The Importance of Dynamic Load Balancing among OpenMP Thread Teams for Irregular Workloads

Xiong Xiao*, Shoichi Hirasawa*, Hiroyuki Takizawa*, and Hiroaki Kobayashi*
*Tohoku University, Sendai, Miyagi, 980-8579 Japan
Email: {xiaoxiong, hirasawa}@sc.cc.tohoku.ac.jp
{takizawa, koba}@tohoku.ac.jp

***Abstract*—Recently, massively-parallel many-core processors such as Intel Xeon Phi coprocessors have attracted researchers' attentions because various applications are significantly accelerated with those processors. In the field of high-performance computing, OpenMP is a standard programming model commonly used to parallelize a kernel loop for many-core processors. For hierarchical parallel processing, OpenMP version 4.0 or later allows programmers to group threads into multiple thread teams. In this paper, we first show the performance gain of using multiple thread teams even for one many-core processor. Then, we demonstrate that dynamic load balancing among those thread teams has a potential of significantly improving the performance of irregular workloads on a many-core processor. Although the current OpenMP specification does not offer such a dynamic load balancing mechanism, we discuss possible benefits of dynamic load balancing among thread teams through experiments using the Intel Xeon Phi coprocessor.**

I. Introduction

Recent years, accelerators and coprocessors such as Intel Xeon Phi have gained more and more attraction, especially in the field of high performance computing (HPC), because they can accelerate HPC applications by up to several orders of magnitude speedup compared to general-purpose processors, i.e., central processing units (CPUs). The significant performance improvement is due to the massive parallel architecture of the coprocessors, which allows a computation code to be executed in a highly-parallel fashion.

Thanks to the advantage of coprocessors, platforms equipped with both CPUs and coprocessors, so-called heterogeneous platforms, have been widely used to accelerate the execution of various applications especially in the HPC field. This paper focuses on Intel Xeon Phi coprocessors. In a heterogeneous platform, Xeon Phi is used as an accelerator to speedup the execution of data-parallel workloads. On the other hand, a CPU handles non data-parallel part, such as serial code execution or data movement between the CPU and Xeon Phi coprocessor.

Although heterogeneous computing brings significant performance improvement, it requires more complicated programming. Fortunately, there exist several high-level programming models (e.g., OpenMP [1] and OpenACC [2]) for parallel programming of heterogeneous systems. OpenMP is a well-known standard for parallel programming. It recently supports device constructs for programming heterogeneous systems. Moreover, it supports multiple thread teams on the target device. Multiple threads can be grouped into a team, and multiple teams can be grouped into a league. The main purpose of introducing thread league and team is to provide a hierarchical execution model for exploring nested parallelism (e.g., nested tasks) in OpenMP programming. A thread team can be newly created, and it works completely independently from other spawning threads and their teams. In this work, the OpenMP device constructs are used for offloading parallel executions on Xeon Phi coprocessors.

Load balancing is an important issue for data-parallel execution to achieve 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 focusing on optimizing the workload balance of the hotspots so as to improve the overall performance of an application [3][4][5]. 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 offloaded to one Xeon Phi coprocessor for execution using OpenMP directives. 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. Here, we consider the irregular loop nests, in which the execution time of each iteration significantly varies. Under such a circumstance, we need to carefully manage the workloads across multiple thread

2379-1896/16 $31.00 © 2016 IEEE
DOI 10.1109/CANDAR.2016.48

```
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 */
            }
          }
        }
    }
}
```

Fig. 1. Modified code segment of the ray tracing loop nest

teams as well as within a thread team, because load imbalance can easily occur for irregular loop nests.

The latest OpenMP specification (version 4.5) supports both static and dynamic loop scheduling mechanisms within a thread team, and also supports a static scheduling mechanism across multiple thread teams. However, there is no dynamic workload balancing mechanism *across different thread teams*. Therefore, this work focuses on dynamically balancing the workloads across multiple thread teams created by OpenMP directives on a Xeon Phi coprocessor. Theoretically, the optimal workload distribution should be achieved when multiple thread teams complete their computations at the same time.

The purpose of this work is to clarify the importance of dynamic load balancing among thread teams introduced to the OpenMP programming model. In this paper, we use an irregular workload code as a target application. Firstly, a motivating example shows that using multiple thread teams could be important to achieve a better performance in the case of executing irregular workloads. After that, we use a static load balancing method to emulate dynamic load balancing across thread teams. At last, we evaluate the potential performance gain of dynamic load balancing.

The contributions of this paper are twofold: Firstly, it clarifies the advantage of execution using multiple thread teams. For irregular applications, multiple thread teams outperform a single team. Secondly, it clarifies the importance 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 there is no related work to address the load imbalance issue across thread teams for irregular workloads. In this work, therefore, a dynamic load balancing mechanism is emulated by using a

TABLE I
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

static load balancing mechanism in order to discuss the benefit of dynamic load balancing at execution of irregular workloads.

The rest of this paper is organized as follows. Section II presents the motivation of this work. Section III describes the related work. Section IV discusses our approach to evaluation of dynamic load balancing across thread teams. Section V shows the evaluation results. Section VI gives concluding remarks and future work.

II. MOTIVATION

In this section, a motivating experiment is performed using ray tracing code with minor modifications. The ray tracing algorithm [9][10] is an important technique to generate an image in computer graphics. It renders an image by tracing the path of light when it encounters objects. Fig. 1 shows an example of the hotspot of the ray tracing algorithm, which is based on the code available at [11], and modified to render multiple images by changing the camera position and number of samples per pixel. 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 number of samples per pixel in each image generation, the execution time of each phase is changed. Furthermore, the execution time of each iteration of the innermost loop varies drastically. Thus, the ray tracing loop nest is an irregular one.

The experiment is performed to show 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 hierarchical execution. However, in this experiment, our goal is not to provide such a hierarchical execution for the ray tracing code. We intend to investigate the performance changes according to the number of thread teams. Therefore, in the experiment, we change the number of thread teams, and measure the execution times for different image sizes. The number of thread teams and the thread count in each team can be specified using OpenMP directives. The scheduling method within a team can also be designated. The number of teams ranges 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 static scheduling mechanism across teams, are used in this experiment. The experimental environment is summarized in Table I.

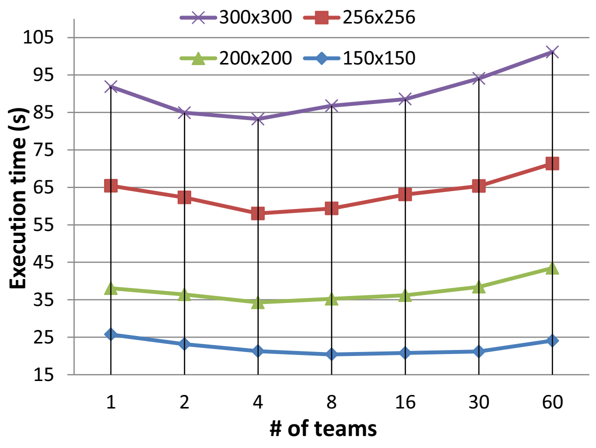

Fig. 2. Performances with different numbers of thread teams across various image sizes

The evaluation results are shown in Fig. 2. Different curves represent different image sizes. The horizontal axis represents the number of thread teams, and the vertical axis represents the execution time in seconds. In Fig. 2, we can see that the different number of teams leads to different performances for a specific image size. Furthermore, using multiple teams is better than using 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 it hurts the parallelism within a team. Therefore, from the above motivating experiment, we can conclude that using multiple teams is potentially better than using a single one for irregular and long loop nests, even though the scheduling mechanism across teams is static.

The above motivating experiment clearly demonstrates that using multiple thread teams is often a wiser choice than using a single one to execute irregular workloads. If multiple thread teams are used to execute an irregular workload, we require that the workloads among the thread teams can be 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 importance of a dynamic load balancing mechanism among thread teams.

III. Related Work

The OpenMP specification [1] provides a target construct to support offloading workloads to a target device for acceleration. The *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 the *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, this work aims to examine the importance of dynamic scheduling across thread teams.

Many researches focus on dynamic load balancing for heterogeneous systems. Wang et al. [12] have proposed a asymptotic profiling method to schedule loop iterations between a CPU and a GPU. Ren et al. [14] have proposed a method to test various work distributions among a CPU and a GPU to find the most efficient distribution. Boyer et al. [15] have presented a dynamic load balancing mechanism that requires no offline training and responds automatically to performance variability. Other approaches, such as in [6][7][8], use modest amounts of initial training to select the workload distribution. Scogland et al. [17] 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. Our work focuses on scheduling the workload of threads on a single device, i.e., the Intel Xeon Phi.

Some related studies use dynamic scheduling for irregular applications. Durand et al. [16] 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. [25] 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 work 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 [12][24] for dynamic load balancing, work stealing [18][19][20][21] is a popular scheduling method. It is proved that work stealing is efficient for scheduling some multi-threaded executions. Prokopec et al. [22] 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 [23] to linear algebra [5].

TABLE II
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 a team
MSD	Multiple teams, static across teams, dynamic within a team
MDS	Multiple teams, dynamic across teams, static within a team
MDD	Multiple teams, dynamic across teams, dynamic within a team

```
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
```

Fig. 3. Statically divide the iteration space of loop y between teams 0 and 1

IV. METHODOLOGY

In this section, we present our methodology to emulate a dynamic load balancing mechanism across thread teams. For simplicity, we only use two thread teams (teams 0 and 1) for the emulation in this work.

Since OpenMP does not support dynamic scheduling across thread teams, we use static load balancing to mimic dynamic one. In this work, we assume that a given application is executed through multiple phases (e.g., N phases, from phase 1 to phase N). A ratio between workloads of the two teams is referred to as a *workload ratio*. For each phase, a workload ratio should be adjusted so that the execution time of every team becomes the same. However, there is no OpenMP mechanism for dynamically adjusting the ratio. Thus, with a fixed workload ratio, we execute the application from beginning to end while measuring the execution time of each phrase. The execution is repeated by changing the workload ratio. As a result, we can obtain the best workload ratio 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 ratio for each phase without any runtime overhead.

For example, if Bi ($i = 1,N$) represents the best workload ratio in phase i, and $Ti(R)$ represents the execution time in phase i when the workload ratio is R, the execution time D of the application using dynamic load balancing can be expressed using Eq. (1).

$$D = \sum_{i=1}^{N} T_i(B_i). \tag{1}$$

Although we have not implemented a dynamic load balancing mechanism for the OpenMP programming model, we can use this performance model to estimate the benefit of introducing such a mechanism. Since the performance model ignores the runtime overhead of dynamic load balancing, it can be considered the estimation of the maximum performance gain by dynamic load balancing. In the preliminary evaluation shown in Section II, the overhead time for dynamic load balancing within a team is negligible. Therefore, we expect that the runtime overhead of dynamic load balancing among teams could be negligible if all the teams are executed on the same device and the application has a sufficiently high computational cost.

V. EVALUATIONS

A. Experimental setup

We use the ray tracing code with minor modifications as the target irregular code shown in Fig. 1. The system used for the evaluation is the same as that in Section II.

TABLE III
THE EXECUTION TIMES WITH DIFFERENT WORKLOAD RATIOS IN EACH PHASE FOR MDS-1.

Time (s)	5%	10%	15%	20%	25%	30%	35%	40%	45%	50%
Phase 1	13.280806	13.276127	12.73217	12.563972	12.305296	12.026693	11.813318	11.275279	10.93588	10.5777
Phase 2	12.166715	12.04817	11.582507	11.426813	11.368547	11.120843	10.843461	10.40929	10.13178	9.69018
Phase 3	14.160308	13.61349	13.53108	13.402047	13.285802	12.961784	12.600458	12.060115	11.375382	10.675707
Phase 4	10.659454	9.573814	8.881163	8.336634	8.051762	7.851624	7.672714	7.689797	7.431081	7.07209
Phase 5	9.818538	8.401837	7.412761	6.285788	**5.271441**	5.555075	5.683935	5.807441	5.95058	5.983956
Phase 6	10.576809	9.480841	8.770037	7.650816	6.899129	**6.454469**	6.779017	7.1607	7.192365	7.36421
Phase 7	24.28761	21.167717	18.902077	16.499147	15.469734	14.481875	**13.486151**	13.503042	13.793523	14.054533
Phase 8	35.261349	32.696686	31.591734	29.844201	27.852688	26.013851	23.894589	21.600173	19.257043	**18.883559**
Time (s)	55%	60%	65%	70%	75%	80%	85%	90%	95%	
Phase 1	10.552273	9.225975	8.255094	**8.222541**	8.65266	9.771817	9.601594	10.090998	10.598833	
Phase 2	8.860193	**8.199121**	8.229264	8.374636	8.45572	8.709244	9.188312	8.786659	9.145615	
Phase 3	9.823923	8.636538	7.588353	**7.049935**	7.073247	7.427765	7.849493	8.125255	8.535403	
Phase 4	6.524818	6.036596	5.621272	**5.396995**	5.732348	5.941514	6.090934	6.430136	6.733037	
Phase 5	6.027773	6.061929	6.071176	6.101066	6.108246	6.190922	6.340552	6.455874	.642985	
Phase 6	7.565831	7.7765	7.972318	8.084793	8.17746	8.213264	8.364628	8.419755	8.441412	
Phase 7	14.484905	14.981089	15.400756	15.503877	15.791718	16.075885	16.287438	16.487791	16.660456	
Phase 8	19.530221	20.238351	21.094643	21.545711	21.946619	22.323836	22.726486	23.09887	23.468081	

TABLE IV
THE EXECUTION TIMES WITH DIFFERENT WORKLOAD RATIOS IN EACH PHASE FOR MDD-1.

Time (s)	5%	10%	15%	20%	25%	30%	35%	40%	45%	50%
Phase 1	10.553705	10.412003	10.125611	10.119266	9.598903	9.581331	9.164175	8.991348	8.458816	7.889396
Phase 2	9.977802	9.922296	9.226597	8.992288	8.80091	8.561445	8.271264	8.144116	7.793046	7.295615
Phase 3	8.121348	8.022866	7.698879	7.617929	7.398923	7.255732	7.017627	6.889794	6.439526	6.387736
Phase 4	6.179109	5.846962	5.704801	5.52535	5.116103	4.974801	4.827154	4.801508	4.534373	4.529525
Phase 5	6.082747	5.672623	5.214004	4.748666	4.398718	3.826143	**3.802912**	3.941952	4.006342	4.008636
Phase 6	9.004516	8.193077	7.525893	6.931282	6.295811	**5.908213**	6.244672	6.426979	6.44206	6.62078
Phase 7	14.887347	13.639588	12.418831	11.741001	11.278173	10.810584	10.51355	**10.284674**	10.77363	11.046884
Phase 8	20.947242	20.128935	19.298938	18.219718	17.85101	16.706297	16.276723	15.276867	14.68112	**14.243807**
Time (s)	55%	60%	65%	70%	75%	80%	85%	90%	95%	
Phase 1	7.558965	7.238691	**6.803903**	7.068571	7.35505	7.48452	7.923447	7.989285	8.137119	
Phase 2	6.978577	**6.450186**	6.473972	6.483469	6.795185	6.890419	7.462948	7.657384	7.713141	
Phase 3	6.130907	5.780993	**5.261207**	5.460973	5.661908	5.637189	5.997791	6.184556	6.380378	
Phase 4	4.18347	4.168981	**3.993115**	4.065152	4.018265	4.166625	4.230215	4.311951	4.548755	
Phase 5	4.098504	4.124763	4.193593	4.106581	4.213115	4.218037	4.343264	4.401429	4.502899	
Phase 6	6.69899	6.890287	6.926821	7.137034	7.135283	7.275867	7.380914	7.427375	7.444481	
Phase 7	11.318389	11.692421	11.972692	12.385042	12.675218	13.068278	13.02636	13.60262	13.853945	
Phase 8	14.554736	15.135219	15.954952	16.398861	16.991214	17.366768	17.987621	18.634717	19.078448	

B. Evaluation results and discussions

We compare the performances of six different scenarios shown in Table II 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 two-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 two-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 two-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 two-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 simple way. However, MDS and MDD are not supported yet. Thus, we use the proposed methodology described in Section IV 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 256x256 in the evaluation. By changing the camera position and direction from phase to phase, we can create movies. We create two

TABLE V
THE EXECUTION TIMES WITH DIFFERENT WORKLOAD RATIOS IN EACH PHASE FOR MDS-2.

Time (s)	5%	10%	15%	20%	25%	30%	35%	40%	45%	50%
Phase 1	12.568251	12.503689	12.345279	12.101752	11.817496	11.67091	11.318349	10.799447	10.454772	9.995805
Phase 2	11.923979	11.884358	11.832429	11.671529	11.669452	11.384989	11.134708	10.648506	10.454318	10.034646
Phase 3	13.730648	13.593288	13.637524	13.541581	13.384242	13.630735	12.928764	12.557689	11.823203	11.005315
Phase 4	10.680989	9.70435	9.065778	8.498481	8.178043	7.864438	7.990988	7.67667	7.626623	7.222382
Phase 5	10.764124	8.817929	7.173078	**5.928271**	6.373182	6.687222	6.767119	6.728096	7.032659	7.089417
Phase 6	11.105508	10.272589	9.593942	8.991259	8.265337	7.715049	**7.189344**	7.702953	7.935753	8.049712
Phase 7	27.664651	25.319088	23.089388	20.859586	18.811462	17.05047	**16.003882**	16.438396	16.925518	17.293338
Phase 8	40.918882	37.597454	34.407975	31.337218	28.308926	25.577919	**23.679942**	24.318072	24.890666	25.506687
Time (s)	55%	60%	65%	70%	75%	80%	85%	90%	95%	
Phase 1	9.059892	8.250007	**7.443739**	7.931726	8.372347	8.47357	9.037018	9.326459	9.526265	
Phase 2	9.13065	8.378482	**8.310164**	8.53081	9.024043	9.031304	9.339059	9.325031	9.856097	
Phase 3	10.215126	9.266835	7.977319	**7.145103**	7.307716	7.526578	7.812443	8.268071	8.578049	
Phase 4	6.657696	6.096725	**5.787701**	6.158076	6.190673	6.368455	6.535518	6.730301	7.132804	
Phase 5	7.152992	7.175816	7.171006	7.216952	7.204794	7.292685	7.360573	7.424494	7.597668	
Phase 6	8.246579	8.417149	9.450384	8.74927	8.888515	8.981056	9.008226	9.06839	9.074053	
Phase 7	17.722599	17.994536	18.404058	18.656395	18.718047	18.933878	19.081286	19.126677	19.265851	
Phase 8	26.068711	26.785054	27.175181	27.35619	27.619708	27.886688	28.100547	28.203676	28.404683	

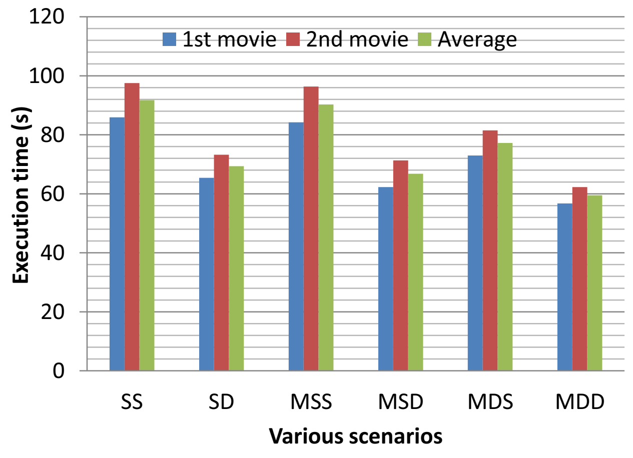

Fig. 4. Performance comparison using six scenarios for both movies

movies by moving the camera in two different ways. Regarding the first movie, the camera position and direction in the i-th phase are (30+20*i, 42+10*i, 290-5*i) and (0, -0.0536, -1), respectively. On the other hand, the camera position and direction of the second movie are different from that of the first one. 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 two parts. One part is executed on team 0, and the other part is executed on team 1, as shown in Fig. 3. By varying the workload ratio between teams 0 and 1, we measure the execution times with each ratio. The workload ratios of team 0 range from 5% to 95%. Then, we find the best workload ratio 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.

Tables III and IV show the execution times with different workload ratios in each phase for MDS-1 and MDD-1, respectively. On the other hand, Tables V shows the execution times for MDS-2. The detailed execution times for MDD-2 cannot be shown due to page limitation. In the tables, the best execution time of each phase is marked red and bold. From the tables, we can see that the best workload ratios in each phase are different from 50%, except phase 8 for MDS-1. It means that workload imbalance exists in almost all the phases.

Fig. 4 shows the total execution time of each scenario for both movies. The average execution times of both movies are also shown. In Fig. 4, we can see that dynamic scheduling within a thread team is always better than static one. This is because dynamic scheduling improves the load balance within a thread team. Moreover, MDS outperforms MSS, and MDD outperforms MSD, for both movies. In average, MDS achieves 14.5% performance improvement in comparison with MSS, and MDD achieves 10.9% performance improvement in comparison with MSD. Thus, dynamic scheduling across thread teams improves performance in comparison with static one because the workloads across thread teams are more balanced. 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. Therefore, in our future work, we will implement a scheduling mechanism for dynamic load balancing among thread teams, and discuss the actual performance gain with considering the scheduling overhead.

VI. Conclusions and Future Work

Massively-parallel many-core coprocessors 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 coprocessor. 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. The motivating experiment shows that using multiple thread teams can achieve a better performance than using a single one.

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 importance of dynamic scheduling across OpenMP thread teams, this work uses an irregular code, the modified ray tracing code, for evaluations. All the results show that dynamic scheduling across thread teams can achieve a better performance than static one.

In this paper, the scheduling overhead within a thread team is included in the evaluation results. Nevertheless, the scheduling overhead across teams is excluded, since we use static scheduling mechanism to mimic dynamic one. Therefore, the future work of this paper includes an efficient dynamic scheduling method to balance the workloads across different thread teams, with considering actual overhead. Moreover, since this paper only uses two thread teams to execute the irregular codes, we will consider a larger number of thread teams in the future work.

References

[1] The OpenMP Application Program Interface. Version 4.5, November 2015. http://www.openmp.org/mp-documents/OpenMP-4.5.pdf.

[2] The OpenACC Application Programming Interface. Version 2.5. October 2015. http://www.openacc.org/sites/default/files/OpenACC_2pt5.pdf.

[3] Franois Broquedis, Olivier Aumage, Brice Goglin, Samuel Thibault, Pierre-Andr Wacrenier, Raymond Namyst. Structuring the execution of OpenMP applications for multicore architectures. 2010 IEEE International Symposium on Parallel and Distributed Processing (IPDPS). Pages 1-10. April 2010.

[4] 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. 2015 Third International Symposium on Computing and Networking. Pages 404-407. 2015.

[5] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, James Demmel. Optimization of sparse matrixvector multiplication on emerging multicore platforms. Journal of parallel computing. Volume 35, Issue 3, March 2009. Pages 178-194. 2009.

[6] C. Luk, S. Hong, and H. Kim. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In International Symposium on Microarchitecture (MICRO), Dec. 2009.

[7] C.Y. Shei, P. Ratnalikar, and A. Chauhan. Automating GPU computing in MATLAB. In International Conference on Supercomputing (ICS), May 2011.

[8] A. Nere, A. Hashmi, and M. Lipasti. Profiling heterogeneous multi-GPU systems to accelerate cortically inspired learning algorithms. In International Symposium on Parallel and Distributed Processing (IPDPS), May 2011.

[9] Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. Proceedings of the 11th annual conference on Computer graphics and interactive techniques. Pages 137-145. 1984.

[10] Timothy J. Purcell, Ian Buck, William R. Mark, Pat Hanrahan. Ray tracing on programmable graphics hardware. ACM Transactions on Graphics. Pages 703-712. Volume 21 Issue 3, July 2002.

[11] Kevin Beason. Smallpt: Global illumination in 99 lines of C++. http://www.kevinbeason.com/smallpt/.

[12] Zhenning Wang, Long Zheng, Quan Chen, Minyi Guo. CPU+GPU scheduling with asymptotic profiling. Journal of parallel computing. Volume 40, Issue 2, February, 2014. Pages 107-115. 2014.

[13] P.Neelakantan. Decentralized load balancing in heterogeneous systems using diffusion approach. International Journal of Distributed and Parallel Systems (IJDPS) Vol.3, No.1, pages 229-239. January 2012.

[14] G. Wang and X. Ren. Power-efficient work distribution method for CPU-GPU heterogeneous system. In International Symposium on Parallel and Distributed Processing with Applications (ISPA). Pages 122-129. Sept. 2010.

[15] Michael Boyer, Kevin Skadron, Shuai Che, and Nuwan Jayasena. Load Balancing in a Changing World: Dealing with Heterogeneity and Performance Variability. Proceedings of the ACM International Conference on Computing Frontiers (CF'13). Article No. 21. 2013.

[16] Marie Durand, Francois Broquedis, Thierry Gautier, Bruno Raffin. An Efficient OpenMP Loop Scheduler for Irregular Applications on Large-Scale NUMA Machines. International Workshop on OpenMP (IWOMP), Sep 2013.

[17] Thomas R. W. Scogland, Barry Rountree, Wu-chun Feng, and Bronis R. de Supinski. Heterogeneous Task Scheduling for Accelerated OpenMP. Proceedings of 26th International Parallel and Distributed Processing Symposium. Pages 144-155. 2012.

[18] Robert D.Blumofe; Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, Yuli Zhou. Cilk: An efficient multi-threaded runtime system. Journal of Parallel and Distributed Computing. Volume 37, Issue 1. Pages 5569. 1996.

[19] Robert D. Blumofe, Charles E. Leiserson. Scheduling multithreaded computations by work stealing. Journal of the ACM (JACM). Volume 46 Issue 5. Pages 720-748. Sep. 1999.

[20] Hrushit Parikh, Vinit Deodhar, Ada Gavrilovska, Santosh Pande. Efficient distributed work stealing via matchmaking. Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. Article No. 37. 2016.

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

[22] Aleksandar Prokopec and Martin Odersky. Near Optimal Work-Stealing Tree Scheduler for Highly Irregular Data-Parallel Workloads. The 26th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2013). Pages 55-86. 2013.

[23] 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. Proceedings of the 28th International Parallel and Distributed Processing Symposium. Pages 725-734. 2014.

[24] 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. Volume 3, special Issue 5. Pages 118-125. 2015.

[25] Changwoo Min and Young Ik Eom. DANBI: Dynamic Scheduling of Irregular Stream Programs for Many-Core Systems. Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques. Pages 189-200. 2013.