial and parallel regions. As shown in Fig.1, an agent may have no parallel region, or have multiple parallel regions and organize them in terms of tightly coupled acc regions as well as handle the required data communication and synchronization. Spawned asynchronously by the master thread, an agent starts its execution as a task, with its serial regions running on the host and its accs running on some workers. The OpenMC runtime system decides when an agent can be scheduled and how.

• *Acc*. An acc region is a parallel region, a parallel loop, or a function, marked in an agent, which is executed on a worker. How an acc is mapped to a particular worker is determined by a *worker* directive (will be discussed in Subsection 2.3) or the OpenMC runtime system. To exploit the performance characteristics of some accelerators more effectively, an acc region can be written in a domain-specific language, such as CUDA.

We discussed the three limitations in Section 1 regarding existing directive-based intra-node programming models. OpenMC address them 1) by abstracting heterogeneous hardware resources in a compute node uniformly as workers and 2) by exploiting asynchronous task parallelism in an OpenMC program efficiently on the workers. In addition, our preliminary experience, as reported in Section 5, suggests that intra-node asynchronous task parallelism integrates well with inter-node MPI communications.

2.2 Memory Model

In principle, the OpenMC memory model is similar to those adopted by existing ones such as OpenACC[8] and OpenHMPP. In an OpenMC program, the serial code in an agent runs on the host, where the master thread runs. As a result, their program states are mutually visible. However, programmers are able to create agent-private copies for certain variables (e.g., loop variables) for a particular agent.

The cores in a worker share the same memory space. If two workers have different memory spaces, then data movement between the two spaces is either annotated by programmers or deduced by the OpenMC compiler. For example, explicit data transfers will happen for agents and acc regions running on workers with different memory spaces.

An agent is responsible for managing data communication and synchronization required by its accs. Presently, data movement between two workers must always be done via the host even if both represent accelerators.

2.3 API

We describe the OpenMC API, which enables programmers to annotate code in a style that is conceptually similar to OpenMP, except that OpenMP's sentinel "!$*omp*" is replaced by "!$*omc*". Presently, OpenMC can be used to annotate C programs.

Table 1 lists the OpenMC directives, together with their clauses, which fall into three categories: 1) hardware abstraction, which is defined by using the *worker* directive, 2) task management, which is accomplished by using *agent* and *acc* directives, and 3) synchronization, which is specified by *atomic*, *wait* and *bi-barrier* directives. As an agent/acc directive can be modified with data clauses specifying data movement, OpenMC does not have data directives.

At this stage, we have kept OpenMC simple and included only the core features essential to demonstrate

Table 1. OpenMC Directives and Their Clauses

Directive	Clauses
Worker	name(*type:id*[*:subid*1-*subid*2*:stride*], *string*)
Agent	flag(*expr*), deps(*expr*1[*:expr*2]), private(*list*), on(*list*), priority(n), time(*kind, expr*), copyin(*list*), copyout(*list*), copy(*list*), create(*list*)
Acc	flag(*expr*), deps([*expr*1*:*]*expr*2), time(*kind, expr*), copyin(*list*), copyout(*list*), copy(*list*), create(*list*), present(*list*), implemented-with(*list*)
Atomic	read, write, update, capture
Wait	all, flag(*expr*1[*:expr*2])
Bi-barrier	flag(*expr*)

its effectiveness. Once we have gained more experience with OpenMC, we expect to enrich OpenMC with more directives to turn it into a more productive programming model. Below we describe the current OpenMC directives in three subsections.

2.3.1 Hardware Abstraction

A *worker* directive is used to define a group of cores in a multi-core device (CPU or accelerator) with the same memory space as a worker. When its solo clause in Table 1 is used, the worker is named as *string* and comprises the cores specified in the range *subid*1-*subid*2*:stride* from the *id*-th device of type *type*. If only *type:id* is given, the worker denotes the entire device, which is typically the case for GPUs.

Note that workers with different names are logically different even if they share some cores. Programmers should be aware of the resulting performance implications if they opt to organize hardware resources in this way.

Just like a variable declaration in C, a worker directive introduces its name into the scope where it is declared. This enables different abstractions to be defined at different scopes to achieve better performance while also keeping annotated code better structured, as will be discussed in Subsection 3.4.

2.3.2 Task Management

In OpenMC, programmers use *agent* and *acc* directives for task management as described below.

Agent. An *agent* directive defines an agent as a code region containing some parallel loops and/or function invocations in an OpenMC program. During the execution of the master thread, an agent is asynchronously spawned to run as a relatively independent task in a (slave) thread. An *agent* directive plays a similar role as a *task* directive in OpenMP. The clauses given in Table 1 for an *agent* directive are straightforward, except for 1) *deps*, which allows task dependences to be identified, and 2) *on*, which suggests a list of executing workers for the accs contained in the agent being defined.

The acceptable clauses for an agent directive are:

• *flag*(*expr*). This clause assigns a unique identifier, obtained as the runtime value of *expr*, to this particular execution (or task) of the agent. Programmers must ensure that between two consecutive global synchronization points (in the master thread), different executions of the same or different agents have different IDs. Otherwise the program behavior is undefined.

• *deps*(*expr*1[:*expr*2]). This clause dictates that the current agent cannot start until the agent (acc) with an ID of *expr*1 (*expr*1 :*expr*2) has finished. The OpenMC runtime system will ensure that the dependence is satisfied before the current agent can proceed.

• *private*(*list*). This requests the variables on *list* from the master thread to be privatized in the current agent. This clause is often used when an agent is nested inside a loop to make its loop variable agent-private.

• *on*(*list*). This clause identifies all potential workers used for executing the acc regions in the current agent. When such an acc is ready, the OpenMC runtime system will schedule it to run on the first idle worker found from *list* and defer its execution otherwise. All the workers on *list* are restricted to share the same memory space. As a result, a clause on data movement using *copyin/copyout/copy/create* (described below) can be uniformly applied to all the workers.

• *priority*(n). This clause states that all the acc regions in the current agent will run with a priority of n. The lowest priority is 0, which is the default. The OpenMC runtime system will schedule a given set of acc regions in non-decreasing order of their priorities. This clause is associated with an *agent* directive instead of an *acc* directive, because all the accs in an agent are tightly coupled and will run with the same priority. Note that agents are not assigned priorities; their serial codes run on the host scheduled by the host OS.

• *time*(*kind, expr*). This clause advises the OpenMC runtime system to monitor the execution time of the agent for abnormal behavior. If the agent runs longer (shorter) than the time given in *expr* (in seconds) when *kind* is *ubound* (*lbound*), a warning is issued. This helps detect abnormal nodes in large-scale systems.

• *copyin/copyout/copy/create*(*list*). If an acc region in the current agent is scheduled to run on a worker that does not share the same memory space as the host, one of these data clauses can be used to define explicit data movement between the two memory spaces. Behaving similarly in OpenACC[8], *copyin*, *copyout*, and *copy* allocate the memory for the variables on *list* on the worker and copy data from the host to the worker,

or back, or both, before the agent starts its execution, and release the memory at its end. Note that *create* only allocates the memory for *list*.

In summary, an agent is a portion of an OpenMC program including a code region and a data region. All agents are asynchronously spawned from the master and run potentially in parallel unless their dependencies are annotated.

Acc. An *acc* directive marks a parallel code region in an agent, as illustrated in Fig.1. Among its nine clauses given in Table 1, seven are also applicable to an *agent* directive and two are new. The seven clauses, *flag*, *deps*, *time*, *copyin*, *copyout*, *copy*, and *create*, which can also modify an *agent* directive, have similar semantics.

A *flag*(*expr*) modifies an *acc* directive to assign a unique ID to this invocation of the acc. The accs in an agent must be executed with different IDs. By using a *deps*([*expr*1 :]*expr*2) clause, programmers can specify a dependent agent (acc) within an ID of *expr*1 :*expr*2 (*expr*2 in the same agent). Note that a data clause, *copyin/copyout/copy/create*, can modify both an *agent* directive and an *acc* directive. Whenever possible, data movement should be done at the level of agents to avoid redundant data transfers and exploit producer-consumer locality among the accs in an agent.

Let us examine the two new clauses for an *acc* directive:

• *present*(*list*). This clause states that the data on *list* are available on the worker executing the acc. Note that if no *copyin/copyout/copy/create/present* clause is given, then the OpenMC compiler will deduce which of the five clauses should be applied.

• *implemented-with*(*list*). OpenMP remains to be the most widely used standard for SMP[9]. In addition, a lot of legacy OpenMP code has been written. For these two reasons, OpenMC presently makes use of the OpenMP API, by default, to allow parallel regions, such as parallel loop and parallel section, to be identified as accs in an agent. For a worker that does not support OpenMP, an acc running on it can be a list of implementations written in the languages on *list* in that order, using e.g., *implemented-with*(CUDA,OpenCL). The corresponding compilers will be invoked to generate a correct executable for the acc in each case. By default, *implemented-with*(OpenMP) is assumed.

2.3.3 Synchronization

There are three types of synchronization directives, *atomic*, *wait*, and *bi-barrier*, which are used in the master and/or an agent but outside an acc. In an acc region, which is presently implemented in OpenMP or domain-specific languages such as CUDA and OpenCL, any synchronization operations required are implemented using these languages.

Atomic. As multiple agents run in different threads, accesses to some shared variables must happen atomically. An *atomic* directive is used to guarantee the atomicity of some operations performed by the master thread and asynchronously executing agents. The four associated clauses, *read*, *write*, *update*, and *capture* are carried over from the *atomic* directive in OpenMP 3.1.

Wait. A *wait* directive has two clauses:

• *all*. If the *wait* directive appears in the master thread, an *all* clause instructs the master to wait for all spawned agents to finish. In this case, the directive serves as a global synchronization point. If the *wait* directive appears in an agent, an *all* clause instructs the agent to wait for all its generated acc regions to finish. There is an implicit *wait* directive at the end of an agent. The default clause for a *wait* directive is *all*.

• *flag*(*expr*1[:*expr*2]). This clause instructs the underlying master/agent to wait until the dependent agent/acc specified has finished. If the runtime system fails to find the dependent agent/acc, this clause is ignored.

Bi-barrier. This directive allows a pair of agents to synchronize with each other at some specified points. A *bi-barrier* directive has only one clause, *flag*(*expr*). Consider two agents, $A1$ specified with *flag*(*expr*1) and $A2$ specified with *flag*(*expr*2). By inserting a *bi-barrier flag*(*expr*2) in agent $A1$ and a *bi-barrier flag*(*expr*1) in agent $A2$, the two agents perform a barrier synchronization at the points marked by the two directives. To avoid deadlock, the runtime system will allow one agent to resume its execution if it detects that the matching agent does not exist (since a matching *bi-barrier* directive is missing). In this case, such a *bi-barrier* behaves like a *wait* directive.

2.4 Runtime Library

OpenMC provides a set of runtime functions to help programmers manage workers and tasks. For example, programmers can call *bool omc_test_free*(*name*) to query the specified (logical) worker to see if it is busy executing a task or not. Whether the worker shares some cores with another worker physically is irrelevant for this query.

3 HPL: A Programming Example

We illustrate how to apply OpenMC to parallelize HPL (High-Performance Linpack) for a compute node in the TianHe-2 supercomputer. HPL is used by the Top500 (Green500) list to rank the fastest (most energy-efficient) supercomputers in the world.

Subsection 3.1 describes the architecture of a com-

pute node in TianHe-2. Subsection 3.2 reviews the HPL algorithm. Subsection 3.3 describes our OpenMC HPL for this node. In Subsection 3.4, we discuss how to define a logical worker abstraction and make the trade-off between performance and portability.

3.1 TianHe-2 Compute Node

As shown in Fig.2, a single compute node consists of two 8-core Intel® Xeon® CPUs and three Intel® Xeon® Phi™ coprocessors (MIC). Further architectural details about this node is given under "CPU-MIC" (Platform B) in Table 2.

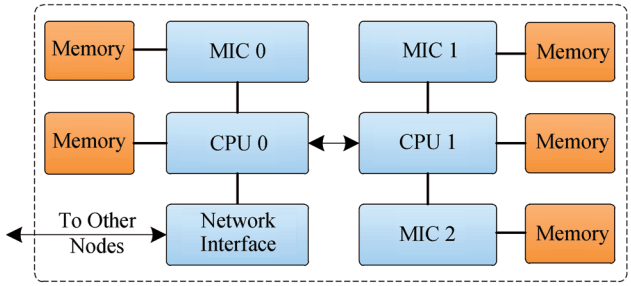


Fig.2. Compute node in TianHe-2.

Table 2. Two Types of TianHe Compute Nodes

Platform	A: CPU-GPU	B: CPU-MIC
CPU	Intel® Xeon® E5-2680	Intel® Xeon® E5-2670
Number of CPU	2	2
Accelerator	Nvidia M2050 GPU	Intel® Xeon® Phi™
Number of Accelerators	2	3
Memory	64 GB	96 GB
Host compiler	GCC4.7.2	GCC4.7.2
Accelerator compiler	CUDA-5.0	ICC 13.0.0

The two CPUs share the same memory space while each MIC has its own separated memory. As the cores in an MIC are cache-coherent, OpenMP is inherently supported.

3.2 HPL

The baseline algorithm[10] is sketched in the top of Fig.3. HPL solves a system $\boldsymbol{Ax} = \boldsymbol{b}$ of linear equations of by performing LU factorization with pivoting. As a blocked algorithm, $(\boldsymbol{A}, \boldsymbol{b})$ is partitioned into $nb \times nb$ blocks and block-cyclically distributed to a two-dimensional (2D) process grid.

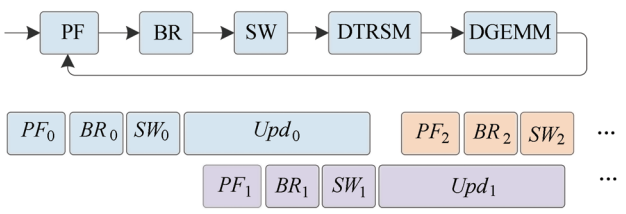


Fig.3. HPL: baseline and look-ahead algorithms.

For each participating process, each iteration of its main loop proceeds in five steps: 1) panel factorization (PF) performed on a panel of nb columns of the matrix allocated to the process, 2) broadcast (BR) for sending the factored panel to the other processes, 3) swap for pivoting (SW), 4) DTRSM, and 5) DGEMM for updating the trailing matrix, the remaining part of the matrix. DTRSM and DGEMM are amenable to acceleration with MIC as they are compute-intensive. BR and SW will run on the host as they are memory- and I/O-intensive (both within and across the nodes). PF can run on the host or accelerators.

To exploit the inherent parallelism on parallel systems, a look-ahead algorithm[10], as illustrated in the bottom of Fig.3 is used. Upd combines DTRSM and DGEMM to work on the trailing matrix. In this case, PF_i, BR_i and SW_i at iteration i can be performed in parallel with Upd_{i-1} from iteration $i-1$, subject to the underlying data dependencies.

3.3 OpenMC HPL

Fig.4 sketches our OpenMC version of the look-ahead HPL algorithm. In general, an OpenMC program works on a set of $nb \times nb$ blocks of a given matrix allocated to it in a block-cyclic manner. For simplicity, we assume that our OpenMC program works on a sub-matrix of size $N \times N$.

Hardware Abstraction. The fiver workers, including three MIC workers (M0∼M2) and two CPU workers (P0∼P1), are defined in lines 1∼5, one per device in a TianHe-2 node. A CPU worker consists of 16 threads exposed by the OS.

Task Management. There are seven agents, numbered 1∼7, which execute the tasks in HPL, as depicted in Fig.5. To facilitate their pipelined execution as discussed below, agent 2 runs at a priority of 1 and the remaining six at 0.

Agent 1 is invoked only once in the prologue, by performing PF_0 and BR_0 (on the first panel) and SW_0 at the first iteration on the host (lines 8∼11). The other six agents are invoked repeatedly inside the main loop (line 13). Agent 7 performs PF_i and BR_i (on the i-th panel) and SW_i for each subsequent iteration i on the host (lines 52∼55).

Agents 2∼6 (lines 15∼50) are responsible for performing Upd on the trailing matrix, denoted as $\boldsymbol{A'}$ below. Agent 2 (lines 15∼16) updates the first nb columns of $\boldsymbol{A'}$ on M0. Setting its priority to 1 allows it to be scheduled earlier, so that agent 7, which uses the updated columns of $\boldsymbol{A'}$ from agent 2 (indicated by $deps(i)$), can start working on the next iteration as soon

```
1  #pragma omc worker name(MIC:0,"M0")
2  #pragma omc worker name(MIC:1,"M1")
3  #pragma omc worker name(MIC:2,"M2")
4  #pragma omc worker name(CPU:0:0-15:1,"P0")
5  #pragma omc worker name(CPU:1:0-15:1,"P1")
6  ...
7  // prologue
8  #pragma omc agent //Agent 1
9  {
10   PF(0); BR(0); SW(0);
11 }
12 #pragma omc wait all
13 for(i = nb; i < N; i += nb) // main loop
14 {
15 #pragma omc agent flag(i) on(M0) priority(1) //Agent
       2
16   Upd(i, nb);
17 ...
18 #pragma omc atomic capture
19   { j = i + nb; }
20 #pragma omc agent on(M0) private(jstart) //Agent 3
21   while(1){
22     if(omc_test_free(M0))
23     {
24 #pragma omc atomic capture
25       {jstart = j; j += N_MIC;}
26       if(jstart < N)
27         Upd(jstart, N_MIC);
28       else break;
29     }
30     else continue;
31   }
32
33 /*codes on M1 and M2 are the same as that on M0 */
34 #pragma omc agent on(M1) private(jstart) //Agent 4
35 {...}
36 #pragma omc agent on(M2) private(jstart) //Agent 5
37 {...}
38
39 #pragma omc agent on(P0, P1) private(jstart) //Agent
       6
40   while(1){
41     if(omc_test_free(P0)||omc_test_free(P1))
42     {
43 #pragma omc atomic capture
44       {jstart = j; jstart += N_CPU;}
45       if(jstart < N)
46         Upd(jstart, N_CPU);
47       else break;
48     }
49     else continue;
50   }
51 ...
52 #pragma omc agent deps(i) //Agent 7
53   {
54     PF(i/nb); BR(i/nb); SW(i/nb);
55   }
56 #pragma omc wait all
57 }
58 ...
59 void Upd(jstart, n_col)
60 /* Update columns [jstart, min(jstart+n_col-1, N)] */
61 {
62   ...
63 #pragma omc acc flag(jstart)
64   DTRSM(jstart, n_col);
65 #pragma omc acc deps(jstart)
66   DGEMM(jstart, n_col);
67   ...
68 }
```

Fig.4. High-level sketch of the OpenMC HPL.

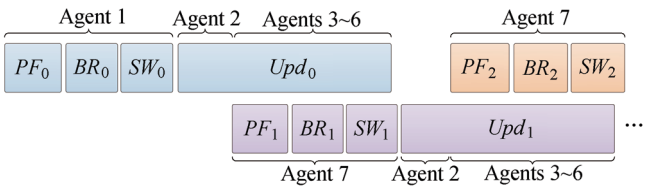


Fig.5. Seven agents in the OpenMC HPL.

as possible. After the first nb columns of $\boldsymbol{A'}$ are updated, its remaining columns are updated by agents 3∼6 inside four while loops. Agents 3, 4 and 5 run on M0, M1 and M2, respectively, while agent 6 runs on P0 and P1. Each MIC agent will asynchronously update N_MIC columns of $\boldsymbol{A'}$ each time and the CPU agent settles with N_CPU columns as long as their designated workers are free. Note that N_MIC and N_CPU are tunable parameters. As a result, all hardware resources are fully utilized.

The *Upd* function invoked by agents 2∼6 contains two acc regions, where the second one depends on the first one. To save space, their associated data clauses are omitted.

There are some differences between agents 1 and 6 even both run on the host. Agent 1, which contains no accs, runs on all the CPU cores of the host subject to the host OS. As for agent 6, its serial code runs on the host but its accs run on P0 and/or P1 under the OpenMC runtime system.

Synchronization. Each of the *wait* directives in lines 12 and 56 ensures that PF_i, BR_i and SW_i are done before Upd_i can start. The accesses to the shared variable j in the master thread and agents 3∼6 are safe as they are atomic.

3.4 Discussion

Worker Abstraction. Admittedly, programming massively heterogeneous systems is not easy. Given a large sequential program to parallelize with OpenMC, domain-expert programmers and computer scientists may often work together to define a suitable "worker abstraction" of the hardware resources in a node so that they can be uniformly and adequately exploited. Furthermore, some "model" abstractions for representative HPC applications can be made available for a given node, allowing them to be leveraged by programmers for other applications. Even if such an abstraction is not used, programmers would still need to spend the same or even more efforts thinking about how to offload computations to different devices in a node, albeit in ad hoc way.

Device Partitioning. In Fig.4, each MIC is a worker as a whole. However, an MIC coprocessor allows its subset of cores to be used independently as a worker, just like an SMP processor. This improves resource efficiency, thereby boosting application performance, as evaluated in Section 5.

Multiple Abstractions. According to Subsection 2.3.1, programmers can define different abstractions in different scopes, thereby improving load balance adaptively, especially for irregular applications. In the OpenMC HPL, as the iteration proceeds, the trail-

ing matrix becomes smaller and smaller. Agents 3∼5 will each have insufficient workload to saturate an MIC. In this case, programmers can use a different abstraction, by splitting an MIC into multiple workers and issuing more agents to run on the MIC, as demonstrated below:

```
1 if (the trailing matrix is smaller than a threshold)
      {
2 #pragma omc worker name(MIC:0:0-29:1, "M00")
3 #pragma omc worker name(MIC:0:30-59:1,"M01")
4 #pragma omc agent on(M00,M01) private(jstart)//Agent
      3
5 // similarly for Agents 4 and 5
6 }
```

In Section 5, we will evaluate the performance benefits of this adaptive scheme for HPL.

Portable Performance. The Holy Grail for software is portable performance. In reality, however, a tradeoff between portability and performance must be made. Given a compute node that is different from Fig.2, programmers are expected to provide a different worker abstraction and possibly modify the *on* clauses (for mapping accs to workers). The other directives for task management and synchronization may hopefully remain unchanged. However, for a new node with radically different devices (e.g., accelerators), a quite different parallelization strategy may be employed, regardless of which directive-based model is used.

To harness the full potential of TianHe supercomputers, we tend to trade portability for performance while still maintaining a unified abstraction in terms of workers across different OpenMC implementations of the same application for different compute nodes. In Section 5, we show that such abstraction helps reduce programming efforts when porting OpenMC applications across different platforms.

4 Reference Implementation

We describe a prototyping framework for OpenMC, as depicted in Fig.6, which consists of a compiler and a basic runtime system. Presently, OpenMC allows programmers to annotate programs written in C (the host language) and supports two types of accelerators, NVIDIA GPUs and Intel MIC. Accs running GPUs are written in CUDA, indicated with an *implemented-with*(*CUDA*) clause, and accs running on MIC (and the host) are parallelized using OpenMP, indicated with an *implemented-with*(*OpenMP*) clause.

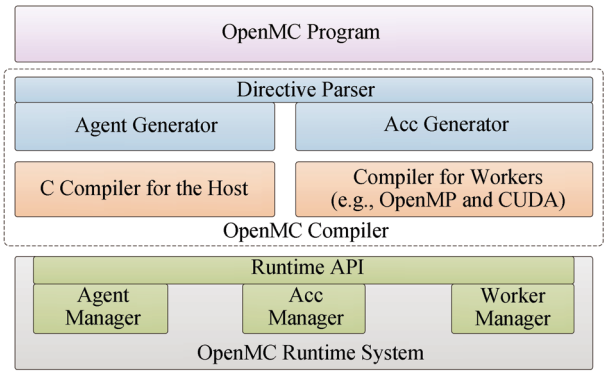


Fig.6. OpenMC framework.

4.1 OpenMC Compiler

Fig.7 outlines the source-to-source compilation process for an OpenMC program. Implemented in GCC 4.7, our compiler handles the OpenMC directives based on an extension of GCC's codebase for handling the OpenMP directives. The source code of an OpenMC program is first parsed by a directive parser. A directive of a particular type, such as worker, agent or atomic, is represented as a node of that type in the intermediate representation of the program.

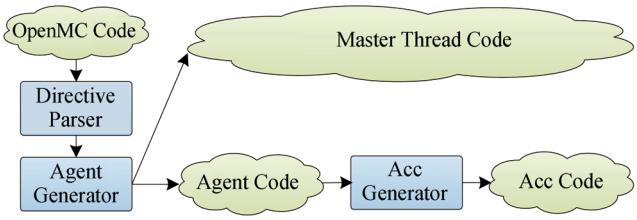


Fig.7. OpenMC compiler.

An OpenMC program, after its agents and accs have been processed, will be compiled by the host (C) compiler into the master thread, which can be viewed a special agent.

The agent generator processes each agent in an OpenMC program as follows. As an agent takes care of the data communication for its accs, the data clauses for the agent and its accs are replaced by data movement operations. In addition, every synchronization directive found is replaced by a call to an equivalent library function. Specifically, an *atomic*, *wait* or *bi-barrier* directive is replaced by a library call to *agent_atomic*, *agent_wait* or *agent_bi-barrier*. Then the transformed agent code is compiled into an executable by the host C compiler. Finally, the agent code is replaced with a library call to *agent_create*, by passing it with a pointer to the executable, together with agent-specific information collected from the corresponding agent directive.

The acc generator scans each *acc* directive in an agent and compiles its acc region (using a compiler for the language, specified with an implemented-with clause, in which the acc is written). Then the acc generator replaces the acc region by a library call to *acc_create*, by passing with it a pointer to the acc exe-

cutable, together with acc-specific information collected from its acc and agent directives.

4.2 OpenMC Runtime System

The runtime API includes those discussed in Subsection 2.4 and others like *agent_create* exposed to the OpenMc compiler. The three managers are described below.

4.2.1 Worker Manager

The worker manager simply maintains a list of workers, with the state of each worker being either busy or idle. Whenever a worker is allocated to run an acc or has finished executing an acc, its state is updated accordingly.

4.2.2 Agent Manager

As shown in Fig.8, this manager is responsible for creating, scheduling and, freeing agent objects. Four queues are maintained: "ready", "running", "suspend", and "finished".

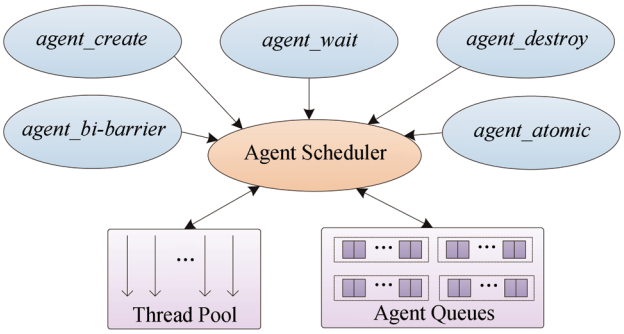


Fig.8. Architecture of the agent manager.

During the execution of the master thread, agents are asynchronously spawned. Every call to *agent_create* causes its associated agent to be created and initialized. Then the agent is inserted into the suspend queue if it has some dependent tasks specified by a *deps*(*expr*1[:*expr*2]) clause and the ready queue otherwise. Just after the agent has finished, its destructor, *agent_destroy*, is called to free the agent object.

During the execution of an agent, whenever some synchronization is performed, a call to *agent_atomic*, *agent_wait* or *agent_bi-barrier* is made. As shown in Fig.8, agent scheduling is triggered by an execution of any of the five library functions, called an *agent scheduling point*. Depending on the nature of a scheduling point, the agent manager will reorganize the four queues appropriately. For example, an agent will be moved into the ready queue if its dependent tasks are all in the finished queue or a global synchronization pointer is reached (in which the case some annotated dependent tasks are non-existent). The agent manager will issue a warning when a cyclic dependence is detected and will break the dependence arbitrarily. When scheduling an agent in the ready queue to run on the host, the agent manager will try to reuse an idle thread maintained in the thread pool for efficiency considerations.

4.2.3 Acc Manager

The acc manager, as shown in Fig.9, proceeds similarly as the agent manager, except for a few differences. First, the four queues of the same nature are maintained for scheduling accs. Second, *acc_create* and *acc_destroy* are the analogues of *agent_create* and *agent_destroy* for accs. Third, calls to *acc_create* and *acc_destroy* are the only acc scheduling points when acc scheduling is performed. Unlike agents, synchronization operations in an acc running on a worker do not affect acc scheduling. There are two cases. If the worker is an accelerator, then it executes the acc independently of the host. If the worker is the host, how the acc is executed depends on the host OS. At an acc scheduling point, the acc manager will schedule an acc from the ready queue with the highest priority possible to run on a free worker according to the *on* clause for the containing agent.

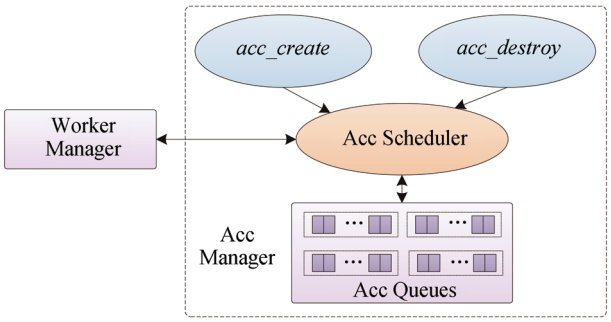


Fig.9. Architecture of the acc manager.

5 Experimental Evaluation

We use six representative programs on two types of TianHe compute nodes to evaluate OpenMC in terms of programmability, portability, and performance achieved. Due to the lack of directive-based open-source compilers supporting both GPUs and Intel MIC, we will compare OpenMC with OpenMP (on the host only) and hand-tuned implementations.

5.1 Platforms

Table 2 lists two types of compute nodes used. Platform *A* stands for an upgraded compute node in TianHe-1A[4] (enhanced with new CPUs and one more GPU) and platform *B* represents a node from TianHe-2 (shown in Fig.2).

Each platform consists of two eight-core CPUs, each being equipped with one or two GPUs/Xeon Phi™ accelerators. For CPU-GPU, the peak performance of two CPUs is 0.35 TFLOPS and that of two GPUs is 1.03 TFLOPS. For CPU-MIC, the peak performance of two CPUs is 0.33 TFLOPS and that of the three Xeon Intel® Phi™ coprocessors is 3 TFLOPS.

5.2 Benchmarks

Six representative applications are selected in HPC:

• `Swim` and `Mgrid`: taken from SPEC CPU2000[11], `Swim` is for shallow water wave modeling and `Mgrid` a multi-grid solver computing a 3D potential field;

• `HPL`[10]: well-known and illustrated in Section 3;

• `MPC`, `SWP` and `Lared-p`[12-13],⑦: these are real-world applications on molecular dynamics. MPC (Metal Particles Collision) simulates two blocks of copper particles moving with high velocity into each other. SWP (Shock Wave Propagation) simulates the propagation of shockwave through metallic foams. Lared-p, a 2D and 3D parallelized particle-in-cell (PIC) code series, is used to study laser-plasma interaction.

Our prototype, as shown in Fig.6, supports C only. `Swim` and `Mgrid` are available in Fortran. We have developed and used their equivalent versions in C below.

5.3 Programmability and Portability

To show that OpenMC simplifies programming, Table 3 gives a feel about programming efforts made in terms of LOC added/deleted/changed to the original programs by comparing OpenMC with OpenMP and hand-tuning for platforms A and B. In hand-crafted versions, the platform-specific APIs (CUDA for A and SCIF/COI for B), are used. Note that annotating a program may necessitate code deletion and modification to make it more amenable to annotation.

In the "OpenMP" column, the result for HPL is not given. The HPL program contains no OpenMP directives itself, as it relies on the BLAS library for parallelization.

Table 3. LOC Added/Deleted/Changed

Applications	Original	OpenMP (%)	OpenMC (%)		Hand-Tuned (%)	
			A	B	A	B
`Swim`	435	5.98	12.41	19.08	23.14	26.10
`Mgrid`	490	14.69	19.80	23.06	25.01	32.97
`HPL`	34 270	-	9.82	14.10	29.88	43.25
`MPC`	2 561	3.05	7.26	12.18	29.78	35.33
`SWP`	2 320	3.75	9.87	13.28	32.47	36.43
`Lared-p`	4 570	3.28	8.00	8.99	29.96	38.45
Avg. Δ	-	6.15	11.20	15.12	28.37	35.42

To annotate each program, programmers write slightly more code with OpenMC than with OpenMP but much less than with hand-turning. On average, the LOC added/deleted/changed for OpenMP, OpenMC, and hand-tuning are 6.15%, 11.20% on A and 15.12% on B, and 28.37% on A and 35.42% on B, respectively. Note that for each program, Intel MIC is harder to program than GPUs for both OpenMC and hand-tuning. Platform A boasts one more accelerator than platform B. Under OpenMC, more agent and acc regions are therefore required for B than A, as explained shortly, in order to fully exploit the three accelerators. In hand-tuning, the hand-tuned vectorization in terms of Intel's intrinsics for Intel MIC on platform B incurs some more coding efforts.

We have collected more statistics to understand further programming complexity reduced and the degree of portability supported by OpenMC. Table 4 lists the number of workers, agents, accs, and synchronization directives introduced (under columns 2∼5) for a program. Note that the number of directives added to a program on a platform (column 6) is smaller than the corresponding total LOC added in Table 3, due to some necessary modifications made to the program to facilitate its annotation, as mentioned earlier.

To assess the degree of portability provided when migrating OpenMC programs between platforms, column "$A \rightarrow B$" gives the total LOC modified to the OpenMC programs written for A (CPU-GPU) when ported to B (CPU-MIC). Due to the architectural differences, some

Table 4. Statistics on the OpenMC Directives

Applications	Number of Workers		Number of Agents		Number of Accs		Number of Synchronization		Total		$A \rightarrow B$
	A	B	A	B	A	B	A	B	A	B	(%)
`Swim`	4	8.0	20.0	28.0	20	28.0	4.0	4	48.0	68.0	11.83
`Mgrid`	4	8.0	22.0	32.0	22	32.0	10.0	12	58.0	84.0	16.06
`HPL`	4	15.0	158.0	194.0	86	98.0	75.0	88	323.0	395.0	7.72
`MPC`	4	14.0	42.0	59.0	42	59.0	18.0	24	106.0	156.0	10.38
`SWP`	4	8.0	36.0	48.0	36	48.0	15.0	26	91.0	130.0	7.85
`Lared-p`	4	11.0	61.0	70.0	40	40.0	15.0	20	120.0	141.0	7.87
Avg.	4	10.7	56.5	71.8	41	50.8	22.8	29	124.3	162.3	10.28

⑦ http://www.iapcm.ac.cn/jasmin/index.php?page=lared-p, Apr. 2014.

agent and acc regions had to be re-organized. However, on average, only 10.28% of the code was modified. In addition, the programmer who originally annotated a program for A will spend much less time migrating the annotated program for A to B, compared with when the programmer annotates the program directly for B. Our estimated saving in terms of programming time is 63.3%. This validates our claim in Subsection 3.4 that by providing a unified abstraction of hardware resources, OpenMC makes it easier to migrate OpenMC programs between platforms.

5.4 Performance

OpenMC is an intra-node programming model. We evaluate OpenMC by comparing it with OpenMP and hand-tuning. For the six benchmarks selected, the OpenMC and hand-tuned programs will run on platforms A and B. The OpenMP programs will run on the two CPUs only in each platform.

5.4.1 *Platform A: CPU-GPU*

Table 5 gives the problem sizes used for the six benchmarks on this platform. Fig.10 plots the speedups of OpenMP, OpenMC and hand-tuning against the serial execution time. With 16 cores on both CPUs, OpenMP achieves an average speedup of 10.6x, with `HPL` enjoying the highest speedup (15.8x) due to the abundance of parallelism in the code. Accelerated further by two GPUs, OpenMC and hand-tuning obtain better speedups, 19.7x and 33.1x, respectively, on average, with `HPL` still being the best performer. Note that `Lared-p` is not sped up much in all the three cases as it has a large portion of serial code. `Swim` is the second worst due to its low ratio of compute-to-memory access and a large number of I/O operations incurred.

Table 5. Problem Sizes on Platform A (CPU-GPU)

Applications	Size
`Swim`	*ref* dataset
`HPL`	$N = 40\,000$
`SWP`	1 680 000 particles
`Mgrid`	*ref* dataset
`MPC`	8 000 particles
`Lared-p`	686 000 particles

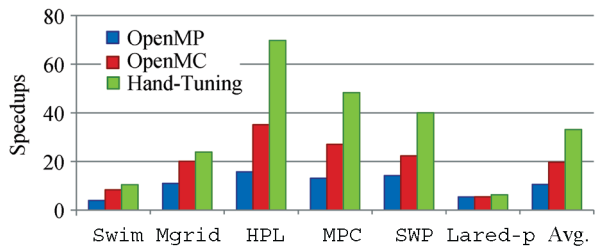


Fig.10. Speedups on platform A (CPU-GPU).

For each benchmark, the acc regions in its OpenMC version are written in CUDA and compiled by the NVCC compiler. These accs are expected to run as efficiently as the corresponding parts in the hand-tuned version. However, the hand-tuned version is (expectedly) much faster, because its code has been crafted to hide communication latency as much as possible by overlapping CPU computation, GPU computation, and CPU-GPU communication. This is one area where OpenMC will be improved in future work.

Fig.11 shows the speedups achieved by OpenMC and hand-tuning for the four real-world applications with several different problem sizes. Just like hand-tuning, OpenMC is scalable. `Lared-p` does not scale well in both cases because it has a large portion of serial code, as noted earlier.

5.4.2 *Platform B: CPU-MIC*

Table 6 lists the problem sizes used for the six benchmarks for CPU-MIC. Fig.12 is an analogue of Fig.10 for CPU-MIC. As in the case of CPU-GPU, `HPL` remains to be the best performer under OpenMP, OpenMC, and hand-tuning on CPU-MIC. By comparing Figs. 10 and 12, we observe that OpenMP achieves similar speedups on both platforms: 10.6x on CPU-GPU and 10.4x on CPU-MIC. However, OpenMC and hand-tuning have delivered better average performance improvements on CPU-MIC than on CPU-GPU. The average speedups of OpenMC and hand-tuning are 25.58x and 45.3x, respectively.

Table 6. Problem Sizes on Platform B (CPU-MIC)

Applications	Size
`Swim`	*ref* dataset
`HPL`	$N = 40\,000$
`SWP`	2 250 000 particles
`Mgrid`	*ref* dataset
`MPC`	120 000 particles
`Lared-p`	1 372 000 particles

In addition to tapping into the full potential of multiple accelerators, OpenMC also allows the full-processing power of CPUs to be exploited (uniformly as workers). Take `HPL` as an example. Its speedup achieved by OpenMC will drop from 33.88x to 24.4x if the two CPUs, i.e., P0 and P1 (in lines 4∼5 in Fig.4) are not used.

Fig.13 is an analogue of Fig.11, demonstrating again the scalability of OpenMC on platform B (CPU-MIC).

As in the case of CPU-GPU, hand-tuning delivers much better performance than OpenMC for our benchmarks on CPU-MIC, for the same key reason. The hand-tuned code has been elaborately designed to maximize computation and communication overlap by using the SCIF/COI API for MIC.

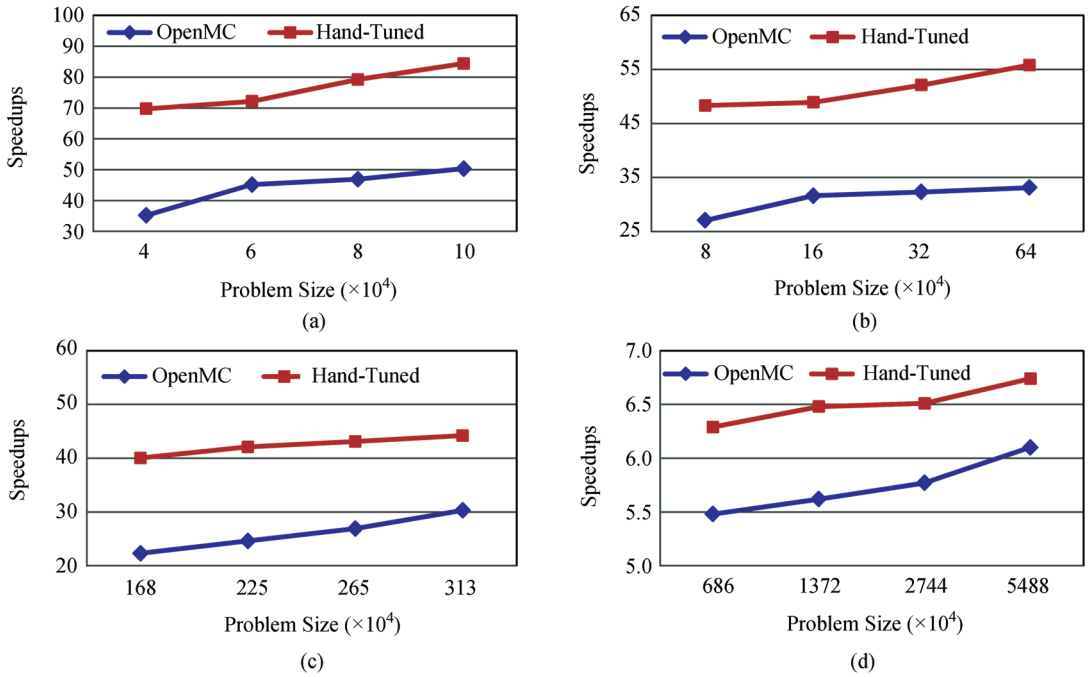


Fig.11. Scalability of OpenMC and hand-tuned code on platform A (CPU-GPU). (a) `HPL`. (b) `MPC`. (c) `SWP`. (d) `Lared-p`.

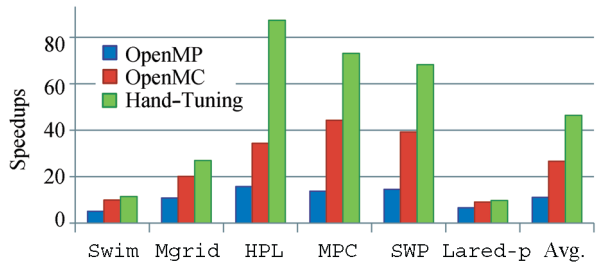


Fig.12. Speedups on platform B (CPU-MIC).

In Fig.12, each MIC is used entirely as one worker. As discussed in Subsection 3.4, when the trailing sub-matrix is small enough, we can split a single MIC into multiple workers and map multiple agents to run on the same MIC. Fig.14 demonstrates some potential improvements obtained using this parallelization strategy, denoted as OpenMC-SPLIT.

In this experiment, each MIC is partitioned into two equal-sized workers when the trailing sub-matrix has reached half of the original matrix. By providing better load balancing, OpenMC-SPLIT outperforms OpenMC by 21.75%, 19.57%, 23.23% and 25.53% in the four problem sizes tested.

Finding a best partitioning strategy is beyond the scope of this work, which may be obtained based on domain knowledge, compiler analysis, and performance tuning. However, OpenMC provides a unified abstraction of hardware resources that allows such parallelization schemes to be explored.

6 Related Work

CUDA[5] and OpenCL[6] are the most widely used programming models for heterogeneous (predominantly CPU-GPU) systems. CUDA is vendor-specific for NVIDIA GPUs only. In contrast, OpenCL offers a unified programming interface for a variety of accelerators. For example, SnuCL is an OpenCL framework recently proposed for heterogeneous CPU/GPU clusters[14]. While CUDA and OpenCL are the two most common programming environments for GPU accelerators, programming directly at this level is considered to be complex and error-prone, making it difficult to achieve portability and correctness for non-expert programmers.

There are a number of attempts on leveraging existing parallelizing compiler techniques, which are originally developed for multiprocessors, to enable CPU-GPU systems to be programmed with traditional languages, models and/or environments[15-26]. Some of these attempts focus on translating C to CUDA[19], OpenMP to CUDA[21-23], OpenACC to CUDA[27] and X10 to CUDA[20].

Recently, directive-based programming of GPUs or Intel MIC has become more prevalent, because it provides better productivity than CUDA and OpenCL, allows incremental parallelization, and achieves reasonable performance[7]. A number of directive-based programming models have been introduced, including OpenACC[8], PGI Accelerator, OpenHMPP, hiCU-

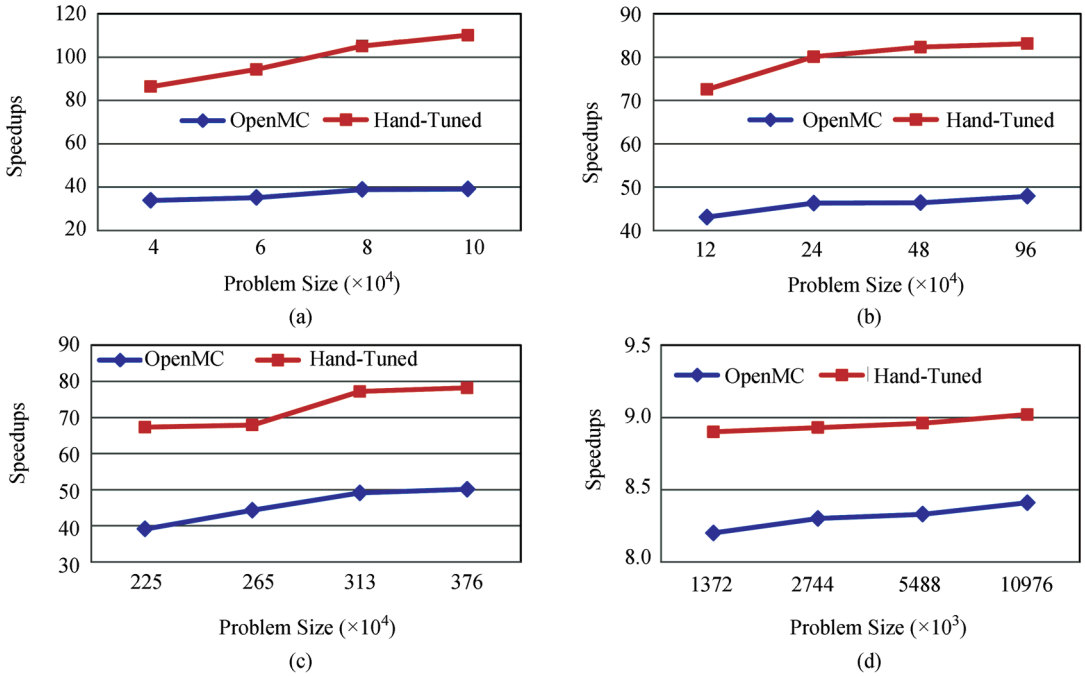


Fig.13. Scalability of OpenMC and hand-tuned code on platform B (CPU-MIC). (a) `HPL`. (b) `MPC`. (c) `SWP`. (d) `Lared-p`.

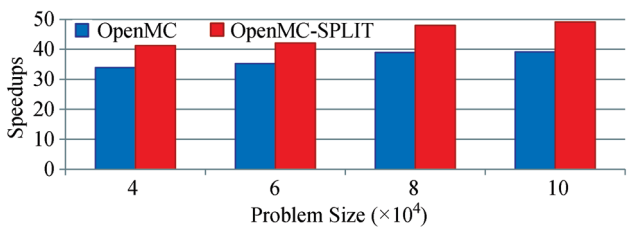


Fig.14. Speedups of HPL with and without splitting a single MIC into two equally-split workers.

DA[28], OmpSs[29], Lime[30-31] and "Offload"[32]. Furthermore, the OpenMP Program Committee has recently set up a sub-committee to develop an accelerator-oriented programming model[33].

A directive-based approach is promising for programming massively heterogeneous systems. However, as discussed in Section 1, existing directive-based models are inadequate for programming compute nodes in such large-scale systems, since they are mostly designed to offload a single task to a single device, failing to exploit asynchronous task parallelism uniformly and adequately across the multiple CPUs and accelerators in a node. While OpenHMPP supports multiple accelerators, we are not aware of any directive-based open-source compiler that supports multiple accelerators, including both GPUs and Intel MIC. The OpenACC/OpenHMPP-based CAPS compilers are proprietary. OpenMC aims to address these limitations by providing a unified abstraction of the hardware resources (CPUs, GPUs, and Intel MIC) as workers in a node and by exploiting asynchronous task parallelism across all the workers.

Asynchronous parallelism is widely supported in asynchronous PGAS models, such as UPC[34], X10[35], and Chapel[36]. While simplifying programming by exposing an abstracted shared space to programmers, these models are focused on distributed shared-memory clusters.

In addition to automatic parallelization and directive-based programming for heterogeneous systems, Thrust[37] provides library interfaces that abstract the details of these systems. Phalanx[38] represents a unified programming model for heterogeneous systems, with an asynchronous task model providing constructs for launching large, structured collections of cooperating threads, thus supporting both coarse-grained task parallelism and fine-grained thread parallelism.

Accelerators such as GPUs are constantly improving. For example, Hyper-Q supported in NVIDIA's Kepler allows concurrent GPU kernel execution from multiple processes and Intel® Xeon Phi™ allows multiple tasks to share the same MIC. Programmers may shoulder the responsibility of massaging the code to produce the desirable performance for a program. Experiences show that such responsibility presents a major burden on even expert programmers. As a directive-based model, OpenMC provides a higher-level abstraction, allowing programmers to specify directives to guide the

compiler in tuning program performance. However, some performance-critical code regions can still be written by experts, if desired, using an *implemented-with* clause.

7 Conclusions

In this paper, we addressed one of the programming challenges for heterogeneous systems by introducing OpenMC, a new model for intra-node programming. OpenMC provides a unified abstraction for the hardware resources in a node as workers and emphasizes asynchronous task parallelism, which, we believe, is critical for harnessing the full potential of petascale and exascale systems. An OpenMC worker can be either an entire multi-core device or a subset of its cores, so as to provide a flexible hardware abstraction mechanism. We introduced a prototype implementation for OpenMC, including a compiler and a runtime system. Our preliminary experience from our experiments suggests that OpenMC is promising in terms of programmability, portability and performance achieved.

One future work is to improve OpenMC to support computation and communication overlap to narrow the performance gap between OpenMC and hand-tuning. Such improvement is of great importance for heterogeneous systems with separate address spaces. Another is to add new directives to provide an application-level fault-tolerance API. During the development of large-scale parallel computing systems, we observe that some nodes exhibit abnormal performance behaviors from time to time, which does not cause a system crash but can slow down the performance of the entire system considerably. Such abnormal performance behaviors are often difficult to locate and fix. We have found that associating a *time* clause with an *agent* directive is useful in detecting these abnormal nodes, especially in TianHe-2 with 16 000 compute nodes.

References

[1] Owens J, Luebke D, Govindaraju N *et al.* A survey of general purpose computation on graphics hardware. *Computer Graphics Forum*, 2007, 26(3): 80-113.

[2] Sherlekar S. Tutorial: Intel many integrated core (MIC) architecture. In *Proc. the 18th ICPADS*, Dec. 2012, p.947.

[3] Yang X, Liao X, Xu W *et al.* TH-1: China's first petaflop supercomputer. *Frontiers of Computer Science in China*, 2010, 4(4): 445-455.

[4] Yang X, Liao X, Lu K *et al.* The TianHe-1A supercomputer: Its hardware and software. *Journal of Computer Science and Technology*, 2011, 26(3): 344-351.

[5] Kirk D. NVIDIA CUDA software and GPU parallel computing architecture. In *Proc. International Symposium on Memory Management*, Oct. 2007, pp.103-104.

[6] Gaster B, Howes L, Kaeli D *et al.* Heterogeneous Computing with OpenCL — Revised OpenCL 1.2 Edition. Morgan Kaufmann, 2013.

[7] Lee S, Vetter J. Early evaluation of directive-based GPU programming models for productive exascale computing. In *Proc. Int. Conf. High Performance Computing, Networking, Storage and Analysis*, Nov. 2012, Article No.23.

[8] Wienke S, Springer P, Terboven C *et al.* OpenACC: First experiences with real-world applications. In *Proc. the 18th Int. Conf. Euro-Par Parallel Processing*, Aug. 2012, pp.859-870.

[9] Chapman B, Gropp W, Kumaran K *et al* (eds.). OpenMP in the Petascale Era Springer, 2011.

[10] Petitet A, Whaley R, Dongarra J *et al.* HPL — A portable implementation of the high-performance linpack benchmark for distributed-memory computers, Sept. 2008. http://www.netlib.org/benchmark/hpl/, Mar. 2014.

[11] Henning J. SPEC CPU2000: Measuring CPU performance in the new millennium. *Computer*, 2000, 33(7): 28-35.

[12] Plimpton S. Fast parallel algorithms for short-range molecular dynamics. *J. Computational Physics*, 1995, 117(1): 1-19.

[13] Zhang A, Mo Z. Parallelization of lared-p codes for simulation of laser plasma interactions. Technical Report, ZW-J-2002045, Institute of Applied Physics and Computational Mathematics, 2002.

[14] Kim J, Seo S, Lee J *et al.* SnuCL: An OpenCL framework for heterogeneous CPU/GPU clusters. In *Proc. the 26th ACM Int. Conf. Supercomputing*, Jun. 2012, pp.341-352.

[15] Cui H, Wang L, Xue J *et al.* Automatic library generation for BLAS3 on GPUs. In *Proc. IEEE Int. Parallel and Distributed Processing Symposium*, May 2011, pp.255-265.

[16] Di P, Wan Q, Zhang X *et al.* Toward harnessing DOACROSS parallelism for multi-GPGPUs. In *Proc. the 39th Int. Conf. Parallel Processing*, Sept. 2010, pp.40-50.

[17] Di P, Xue J. Model-driven tile size selection for DOACROSS loops on GPUs. In *Proc. 2011 Int. Conf. Euro-Par Parallel Processing*, Aug. 2011, pp.401-412.

[18] Diogo M, Grelck C. Towards heterogeneous computing without heterogeneous programming. In *Proc. the 13th Int. Symp. Trends in Functional Programming*, June 2012, pp.279-294.

[19] Baskaran M, Ramanujam J, Sadayappan P. Automatic C-to-CUDA code generation for affine programs. In *Proc. the 19th Int. Conf. Compiler Construction*, Mar. 2010, pp.244-263.

[20] Cunningham D, Bordawekar R, Saraswat V. GPU programming in a high level language: Compiling X10 to CUDA. In *Proc. the 2011 ACM SIGPLAN X10 Workshop*, Jun. 2011, Article No.8.

[21] Ohshima S, Hirasawa S, Honda H. OMPCUDA: OpenMP execution framework for CUDA based on Omni OpenMP compiler. In *Proc. the 6th Int. Workshop. OpenMP*, June 2010, pp.161-173.

[22] Lee S, Min S, Eigenmann R. OpenMP to GPGPU: A compiler framework for automatic translation and optimization. In *Proc. the 14th PPoPP*, Feb. 2009, pp.101-110.

[23] Lee S, Eigenmann R. OpenMPC: Extended OpenMP programming and tuning for GPUs. In *Proc. the 2010 ACM/IEEE Int. Conf. High Performance Computing, Networking, Storage and Analysis*, Nov. 2010, pp.1-11.

[24] Hormati A, Samadi M, Woh M *et al.* Sponge: Portable stream programming on graphics engines. In *Proc. the 16th Int. Conf. Architectural Support for Programming Languages and Operating Systems*, Mar. 2011, pp.381-392.

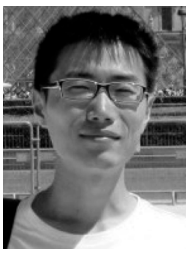
[25] Yang Y, Xiang P, Kong J *et al.* A GPGPU compiler for memory optimization and parallelism management. *ACM SIGPLAN Notices*, 2010, 45(6): 86-97.

[26] Wu B, Zhao Z, Zhang E *et al.* Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on GPU. In *Proc. the 18th PPoPP*, Feb. 2013, pp.57-68.

[27] Reyes R, Lopez I, Fumero J *et al.* accull: An user-directed approach to heterogeneous programming. In *Proc. IEEE the 10th ISPA*, Jul. 2012, pp.654-661.
[28] Han T, Abdelrahman T. hiCUDA: A high-level directive-based language for GPU programming. In *Proc. the 2nd Workshop on General Purpose Processing on Graphics Processing Units*, Mar. 2009, pp.52-61.
[29] Duran A, Ayguadé E, Badia R *et al.* OmpSs: A proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 2011, 21(2): 173-193.
[30] Auerbach J, Bacon D, Burcea I *et al.* A compiler and runtime for heterogeneous computing. In *Proc. the 49th Annual Conference on Design Automation*, Jun. 2012, pp.271-276.
[31] Dubach C, Cheng P, Rabbah R *et al.* Compiling a high-level language for GPUs: (Via language support for architectures and compilers). In *Proc. the 33rd PLDI*, Jun. 2012, pp.1-12.
[32] Cooper P, Dolinsky U, Donaldson A *et al.* Offload–automating code migration to heterogeneous multicore systems. In *Proc. the 5th HiPEAC*, Jan. 2010, pp.337-352.
[33] Beyer J, Stotzer E, Hart A *et al.* OpenMP for accelerators. In *Proc. the 7th Int. Conf. OpenMP in the Petascale Era*, June 2011, pp.108-121.
[34] UPC Consortium. UPC language specifications v1.2. Technical Report LBNL-59208, Lawrence Berkeley National Lab, 2005. http://upc.gwu.edu/docs/upc_specs_1.2.pdf, Mar. 2014.
[35] Saraswat V, Bloom B, Peshansky I *et al.* X10 language specification version 2.4. Technical Report, IBM, January 2012, http: //x10.sourceforge.net / documentation/languagespec/x-10-latest.pdf, Mar. 2014.
[36] Chamberlain B, Callahan D, Zima H. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications*, 2007, 21(3): 291-312.
[37] Hwu W W. GPU Computing Gems Jade Edition. Morgan Kaufmann, 2011.
[38] Garland M, Kudlur M, Zheng Y. Designing a unified programming model for heterogeneous machines. In *Proc. the International Conference on High Performance Computing, Networking, Storage and Analysis*, Nov. 2012, Article No.67.

Xiang-Ke Liao received his B.S. and M.S. degrees both in computer science from Tsinghua University, Beijing, and National University of Defense Technology (NUDT), Changsha, in 1985 and 1988, respectively. He is now a professor and the dean at the School of Computer Science, NUDT. His research interests include parallel and distributed computing, high-performance computer systems, operating system, and networked embedded system. He is a member of IEEE and ACM.

Can-Qun Yang received the M.S. and Ph.D. degrees both in computer science from NUDT, in 1995 and 2008, respectively. Currently he is a professor at NUDT. His research interests include programming languages and compiler implementation. He is the major designer dealing with the compiler system of the TianHe supercomputer.



Tao Tang received his Ph.D. degree in computer science from the School of Computer science, NUDT, in 2011. He is currently an associate professor at the university. His research interests lie in high performance computing and compiler optimizations.

Hui-Zhan Yi received his Ph.D. degree in computer science from NUDT. He is currently an associate professor at the university. His research interests include programming languages, parallel programming, and compiler optimizations and verifications.

Feng Wang received his Ph.D. degree in computer science from NUDT, in 2013. He is currently an associate professor at the university. His research interests include programming languages, parallel programming, and compiler optimizations and verifications. He is a member of CCF and ACM.

Qiang Wu received his M.S. and Ph.D. degrees both in computer science from NUDT, in 2009 and 2013, respectively. His research interests include compiler techniques for high performance, compiler techniques for embedded systems, and parallel programming. He is a member of IEEE.

Jingling Xue received his B.S. and M.S. degrees in computer science and engineering from Tsinghua University in 1984 and 1987, respectively. He received his Ph.D. degree in computer science and engineering from Edinburgh University in 1992. He is currently a professor of computer science and engineering at the University of New South Wales (UNSW). He leads the Programming Languages and Compilers Group and its subgroup Compiler Research Group (CORG) at UNSW. His research interests are programming languages, compiler optimisations, computer architecture, parallel computing, distributed systems and cluster computing, and embedded systems.