International Journal of Networking and Computing – www.ijnc.org
ISSN 2185-2839 (print) ISSN 2185-2847 (online)
Volume 7, Number 2, pages 387–404, July 2017

Toward Dynamic Load Balancing across OpenMP Thread Teams for Irregular Workloads

Xiong Xiao
Graduate School of Information Sciences, Tohoku University
6-3 Aramaki-Aza-Aoba, Aoba-ku, Sendai, 980-8578, Japan

Shoichi Hirasawa
Information Systems Architecture Science Research Division, National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo, 101-8430, Japan

Hiroyuki Takizawa
Cyberscience Center, Tohoku University
6-3 Aramaki-Aza-Aoba, Aoba-ku, Sendai, 980-8578, Japan

and

Hiroaki Kobayashi
Graduate School of Information Sciences, Tohoku University
6-6-01 Aramaki-Aza-Aoba, Aoba-ku, Sendai, 980-8579, Japan

Received: February 16, 2017
Revised: May 6, 2017
Accepted: June 2, 2017
Communicated by Akihiro Fujiwara

Abstract

In the field of high performance computing, massively-parallel many-core processors such as Intel Xeon Phi coprocessors are becoming popular because they can significantly accelerate various applications. In order to efficiently parallelize applications for such many-core processors, several high-level programming models have been proposed. The de facto standard programming model mainly for shared-memory parallel processing is OpenMP. For hierarchical parallel processing, OpenMP version 4.0 or later allows programmers to create multiple thread teams. Each thread team contains a bunch of newly-created synchronizable threads. When multiple thread teams are used to execute an application, it is important to have dynamic load balancing across thread teams, since static load balancing easily encounters load imbalance across teams, and thus degrades performance. In this paper, we first motivate our work by clarifying the benefit of using multiple thread teams to execute an irregular workload on a many-core processor. Then, we demonstrate that dynamic load balancing across those thread teams has a potential of significantly improving the performance of irregular workloads on a many-core processor, with considering the scheduling overhead. Although such a dynamic load balancing mechanism has not been provided by the current OpenMP specification, the benefits of dynamic load balancing across thread teams are discussed through experiments using the Intel Xeon Phi coprocessor. We evaluate the performance gain of dynamic load balancing across thread teams using a ray tracing code. The results show that such a dynamic load balancing mechanism can improve the

performance by up to 14% compared to static load balancing across teams, with considering scheduling overhead.

Keywords: OpenMP, Thread team, Load balancing, Irregular workload

1 Introduction

Recent years, accelerators and coprocessors such as Intel Xeon Phi [11] have gained more and more attention, especially in the field of high performance computing (HPC), because they sometimes achieve several orders of magnitude higher performance in comparison with general-purpose processors, i.e., central processing units (CPUs). The significant performance gain is due to the massively parallel architecture of the coprocessors, which allows a computation code to be executed in a highly-parallel fashion.

Thanks to the powerful computing capability of coprocessors, computing systems equipped with both CPUs and coprocessors, so-called heterogeneous systems, have been widely used to accelerate the execution of HPC applications. This paper uses a system equipped with Intel Xeon CPUs and Intel Xeon Phi coprocessors as an example of the heterogeneous systems. In the heterogeneous system, Xeon Phi is used as an accelerator to speedup the execution of data-parallel portion of an application. On the other hand, CPU handles non data-parallel parts, such as serial code execution or data movement between the CPU and Xeon Phi coprocessor. Although Intel commercial products are used as a case study in this paper, dynamic load balancing across teams introduced in this paper is applicable to the systems from other vendors.

Since many-core processors need massive parallelism for efficient execution, programming for such a processor to exploit its computing power is becoming a challenge. Fortunately, there exist several high-level programming models (e.g., OpenMP [5] and OpenACC [21]) for parallel programming of many-core processors. OpenMP is one of the standards for parallel programming. It recently provides a *target* construct for offloading a parallel computation to a target device. Moreover, it provides a *teams* construct to create multiple thread teams to execute a computation on the target device. A *teams* construct must be contained within a *target* construct. A thread team is a team of newly-created synchronizable threads, and multiple thread teams are called a thread league in the OpenMP terminology. The maximum number of created thread teams can be specified using a *num_teams* clause, and the maximum number of threads in each team can be specified using a *thread_limit* clause. Introducing the concepts of a thread league and teams can provide a hierarchical execution model for exploiting nested parallelism in the OpenMP programming. A thread team can be newly created, and it works independently from other spawning threads and their teams. In this work, an OpenMP *target* construct is used for offloading parallel executions on a Xeon Phi coprocessor, and a *teams* construct is used to create multiple thread teams on the coprocessor.

The execution model of OpenMP is a fork-join model. When a thread encounters a *parallel* construct, it creates a parallel region that is executed by a thread team. The encountering thread becomes the master thread of the team. When a thread finishes its computation, it waits for the others in the current team at an implicit barrier at the end of the parallel region. When all threads in the current team reach the barrier, they join the master thread.

Load balancing is a fundamental issue for data-parallel execution in order to achieve a high performance. In fact, programmers often find that the performance of parallel execution is far worse than the peak performance because of workload imbalance. During execution, some threads are busy with computation while others might have less computation to deal with. What is worse, some threads remain completely idle until others finish their work. It will badly hurt the computing efficiency of parallel execution. Overall, load imbalance leads to performance degradation of parallel execution. Conversely, if programmers can perfectly balance the workload among threads, it is expected that they can achieve a better performance.

In many cases, the iterative computation of a particular loop nest consumes most of the execution time of a given application. Such a loop nest is called a *hotspot* of the application. There are several recent studies [7, 23, 31] focusing on optimizing the workload balance of the hotspots so as to improve the overall performance of an application. In this work, we are concentrating on load balancing for

such hotspots executed on a Xeon Phi coprocessor using OpenMP directives.

This work assumes that irregular loop nests are executed on one Xeon Phi coprocessor using OpenMP directives. A loop nest is considered *irregular* if the execution time of each iteration significantly varies. If the execution time of such a loop nest is dominant in the total execution time of an application, the application is considered an irregular workload. Programmers need to carefully manage the workloads across multiple thread teams as well as within a thread team, because load imbalance can easily occur for irregular loop nests. As shown later in this paper, using multiple thread teams is better than using a single one for irregular loop nests in terms of improving performance.

The current OpenMP specification supports both static and dynamic loop scheduling mechanisms within a thread team, and also supports a static scheduling mechanism across multiple thread teams. When multiple thread teams are used to execute an application, it is strongly required to have dynamic load balancing across thread teams, since static load balancing easily encounters load imbalance across teams, and thus degrades performance. Therefore, this work focuses on dynamically balancing the workloads across multiple thread teams created by OpenMP directives. Theoretically, the optimal workload distribution should be achieved when multiple teams complete their computations at the same time.

In this paper, we have two general assumptions. First, for a specific application, the overhead of inter-team dynamic scheduling can be estimated using the overhead of intra-team scheduling of a single-team execution. This assumption implies that the data exchange overhead among thread teams is similar to that among threads. Since thread teams as well as threads can share a memory region, we expect that the data exchange overhead among teams does not greatly differ from that among threads. Second, the overhead of inter-team dynamic scheduling increases exponentially with the number of thread teams. The results in [15] have shown that the overhead of intra-team dynamic scheduling is likely to increase exponentially with the number of threads. Similarly, the second assumption is considered reasonable. Based on the first assumption, we estimate the overhead of inter-team dynamic scheduling for two-team execution using the overhead of intra-team dynamic scheduling. Then, we further estimate the overhead of inter-team dynamic scheduling for more than two teams based on the second assumption. In this paper, a scheduling overhead only refers to a dynamic scheduling overhead, and no scheduling overhead is assumed in static scheduling.

In OpenMP, intra-team dynamic scheduling methods use a so-called task pooling algorithm [13, 16], which allows a scheduler to distribute loop iterations chunk by chunk to the threads in a team. The chunk size is the number of iterations in the chunk, and it can be tuned to achieve a high performance. Usually, the overhead of dynamic scheduling decreases as the chunk size increases, as discussed in [15]. Hence, the intra-team dynamic scheduling overhead is maximum when the chunk size is *1*. Though a large chunk size reduces the scheduling overhead, it might lead to load imbalance that degrades performance. Thus, the chunk size needs to be tuned with considering a trade-off between load balancing and the scheduling overhead.

The objective of this work is to clarify the advantage of dynamic load balancing among thread teams. In this paper, we use an irregular workload code as a target application. First, a motivating example shows that use of multiple thread teams could be important to achieve a better performance in the case of executing irregular workloads. Then, we use a static load balancing method to emulate dynamic load balancing across thread teams. Afterwards, the inter-team scheduling overhead is estimated using the intra-team scheduling overhead. Finally, we evaluate the performance gain of dynamic load balancing with an estimated scheduling overhead.

The contributions of this paper are threefold: First, it clarifies the advantage of execution using multiple thread teams. For irregular applications, execution with multiple thread teams outperforms that with a single thread team in terms of execution performance. Second, it discusses the overhead of inter-team dynamic load balancing under the aforementioned assumptions. Third, it clarifies the benefit of dynamic load balancing across thread teams for irregular workloads. Static load balancing might lead to a poor performance because irregular workloads can easily encounter load imbalance across thread teams. To the best of our knowledge, this work is the first to discuss the dynamic load balancing issue across thread teams for irregular workloads. In this work, a dynamic load balancing mechanism is emulated by using a static one for the discussion.

```
for(int t=0;t<N;t++){   //   'N' phases
    Ray cam = change_camera_position_here (t);
    for (int y=0; y<h; y++){   // Image columns
        samps = change_sampling_rate_here (t, y);
        for (int x=0; x<w; x++){   // Image rows
          for (int sy=0, i=(h-y-1)*w+x; sy<2; sy++){   //Subpixel
            for (int sx=0; sx<2; sx++, r=Vec()){   // Subpixel
              for (int s=0; s<samps; s++){   // Sampling
                  /*Calculate radiance */
                  /* It depends on materials of objects*/
                  /* It also depends on camera position & direction*/
              }
              /*Accumulate radiance */
            }
          }
        }
    }
}
```

Figure 1: A code example of ray tracing. N phases are created. The camera position and direction, as well as the sampling rate, are changed from phase to phase.

The rest of this paper is organized as follows. Section 2 shows a motivating example of this work, and discusses the importance of dynamic load balancing among thread teams. Section 3 describes the related work. Section 4 discusses our approach to evaluation of inter-team dynamic load balancing. It also discusses a method to evaluate the overhead of intra-team dynamic load balancing. Section 5 shows the evaluation results, and Section 6 gives concluding remarks.

2 Importance of dynamic load balancing across thread teams in the OpenMP programming model

In this section, a motivating example is shown using a ray tracing code. The ray tracing algorithm [10, 26] is one of the basic techniques to generate a photo-realistic image in computer graphics. It renders an image by tracing the paths of rays when they encounter objects. Fig. 1 shows an example of the hotspot of the ray tracing algorithm, which is based on the code available at [2], and modified to render multiple images by adding the outermost loop and changing the camera position and the sampling rate. Let a phase be a time period, during which an independent unit of work is executed by a given application. In the code segment in Fig. 1, we assume that there are N phases, and one image is generated in each phase. By changing the camera position and the sampling rate for each image generation, the execution time of each phase could be significantly varied. Moreover, the execution time of each iteration of the innermost loop varies drastically. Accordingly, the modified ray tracing loop nest is irregular.

This section shows that multiple thread teams rather than a single one are required to achieve a higher performance with the code. Multiple teams in OpenMP are originally introduced for nested parallelism. However, in this experiment, our purpose is not to provide such nested parallelism for the ray tracing code. Our intention is to investigate the performance changes according to the number of thread teams. Therefore, in the experiment, we measure the execution times for different image sizes by changing the number of thread teams. The number of thread teams and the thread count in each team can be specified using the OpenMP directives. The scheduling method within a team can

Table 1: The experimental environment in motivating experiment

OS	CentOS Release 6.7
Host	Intel Xeon CPU E5-2690 @2.9GHz
Device	Intel Xeon Phi 5110P, 60 cores in total
Compiler	Intel compiler version 16.0.2 with -O3 option

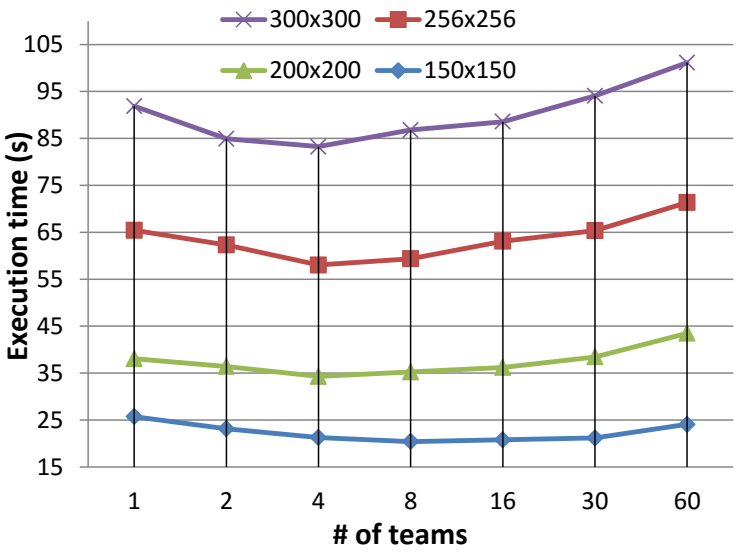

Figure 2: Performances with different numbers of thread teams across various image sizes.

also be designated. The number of teams varies from 1 to 60, while we maintain the overall thread count as 240 regardless of how many teams are created. OpenMP's dynamic scheduling mechanism within a team, and the static scheduling mechanism across teams, are used in this experiment. The experimental environment is shown in Table 1.

The results of the above experiment are shown in Fig. 2. Different curves represent different image sizes. The horizontal axis represents the number of thread teams used for execution, and the vertical axis represents the execution time in seconds. Fig. 2 illustrates that a different number of teams leads to a different performance even if the image size is the same. Furthermore, multiple teams outperform a single one across all image sizes when the number of teams is less than 16. If the number of teams is too large such as 60, it might lead to a worse performance than using a single team, because too few threads in each team hurt the parallelism. Thus, from the above motivating experiment, we conclude that using multiple teams is potentially better than using a single team for irregular loop nests, even if the inter-team scheduling mechanism is a static one.

The reason why multiple thread teams outperform a single one is that the execution using multiple thread teams can reduce the number of OpenMP threads joining internal synchronization at runtime, because threads in different teams do not need to synchronize while threads within a team need to synchronize. As shown in [8, 15], the synchronization overhead increases with the number of threads that are involved in the synchronization. Therefore, the synchronization overhead of multiple-team execution is smaller than that of single-team execution.

The above motivating experiment shows that using multiple thread teams is often a better choice than using a single team to execute irregular workloads. If multiple thread teams are used to execute an irregular workload, it requires that the workloads among the thread teams are balanced, in order to achieve a high performance. However, a static load balancing mechanism might easily suffer from load imbalance, especially for irregular workloads. Therefore, this paper clarifies the expected performance gain by introducing a dynamic load balancing mechanism across thread teams.

3 Related Work

The OpenMP specification [5] provides a *target* construct to support offloading workloads to a target device for acceleration. An *omp teams* construct creates a league of thread teams on the target device, and the master thread of each team executes the associated code region. A thread league consists of a set of thread teams, while a thread team consists of a set of synchronizable threads. Currently, OpenMP supports both static and dynamic scheduling mechanisms within a thread team. Programmers can easily specify a scheduling mechanism by using a *schedule* clause in the OpenMP directive. However, only a static scheduling mechanism is supported across different thread teams. For irregular codes, static scheduling may lead to a poor performance. Thus, our work aims to examine the benefits of dynamic scheduling across thread teams.

Many researches focus on dynamic load balancing for heterogeneous systems. Wang et al. [30] have proposed an asymptotic profiling method to schedule loop iterations between a CPU and a GPU. Ren et al. [29] have proposed a method to test various work distributions between a CPU and a GPU to find the most efficient distribution. Boyer et al. [6] have presented a dynamic load balancing mechanism that requires no offline training and responds automatically to performance variability. Other approaches, such as in [17, 20, 28], use modest amounts of initial training to select the workload distribution. Scogland et al. [27] also have proposed a dynamic load balancing mechanism across CPUs and GPUs. They consider different computation patterns of the accelerated code region. All those researches only concern about how the workload should be distributed in the heterogeneous systems. They do not care about fine-grained thread scheduling either on hosts or devices. Unlike other researches, our work focuses on dynamic scheduling of the workloads across thread teams on a device. Although we use Intel Xeon Phi as an example of the device, the dynamic scheduling mechanism is applicable to other devices.

Some related studies use dynamic scheduling for irregular applications. Durand et al. [14] have proposed a new OpenMP loop scheduler that is able to perform dynamic load balancing while taking memory affinity into account for irregular applications. Min et al. [19] have presented BANBI, a dynamic scheduling method for irregular programs on many-core systems. It is specifically designed for stream programs. Their applications do not use OpenMP device constructs for execution on a target device. They mainly schedule the iterations among different threads or different cores. Our work uses high-level OpenMP device constructs for offloading the workloads to a device. We specifically use multiple thread teams to execute the codes, and demonstrate that dynamic scheduling across thread teams is crucial for high performance.

In addition to some centralized scheduling methods [18, 30] for dynamic load balancing, work stealing [1, 4, 12, 22] is a popular scheduling method. It is proved that work stealing is efficient for scheduling some multi-threaded executions. Prokopec et al. [25] have presented a work-stealing algorithm for irregular data-parallel workloads. Their method allows workers to decide the workload distribution in a lock-free, workload-driven manner. However, work stealing requires data synchronization whenever a steal happens. Frequent steals might introduce much overhead, and thus degrade performance. In addition to general-purpose scheduling methods, some studies have proposed domain-specific scheduling mechanisms for dynamic load balancing. Their domains vary from fluid dynamics [32] to linear algebra [31]. All those works do not address dynamic load balancing across OpenMP thread teams.

Extensive studies [3, 8, 9, 15, 24] have been performed to measure the overheads of OpenMP worksharing and mutual exclusion directives in the literature. Bull [8] has measured the synchronization overheads of some basic OpenMP constructs, and it also has introduced a method to measure the overhead of intra-team dynamic scheduling. In this work, we use the idea in [8] to measure the intra-team dynamic scheduling overhead. More details are described in Section 4.2. Fredrickson et al. [15] have presented performance characteristics (e.g., synchronization and scheduling overheads) of OpenMP constructs on a large symmetric multiprocessor using various benchmark suites. The results in [15] have shown that intra-team dynamic scheduling overhead is likely to increase exponentially with the number of threads. Based on the results, we make a similar assumption that inter-team dynamic scheduling overhead exponentially increases with the number of thread teams.

Table 2: Six scenarios

Scenarios	Description
SS	Single team, static scheduling within the team
SD	Single team, dynamic scheduling within the team
MSS	Multiple teams, static across teams, static within each team
MSD	Multiple teams, static across teams, dynamic within each team
MDS	Multiple teams, dynamic across teams, static within each team
MDD	Multiple teams, dynamic across teams, dynamic within each team

4 Methodology

4.1 Emulation of inter-team dynamic load balancing

This subsection presents our methodology to discuss the benefit of a dynamic load balancing mechanism across multiple thread teams.

Since inter-team dynamic scheduling has not been supported by the current OpenMP specification, we use static load balancing to mimic dynamic one. We assume that a given application is executed through multiple phases (e.g., N phases, from phase 1 to phase N). A workload ratio r_j (j = *0, M-1* and M is the number of thread teams) is defined using Eq. (1).

$$r_j = \frac{Number\ of\ iterations\ executed\ by\ team\ j}{Total\ number\ of\ iterations\ executed\ by\ all\ teams}. \tag{1}$$

A workload vector v is defined as a vector of the workload ratios, as shown in Eq. (2).

$$v = (r_0, r_1, ..., r_{M-1}). \tag{2}$$

For each phase, a workload vector should be adjusted so that all teams finish their computations at the same time. However, the current OpenMP specification does not support dynamical adjustment of the workload vector. Thus, with a fixed workload vector, we execute the application from beginning to end while measuring the execution time of each phrase. The execution is repeated while changing the workload vector. As a result, we can obtain the best workload vector for each phase that can minimize the execution time of the phase. By summing up the shortest execution times of all phases, we can obtain the total execution time of ideal dynamic load balancing that can perfectly predict the best workload vector for every phase without any runtime overhead.

For example, if $ti(v)$ represents the execution time of phase i (i = *1, N*) when the workload vector is v, the execution time d of an application using dynamic load balancing can be expressed using Eq. (3).

$$d = \sum_{i=1}^{N} \min_{v} t_i(v). \tag{3}$$

Although a dynamic load balancing mechanism has not been implemented for the OpenMP programming model, this performance model can be used to estimate the benefit of introducing such a mechanism. The above performance model ignores the runtime overhead of dynamic load balancing. However, we can estimate the potential performance gain by such a dynamic load balancing mechanism if we ignore the scheduling overhead and also assume that the workloads can be balanced immediately and perfectly. Then, based on the estimation, we discuss whether the performance gain could be larger than the dynamic scheduling overheads.

4.2 Scheduling overhead

In this work, we have two assumptions discussed in Section 1. Based on the first assumption, the overhead of inter-team dynamic scheduling for two-team execution is estimated using the overhead

```
#pragma omp parallel
for (j=0; j<repeats; j++){
    #pragma omp for schedule (schetype, chunksize)
    for (i=0; i<itersperthread * omp_get_num_threads(); i++){
        /*Loop Body*/
    }
}
```

Figure 3: Loop (nest) *L1* parallelized by OpenMP directives

```
for (i=0; i<itersperthread; i++){
    /*Loop Body*/
}
```

Figure 4: Loop (nest) *L2* executed using a single thread

of intra-team dynamic scheduling. Then, the overhead of inter-team dynamic scheduling for more than two teams is further estimated based on the second assumption. By combining the performance model in Eq. (3) with the estimated scheduling overhead, we can discuss the performance gain of inter-team dynamic load balancing with runtime overheads.

A method to evaluate the overhead of intra-team dynamic load balancing is described in [8]. We employ the method to evaluate the intra-team dynamic scheduling overhead in this work. The method in [8] is briefly described as follows. Assume that a loop (nest) is parallelized using an OpenMP *parallel for* constructs, denoted as *Loop (nest) L1*, as shown in Fig. 3. We also have a serialized version of the same loop (nest), denoted as *Loop (nest) L2*, as shown in Fig. 4. Our goal is to measure the scheduling overhead of Loop *L1*. First, we measure execution time *t1* of Loop *L1*. Then, we measure execution time *t2* of Loop *L2* using a single thread. Finally, the scheduling overhead t is given by Eq. (4).

$$t = \frac{|t1 - t2|}{repeats}. \tag{4}$$

If the *Loop body*s shown in Figs. 3 and 4 are replaced with the ray tracing loop nest, we can measure the intra-team scheduling overhead of the ray tracing code. The scheduling type can be designated as *dynamic*. To find the upper bound of the overhead, we let the chunk size be *1*. Note that the intra-team scheduling overhead can be used to estimate the inter-team scheduling overhead.

5 Evaluations

5.1 Experimental setup

We use a modified ray tracing code as the target irregular code shown in Fig. 1. The system used for the evaluation is the same as that in Section 2.

Table 3: The best workload vectors in each phase for both movies when two teams are used for execution.

Best workload vector	MDS-1	MDD-1	MDS-2	MDD-2
Phase 1	(0.7, 0.3)	(0.65, 0.35)	(0.65, 0.35)	(0.65, 0.35)
Phase 2	(0.6, 0.4)	(0.6, 0.4)	(0.65, 0.35)	(0.65, 0.35)
Phase 3	(0.7, 0.3)	(0.65, 0.35)	(0.7, 0.3)	(0.65, 0.35)
Phase 4	(0.7, 0.3)	(0.65, 0.35)	(0.65, 0.35)	(0.65, 0.35)
Phase 5	(0.25, 0.75)	(0.35,0.65)	(0.2, 0.8)	(0.35, 0.65)
Phase 6	(0.3, 0.7)	(0.3, 0.7)	(0.35, 0.65)	(0.35, 0.65)
Phase 7	(0.35, 0.65)	(0.4, 0.6)	(0.35, 0.65)	(0.35, 0.65)
Phase 8	(0.5, 0.5)	(0.5, 0.5)	(0.35, 0.65)	(0.35, 0.65)

5.2 Performance of ideal inter-team dynamic load balancing

We compare the performances of six different scenarios shown in Table 2 using the modified ray tracing code. SS stands for single-team execution with static scheduling, which means a single-team execution where static scheduling is used within the team. SD stands for single-team execution with dynamic scheduling, which means a single-team execution where dynamic scheduling is used within the team. MSS stands for multiple-team execution with static scheduling across teams and within each team, which means multiple-team execution where static scheduling is used across teams and within each team. MSD stands for multiple-team execution with static scheduling across teams and dynamic scheduling within each team, which means multiple-team execution where static scheduling is used across teams and dynamic scheduling is used within each team. MDS stands for multiple-team execution with dynamic scheduling across teams and static scheduling within each team, which means multiple-team execution where dynamic scheduling is used across teams and static scheduling is used within each team. MDD stands for multiple-team execution with dynamic scheduling across teams and within each team, which means multiple-team execution where dynamic scheduling is used across teams and within each team. In the six scenarios, SS, SD, MSS, and MSD are already supported in the current OpenMP specification. Their performances can be evaluated in a straightforward way. However, MDS and MDD are not supported yet. Thus, we use the proposed methodology described in Section 4 to mimic MDS and MDD.

In the evaluation, we create eight phases to execute the ray tracing loop nest, which means $N = 8$ in Fig. 1. One image is rendered in each phase. The image size is 256 x 256 pixels in the evaluation. By changing the camera position and direction from phase to phase, we can create a movie. We create two movies by moving the camera in two different ways. MDS-1 and MDD-1 are used to represent the two scenarios for the first movie, while MDS-2 and MDD-2 are used for the second movie. For each movie, we statically divide the iteration space of loop y into M (M is the number of teams) parts. Each part is executed by a corresponding thread team. An example is shown in Fig. 5 when the number of teams M = 2. By varying the workload vector, we measure the execution time with each vector. The workload ratio of a team is at least 5%, increased by 5%. Then, we find the best workload vector that leads to the best performance regarding the phase. Finally, the best execution time of each phase is accumulated to estimate the total execution time with MDS or MDD.

In Section 5.2, we use two thread teams as well as four teams to perform the evaluation. Section 5.2.1 presents the results using two teams, and Section 5.2.2 presents the results using four teams.

5.2.1 Results using two thread teams

The results using two teams are presented as follows. Table 3 lists the best workload vectors in each phase of MDS and MDD, for both movies. In Table 3, we can see that the best workload vectors in each phase are different from (0.5, 0.5), except in Phase 8. It means that workload imbalance exists in almost all the phases. Figs. 6 and 7 show the execution times for phases in

```
for(int t=0;t<N;t++){   //   'N' phases
    Ray cam = change_camera_position_here (t);
    if (omp_get_team_num() == 0) {
      for (int y=0; y<h*workload_ratio; y++){   // Image columns
        samps = change_sampling_rate_here (t, y);
        for (int x=0; x<w; x++){   // Image rows
          for (int sy=0, i=(h-y-1)*w+x; sy<2; sy++){   //Subpixel
            for (int sx=0; sx<2; sx++, r=Vec()){   // Subpixel
              for (int s=0; s<samps; s++){   // Sampling
                  /*Calculate radiance */
                  /* It depends on materials of objects*/
                  /* It also depends on camera position & direction*/
              }
              /*Accumulate radiance */
            }}}}
    }   //End if
    else if (omp_get_team_num() == 1) {
      for (int y=h*workload_ratio; y<h; y++){   // Image columns
        samps = change_sampling_rate_here (t, y);
        for (int x=0; x<w; x++){   // Image rows
          for (int sy=0, i=(h-y-1)*w+x; sy<2; sy++){   //Subpixel
            for (int sx=0; sx<2; sx++, r=Vec()){   // Subpixel
              for (int s=0; s<samps; s++){   // Sampling
                  /*Calculate radiance */
                  /* It depends on materials of objects*/
                  /* It also depends on camera position & direction*/
              }
              /*Accumulate radiance */
            }}}}
    }   // End else if
} // End t=0,N
```

Figure 5: Statically dividing the iteration space of loop y when M = 2

various scenarios. Figs. 6 and 7 shows the results for the first movie and second movies, respectively. In these figures, different curves represent different scenarios. MDS and MDD can be either optimal or suboptimal. The optimal MDS (MDD) means that the best workload ratio in each phase is used, and the suboptimal MDS (MDD) means that the second-best workload ratio in each phase is used to discuss the performance gain with imperfect prediction. In Figs. 6 and 7, we can see that execution times vary from phase to phase for a specific scenario. This is because the ray tracing code is an irregular workload. Moreover, we can see that the optimal MDD is the best scenario among the six scenarios, for both movies. In the rest of this paper, unless otherwise stated, MDS (MDD) represents the optimal MDS (MDD).

Fig. 8 shows the total execution times for both movies when two teams are used for execution. The average execution times of both movies are also shown. In Fig. 8, we can see that dynamic scheduling within a thread team is always better than static scheduling. This is because dynamic scheduling improves the load balance within a thread team. Moreover, MDS outperforms MSS, and MDD outperforms MSD, for both movies. On average, MDS achieves 14.5% performance improvement in comparison with MSS, and MDD achieves 10.9% performance improvement in comparison with

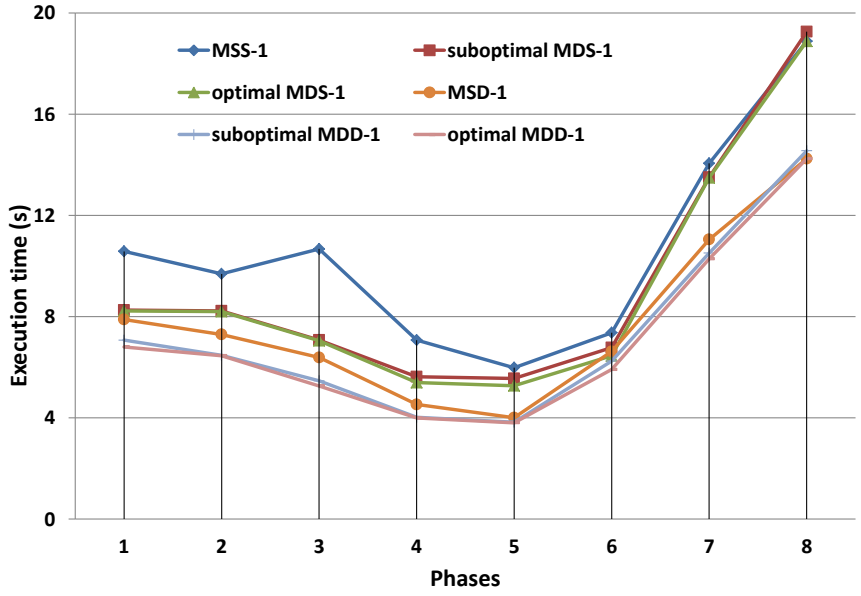

Figure 6: The execution times for phases of the first movie. Two teams are used for the execution. MDS and MDD can be either optimal or suboptimal. The optimal MDS means that the best workload ratio in each phase is used, and the suboptimal MDS means that the second-best workload ratio in each phase is used to discuss the performance gain with imperfect prediction.

Table 4: The best workload vectors in each phase for both movies when four teams are used for execution.

	MDS-1	MDD-1	MDS-2	MDD-2
Phase 1	(0.5,0.25,0.15,0.1)	(0.5,0.25,0.15,0.1)	(0.5,0.25,0.15,0.1)	(0.5,0.25,0.15,0.1)
Phase 2	(0.55,0.25,0.1,0.1)	(0.55,0.25,0.15,0.05)	(0.55,0.25,0.15,0.05)	(0.5,0.25,0.15,0.1)
Phase 3	(0.55,0.25,0.15,0.05)	(0.5,0.25,0.15,0.1)	(0.5,0.25,0.2,0.05)	(0.5,0.25,0.15,0.1)
Phase 4	(0.4,0.3,0.2,0.1)	(0.45,0.3,0.15,0.1)	(0.4,0.3,0.2,0.1)	(0.4,0.3,0.2,0.1)
Phase 5	(0.15,0.2,0.3,0.35)	(0.15,0.2,0.25,0.4)	(0.1,0.15,0.25,0.5)	(0.15,0.2,0.25,0.4)
Phase 6	(0.1,0.15,0.2,0.55)	(0.1,0.15,0.25,0.5)	(0.15,0.2,0.25,0.4)	(0.15,0.2,0.25,0.4)
Phase 7	(0.1,0.15,0.3,0.45)	(0.15,0.2,0.3,0.35)	(0.2,0.2,0.3,0.3)	(0.15,0.2,0.25,0.4)
Phase 8	(0.25,0.25,0.25,0.25)	(0.25,0.25,0.25,0.25)	(0.2,0.2,0.3,0.3)	(0.15,0.2,0.25,0.4)

MSD. Thus, dynamic scheduling across thread teams improves performance in comparison with static scheduling when two teams are used for execution. This is because the workloads across the thread teams are more balanced.

5.2.2 Results using four thread teams

The results using four teams are presented as follows. Since the search space of workload vectors is large in the case of four teams, a heuristic way is used to find the best workload vector in each phase. The best workload vector in each phase is summarized in Table 4, for both movies. Figs. 9 and 10 show the execution times of phases for the first movie and second movie, respectively.

Fig. 11 shows the total execution time of each scenario for both movies when four teams are used for execution. The average execution times of both movies are also shown. In Fig. 11, we can see that MDS outperforms MSS, and MDD outperforms MSD, for both movies. On average, MDS achieves 16.5% performance improvement in comparison with MSS, and MDD achieves 11% performance improvement in comparison with MSD. Thus, dynamic scheduling across thread teams improves performance in comparison with static scheduling when four teams are used for execution.

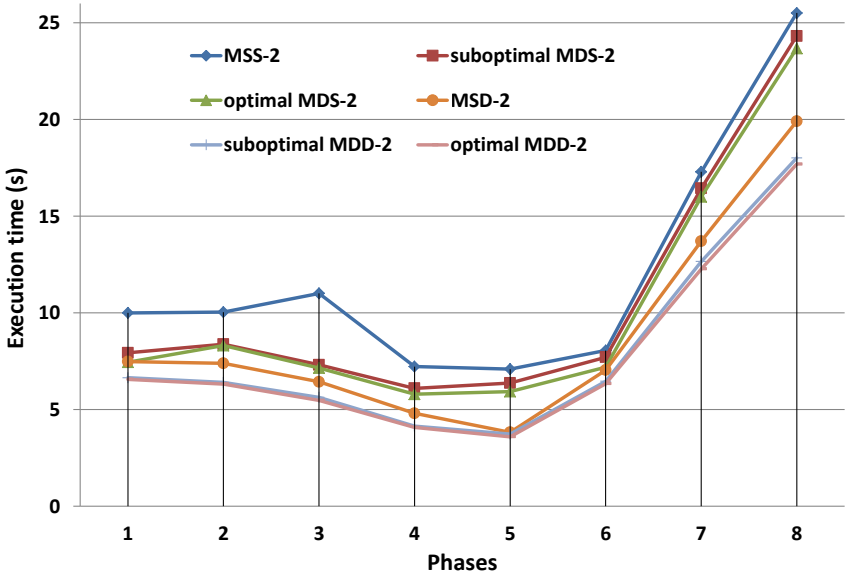

Figure 7: The execution times in each phase of various scenarios for the second movie. Two teams are used for the execution.

In summary, dynamic scheduling within a team is used for balancing the workload within a team, and dynamic scheduling across teams is used for balancing the workload across different teams. Both of them can be used when load imbalance exists both within a team and across different teams. Usually, we cannot know the optimal workload distribution beforehand. That is why static scheduling often encounters load imbalance. With dynamic scheduling, we can dynamically find the best workload distribution across thread teams, though it unavoidably suffers from some scheduling overhead. In Section 5.4, we discuss the inter-team scheduling overhead.

5.3 Normalized performance

The performance model expressed by Eq. (3) assumes that the workload vector is perfectly predicted in each phase. However, in practical use, it is difficult to predict the optimal workload vector in each phase to obtain the optimal performance. If the workload vector decided by the proposed mechanism is not optimal, the performance gain would be less than the optimal one. We discuss the performance gain, assuming imperfect prediction of a workload vector. As an example, suppose that a sub-optimal workload vector in each phase is predicted instead of the optimal one. In such a case, we summarize normalized performances of MDS and MDD in Fig. 12 for both movies, where the normalized performance is given by Eq. (5).

$$Normalized\ performance\ =\ \frac{Optimal\ execution\ time}{Suboptimal\ execution\ time} \times 100\%. \tag{5}$$

The normalized performance indicates how close the suboptimal execution time is to the optimal execution time. In Fig. 12, we can see that the normalized performances are quite high (no less than 95.9%). These results indicate that the performance gain is obtained even if the prediction is not perfect. The performance gain with the sub-optimal workload vector is almost comparable to that with the optimal one. Since we divide the workload ratio by a granularity of 5%, the results also indicate that the granularity is good enough so that we can predict an accurate sub-optimal performance, if not the optimal performance.

In this paper, the inter-team dynamic load balancing mechanism is emulated by using a static one, and thus the obtainable performance improvement heavily depends on how accurate the workload vector is predicted in each phase. In practical use, such a prediction can be performed using

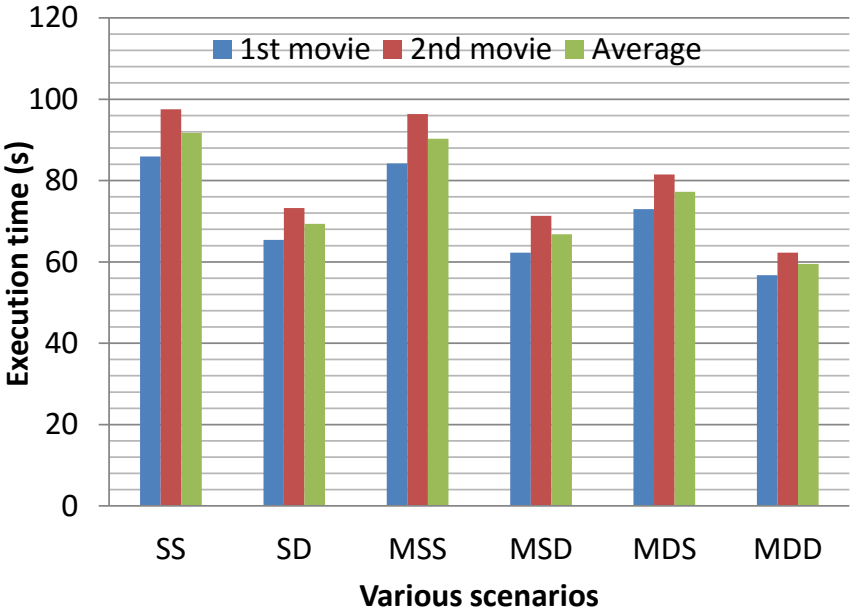

Figure 8: Performance comparison of six scenarios for both movies when two teams are used for execution

Table 5: The maximum overheads in comparison with MDD, for both movies.

	Time (s)	1st movie	2nd movie
2-team execution	Overhead	0.26	0.28
2-team execution	MDD	56.748	62.29777
4-team execution	Overhead	2.18	2.35
4-team execution	MDD	51.1996	57.643

empirical parameter tuning. By executing a given program while changing its workload vector and measuring the performance, we can obtain an appropriate parameter configuration that can achieve high performance. However, the search space of a workload vector significantly increases with more thread teams and more subtle granularity of the workload ratio. Thus, how to prune the search space is a challenge. This will be further discussed in our future work.

5.4 Scheduling overhead

This subsection shows the inter-team scheduling overhead that is estimated based on the assumptions shown in Section 1. The results are shown in Table 5 in comparison with MDD. In the case of two-team execution, the overheads are 0.26 seconds and 0.28 seconds for the first and second movies, respectively. On the other hand, in the case of four-team execution, the overheads are estimated as 2.18 seconds and 2.35 seconds for the first and second movies, respectively. Since the chunk size is chosen as *1*, the overheads in Table 5 can be considered the maximum overheads. In Table 5, we can see that, in the case of two-team execution, the overheads are around 0.46% and 0.45% of the execution times of MDD for the first movie and the second movies, respectively. Thus, the scheduling overheads are considered negligible. In the case of four-team execution, the overheads are around 4.3% and 4.1% of the execution times of MDD for the first movie and the second movies, respectively. With considering the scheduling overhead, MDS outperforms MSS by 13.8% on average of two movies, and MDD outperforms MSD by 7.2% on average of two movies.

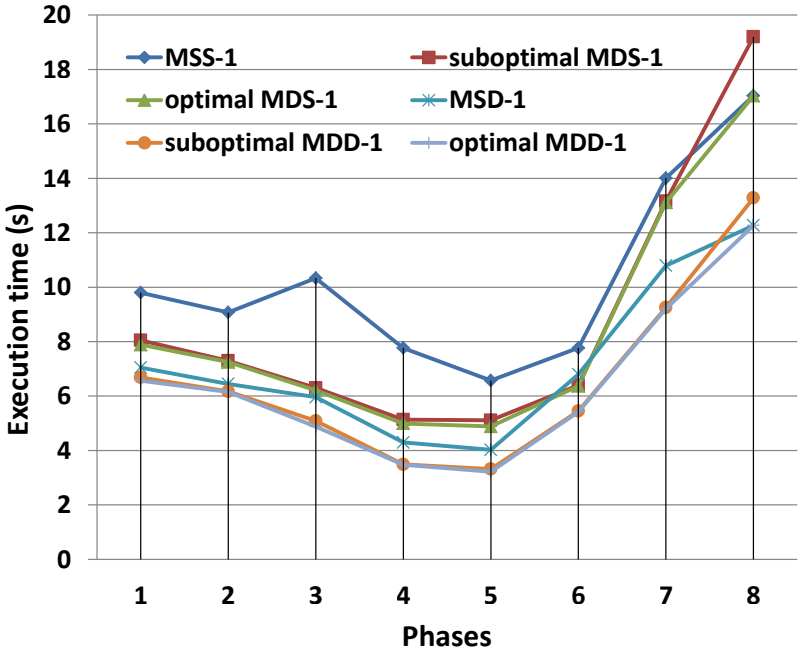

Figure 9: The execution times in each phase of various scenarios for the first movie. Four teams are used for the execution.

6 Conclusions

Massively-parallel many-core processors are gaining more attraction than ever because of their high performance capabilities. Because of the many-core architecture, it is crucial to have balanced workloads on the processor. This work uses OpenMP directives to offload a hotspot to an Intel Xeon Phi coprocessor. Multiple thread teams are created on the coprocessor to execute irregular workloads. A motivating experiment shows that using multiple thread teams can achieve a better performance than using a single one because use of multiple teams reduces the overhead of thread synchronizations compared to a single team.

This paper discusses dynamic scheduling across thread teams that can further accelerate execution in comparison with static one, because the workloads are more balanced. To clarify the benefit of dynamic scheduling across OpenMP thread teams, this work uses an irregular code, the modified ray tracing code, for evaluations. Although this paper only uses the ray tracing code as a target application, other irregular workloads can also benefit from inter-team dynamic load balancing. In this paper, an ideal inter-team dynamic load balancing mechanism is emulated using a static one. Moreover, the scheduling overhead of inter-team dynamic load balancing is estimated based on two simple assumptions. The results show that, in the case of two-team execution, the inter-team dynamic load balancing mechanism can achieve a better performance than a static one with a negligible scheduling overhead. In the case of four-team execution, the overheads are around 4.3% and 4.1% of the execution times of MDD for the first movie and the second movie, respectively. With considering the scheduling overhead, MDS outperforms MSS by 13.8% on average, and MDD outperforms MSD by 7.2% on average. Those results demonstrate that dynamic load balancing across OpenMP thread teams is a promising way to achieve a high performance.

The future work of this paper includes empirically tuning workload vectors to predict the best one. Since the search space of a workload vectors increases dramatically with the number of thread teams and the granularity of workload ratios, pruning the search space is a challenge.

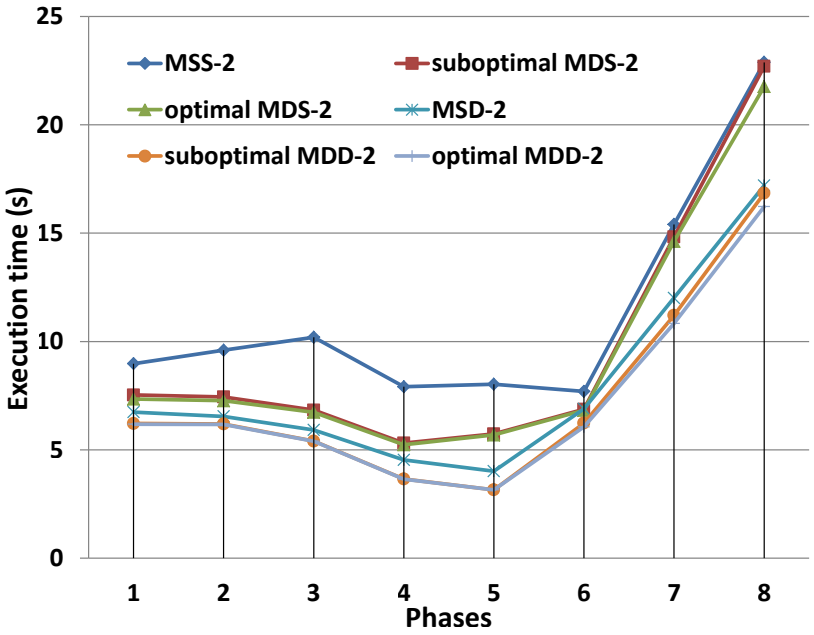

Figure 10: The execution times in each phase of various scenarios for the second movie. Four teams are used for the execution.

Acknowledgments

This research is partially supported by JST CREST "An Evolutionary Approach to Construction of a Software Development Environment for Massively-Parallel Heterogeneous Systems," and Grant-in-Aid for Scientific Research(B) 16H02822. The authors also would like to thank professor Ryusuke Egawa and professor Kazuhiko Komatsu in Tohoku University for their insightful comments on this work.

References

[1] Umut A. Acar, Arthur Charguéraud, and Mike Rainey. Scheduling parallel programs by work stealing with private deques. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP'13)*, pages 219–228, 2013.

[2] Kevin Beason. Smallpt: global illumination in 99 lines of c++. *http://www.kevinbeason.com /smallpt*.

[3] R. Berrendorf and G. Nieken. Performance characteristics for openmp constructs on different parallel computer architectures. *Concurrency: Practice and Experience*, 12(12):1261–1273, 2000.

[4] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46(5):720–748, 1999.

[5] OpenMP Architecture Review Board. The openmp application program interface. *Version 4.5*, pages 1–359, Nov. 2015.

[6] Michael Boyer, Kevin Skadron, Shuai Che, and Nuwan Jayasena. Load balancing in a changing world: Dealing with heterogeneity and performance variability. In *Proceedings of the ACM International Conference on Computing Frontiers (CF'13)*, pages 1–10, 2013.

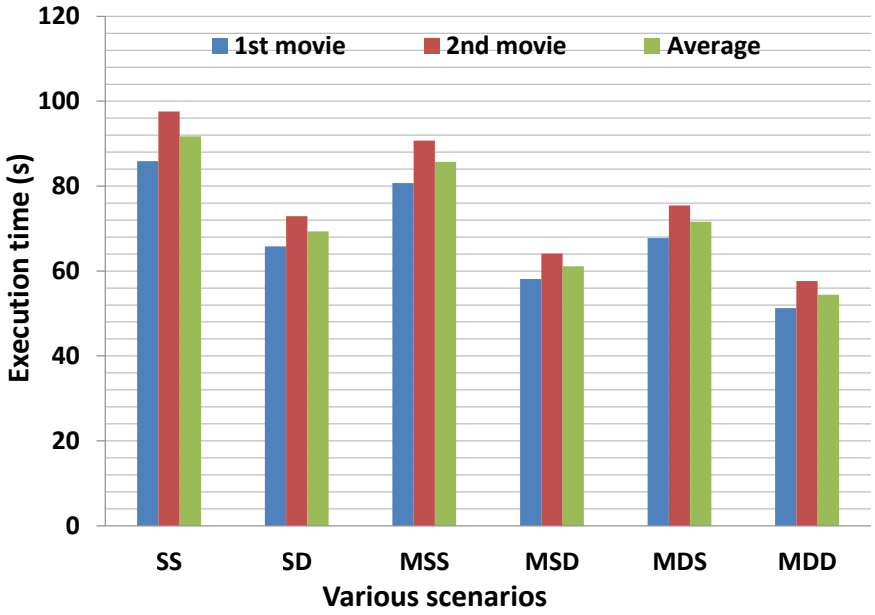

Figure 11: Performance comparison of six scenarios for both movies when four teams are used for execution

[7] Franois Broquedis, Olivier Aumage, Brice Goglin, Samuel Thibault, Pierre-Andr Wacrenier, and Raymond Namyst. Structuring the execution of openmp applications for multicore architectures. In *Proceedings of the 2010 IEEE International Symposium on Parallel and Distributed Processing (IPDPS '10)*, pages 1–10, 2010.

[8] J. M. Bull. Measuring synchronization and scheduling overheads in openmp. In *Proceedings of First European Workshop on OpenMP*, pages 99–105, 1999.

[9] J. M. Bull and D. ONeill. A microbenchmark suite for openmp 2.0. *ACM SIGARCH Computer Architecture News*, 29(5):41–48, 2001.

[10] Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 137–145, 1984.

[11] The Intel Corporation. Intel xeon phi coprocessor instruction set architecture reference manual. pages 1–725, Sep. 2012.

[12] Robert D.Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, 1996.

[13] Alejandro Duran, Julita Corbalan, and Eduard Ayguade. Evaluation of openmp task scheduling strategies. In *Proceedings of the 4th international Workshop on OpenMP*, pages 100–110, 2008.

[14] Marie Durand, Francois Broquedis, Thierry Gautier, and Bruno Raffin. An efficient openmp loop scheduler for irregular applications on large-scale numa machines. In *Proceedings of International Workshop on OpenMP (IWOMP)*, pages 141–155, 2013.

[15] Nathan R. Fredrickson and and Ying Qian Nathan R. Fredrickson. Performance characteristics of openmp constructs, and application benchmarks on a large symmetric multiprocessor. In *Proceedings of the 17th annual international conference on supercomputing (ICS '03)*, pages 140–149, 2003.

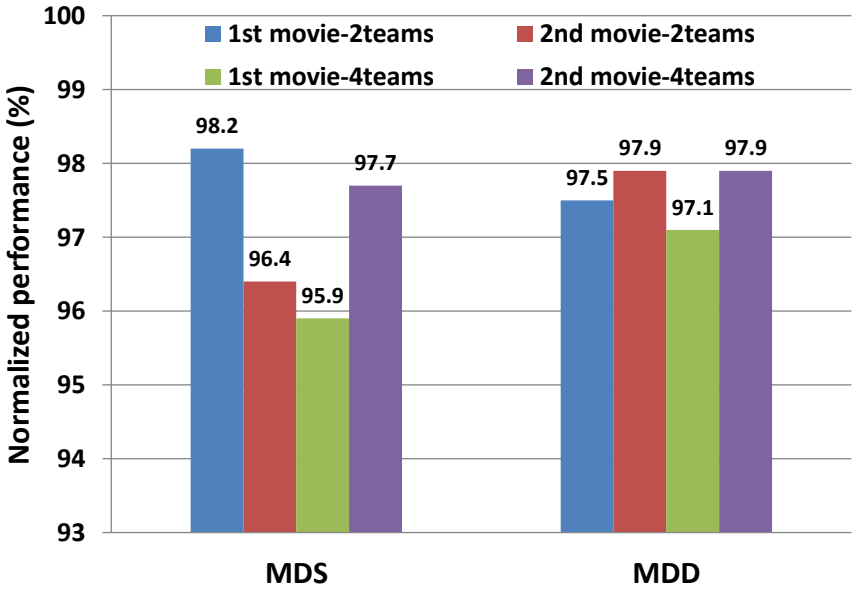

Figure 12: Normalized performances of MDS and MDD

[16] Ralf Hoffmann, Matthias Korch, and Thomas Rauber. Using hardware operations to reduce the synchronization overhead of task pools. In *Proceedings of the 2004 International Conference on Parallel Processing*, pages 241–249, 2004.

[17] C. Luk, S. Hong, and H. Kim. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 45–55, 2009.

[18] Merlyn Melita Mathias and Manjunath Kotari. An approach for adaptive load balancing using centralized load scheduling in distributed systems. *International Journal of Innovative Research in Computer and Communication Engineering*, 3(5):118–125, 2015.

[19] Changwoo Min and Young Ik Eom. Danbi: Dynamic scheduling of irregular stream programs for many-core systems. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, pages 189–200, 2013.

[20] A. Nere, A. Hashmi, and M. Lipasti. Profiling heterogeneous multi-gpu systems to accelerate cortically inspired learning algorithms. In *Proceedings of 2011 IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 906–920, 2011.

[21] OpenACC-standard.org. The openacc application programming interface. *Version 2.5*, pages 1–118, Oct. 2015.

[22] Hrushit Parikh, Vinit Deodhar, Ada Gavrilovska, and Santosh Pande. Efficient distributed work stealing via matchmaking. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–2, 2016.

[23] Max Plauth, Frank Feinbube, Frank Schlegel, and Andreas Polze. Using dynamic parallelism for fine-grained, irregular workloads: a case study of the n-queens problem. In *Proceedings of the Third International Symposium on Computing and Networking*, pages 404–407, 2015.

[24] A. Prabhakar, V. Getov, and B. M. Chapman. Performance comparisons of basic openmp constructs. In *Proceedings of the fourth International Symposium on High Performance Computing (ISHPC)*, pages 413–424, 2002.

[25] Aleksandar Prokopec and Martin Odersky. Near optimal work-stealing tree scheduler for highly irregular data-parallel workloads. In *Proceedings of the 26th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2013)*, pages 55–86, 2013.

[26] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics*, 21(3):703–712, 2002.

[27] Thomas R. W. Scogland, Barry Rountree, Wu chun Feng, and Bronis R. de Supinski. Heterogeneous task scheduling for accelerated openmp. In *Proceedings of 26th International Parallel and Distributed Processing Symposium*, pages 144–155, 2012.

[28] C. Y. Shei, P. Ratnalikar, and A. Chauhan. Automating gpu computing in matlab. In *Proceedings of International Conference on Supercomputing (ICS '11)*, pages 245–254, 2011.

[29] G. Wang and X. Ren. Power-efficient work distribution method for cpu-gpu heterogeneous system. In *Proceedings of International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, pages 122–129, 2010.

[30] Zhenning Wang, Long Zheng, Quan Chen, and Minyi Guo. Cpu+gpu scheduling with asymptotic profiling. *Journal of parallel computing*, 40(2):107–115, 2014.

[31] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. Optimization of sparse matrixvector multiplication on emerging multicore platforms. *Journal of parallel computing*, 35(3):178–194, 2009.

[32] Chuanfu Xu, Lilun Zhang, Xiaogang Deng, Jianbin Fang, Guangxue Wang, Wei Cao, Yonggang Che, Yongxian Wang, and Wei Liu. Balancing cpu-gpu collaborative high-order cfd simulations on the tianhe-1a supercomputer. In *Proceedings of the 28th International Parallel and Distributed Processing Symposium*, pages 725–734, 2013.