TOHOKU UNIVERSITY
Graduate School of Information Sciences

OpenMP Extensions for Irregular Parallel Applications

（不規則な並列アプリケーションのための OpenMP 拡張に関する研究）

A dissertation submitted for the degree of Doctor of Philosophy
(Information Sciences)

Department of Computer and Mathematical Sciences

by

Xiong XIAO

January 15, 2019

OpenMP Extensions for Irregular Parallel Applications

Xiong XIAO

Abstract

Recent years, the node architecture of high performance computing (HPC) systems is becoming more and more complex. On one hand, HPC systems are often coupled with different kinds of processors such as central processing units (CPU) and accelerators (or coprocessors). This kind of system is called a heterogeneous system. Accelerators are often used in the field of HPC because they can accelerate some applications by up to several orders of magnitude, compared to CPUs. On the other hand, more and more cores are integrated into the HPC systems. The many-core architecture of the systems allows a computation code to be executed in a highly-parallel fashion. To efficiently exploit the computing power of a many-core system, parallel programming using many threads is essential. Currently, there exist several parallel programming models. One popular model is OpenMP which is the de-facto standard for shared memory parallel programming. It introduces *thread league* and *thread team* for hierarchical parallelism. A thread league is a league of thread teams, and a thread team is a team of synchronizable threads.

To deal with the heterogeneity of an HPC system, processor selection mechanisms are required to effectively select an appropriate processor to execute a given application. On the other hand, to deal with the many-core nature of an HPC system, load balancing mechanisms are required to keep all the available cores as busy as possible. Therefore, this dissertation focuses on discussing processor selection and load balancing mechanisms.

In the HPC field, the applications are also becoming complex to enable more advanced simulations. Furthermore, the processor selection and load balancing mechanisms are often written in the applications, which further increases the complexity of the applications. In many applications, the iterative computation of a particular loop nest consumes most of the execution time. Such a loop nest is called a *hotspot* of the application. This dissertation focuses on such hotspots because it is expected to improve overall performance of an application by improving the performance of the hotspot. This dissertation targets at *irregular applications*, in which the execution time of each iteration of the hotspot changes drastically. In this dissertation, a hotspot of an application is executed either on a CPU or an accelerator, and the OpenMP *target* constructs are used to offload the hotspot to an accelerator.

The objective of this dissertation is to keep HPC applications separated from pro-

cessor selection and load balancing mechanisms. This dissertation specifically focuses on processor selection and load balancing mechanisms because they are important to deal with heterogeneity and many-core nature of an HPC system. The research approach is to move processor selection and load balancing mechanisms out of the applications, and to integrate those mechanisms into the OpenMP specification. In this way, the applications become easy to maintain in long-term software development because they do not explicitly contain the two complex mechanisms.

This dissertation discusses one processor selection mechanism that is based on compiler directive customization. This dissertation also discusses two load balancing mechanisms that are thread management and workload management. Thread management is to adjust the number of threads, and workload management is to adjust the number of iterations.

First, this dissertation discusses runtime processor selection for heterogeneous systems. Suppose that one application is available for execution, OpenMP directives have already been inserted into the application, and programmers want to avoid as many code modifications as possible. Then, the processor appropriate for the execution depends on the application features. One example of application features is the problem size that usually affects the number of iterations of the hotspot. If the problem size is large, an accelerator should usually be used. Otherwise, CPU execution is usually a better choice. In general, a different version of the given application is suitable for each processor. In the case where the problem size varies, programmers need to manually adjust different problem sizes and maintain various code versions. In order to automatically select the appropriate processor, this dissertation proposes directive customization to generate various code versions, each of which is suitable for a different processor. To customize existing directives and define new ones, an XML format is proposed to express the specifications of the directive, such as directive name and clause name. A code translation framework, Xevolver, as well as a directive parser, is used to transform source code associated with customized directives to various code versions. In this dissertation, a runtime processor selection mechanism based on the problem size is discussed as an example of processor selection. In the case of selecting processors based on other features, different directives would be required. Therefore, this dissertation discusses the necessity of directive customization.

Second, this dissertation discusses the importance of dynamic thread management for OpenMP programming on many-core systems. When an application is executed using many threads, the overhead of synchronization among threads often becomes non-negligible. Usually, the synchronization overhead increases with the number of threads. In this dissertation, multiple thread teams are used for execution to reduce the number of threads participating in synchronization and thereby reduce the synchronization overhead. The current OpenMP specification (version 5.0) allows static adjustment of the number of threads in each thread team, and all thread teams have the same number of threads.

A conventional approach is to assign the maximum number of threads that allows a given system to execute an application in order to maximize the parallelism. However, the conventional approach might be ineffective because synchronization overhead is not taken into account. If the overhead overwhelms the performance gain from using more threads, the performance degrades by using more threads. Thus, there exists an optimal number of threads that leads to the best performance. This dissertation shows that it is better to adjust the number of threads in each team individually for irregular applications. It means that the optimal number of threads in each team can be different. This is because there is load imbalance across thread teams in the case of irregular applications, and one solution to tackle the load imbalance is to use a different number of threads in each team. In this dissertation, a static thread team size adjustment method is used to emulate a dynamic one. The evaluation results show that adjusting the number of threads in each team individually achieves a higher performance than the conventional approach. This shows the importance of dynamic thread management for OpenMP thread teams.

Third, this dissertation discusses the advantages of dynamic workload management among OpenMP thread teams. One more solution to tackle load imbalance across thread teams is workload management, which is to adjust the number of iterations assigned to each thread team. This dissertation uses a static workload management mechanism to mimic a dynamic one, so as to estimate the upper bound of performance gain of dynamic workload management excluding real scheduling overhead. It also estimates the scheduling overhead according to two assumptions. The first one is that inter-team scheduling overhead can be estimated using the intra-team scheduling overhead. The second one is that the inter-team scheduling overhead increases exponentially with the number of thread teams. By considering the estimated scheduling overhead, the evaluation results show that dynamic workload management across thread teams achieves a better performance in comparison with static workload management.

In conclusion, the performances of OpenMP codes are improved without major code modifications if runtime processor selection, dynamic thread management, and dynamic workload management are integrated into future OpenMP specification.

Table of Contents

List of Figures

List of Tables

List of Acronyms

HPC High Performance Computing

CPU Central processing Unit

GPU Graphic Processing Unit

XML eXtensible Markup Language

AST Abstract Syntax Tree

XSLT eXtensible Stylesheet Language Transformations

CCDP Customizable Compiler Directive Parser

KNC Knights Corner

KNL Knights Landing

Chapter 1

Introduction

1.1 Background

The clock frequency of a processor has been tremendously improved over last several decades. Thanks to the improvement, the performance of a processor has been boosted significantly. However, as Sutter mentioned: "The free lunch is over" [1]. Nowadays, increasing the clock frequency is becoming more and more difficult. What is worse, since increasing the clock frequency alone is not sufficient any more to achieve high performance for parallel applications, other approaches must be employed. One promising approach to improve performance is to use a large number of relatively-low-frequency cores in a computing system, so-called a many-core system. Today, many-core processors such as Intel Xeon Phis [2] and NVIDIA Graphic Processing Units (GPUs) [3] are widely used in the field of high performance computing (HPC).

In order to clearly define the terms used in this dissertation, Table 1.1 shows the terminology in this dissertation. A *hotspot* is a loop nest that consumes most of the execution time of an application. This dissertation focuses on the hotspot of a given application because it is expected to improve overall performance of the application by improving the performance of the hotspot. An *irregular application* is an application, in which each iteration of its hotspot consumes a different execution time. An irregular application can have multiple phases. One *phase* is a time period, during which multiple

Table 1.1: The terminology in this dissertation.

Term	Definition
Hotspot	A kernel loop that consumes most of the execution time of an application.
Irregular application	An application in which each iteration of its hotspot consumes a different execution time.
Phase	A time period during which multiple iterations are executed. Each phase has a different performance characteristic.
Thread team size	The number of threads in a thread team.
Load	The amount of computation.
Thread management	Adjustment of the number of threads.
Workload management	Adjustment of the number of iterations.

iterations are executed while performance characteristics are unchanged. *Thread team size* is the number of threads in a thread team, and a *thread team* is a bunch of synchronizable threads. *Load* is the amount of computation. *Thread management* is adjustment of the number of threads, and *workload management* is adjustment of the number of iterations.

Recent years, the node architecture of HPC systems is becoming more and more complex. HPC systems are often coupled with different types of processors such as central processing units (CPU) and many-core accelerators (or coprocessors). Such a system is called a heterogeneous system. In the heterogeneous system, each type of processor has its own performance characteristic, and thus choosing which type of processor to execute a given application is a challenge. Accelerators are often used in the field of HPC because they can accelerate some applications by up to several orders of magnitude, compared to CPUs. One reason of the significant performance improvement is due to the many-core architecture that allows a computation code to be executed in a highly-parallel fashion.

In the HPC field, applications are also becoming more and more complex to enable more advanced numerical simulations. In addition, various mechanisms are often incorporated into a numerical simulation code so that the simulation can exploit the performance of its target system. As a result, the code complexity further increases in comparison with simply programming a numerical computation model.

More specifically, this dissertation targets *irregular applications* since irregular applications can be commonly seen in real-world applications and encounter load imbalance

```
1 for(int d = 0; d < numDomain; ++d){
2   for(int r =0; r < numRegion[d]; ++r){
3     for(int i =0; i < numElem[r]; ++i){
4       /*Loop body*/
5     }
6   }
7 }
```

Figure 1.1: An example of irregular applications.

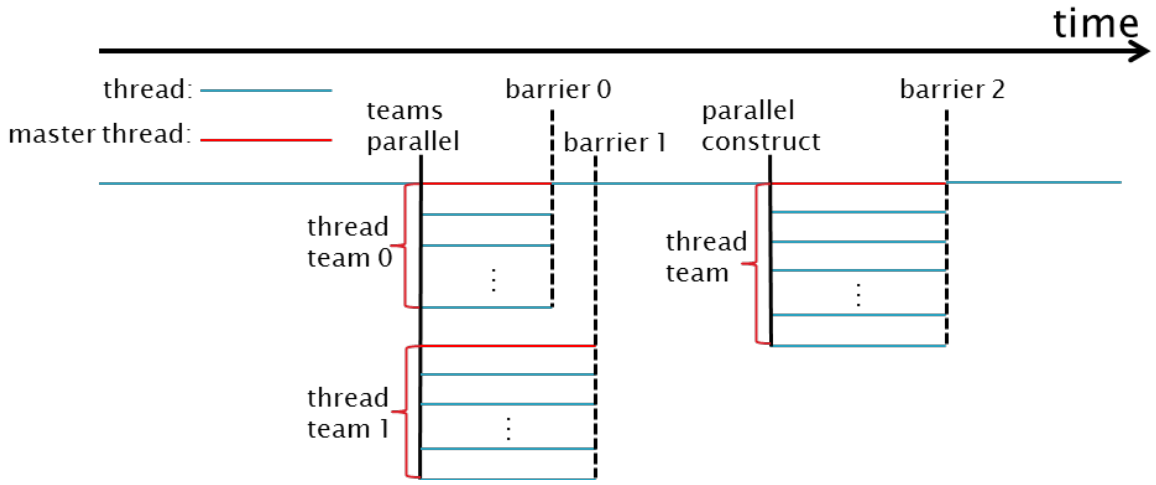


Figure 1.2: Fork-Join execution model of OpenMP.

easily. However, the performances of irregular applications using OpenMP have been rarely studied. Figure 1.1 shows an example of irregular applications. In the example, the lengths of inner loops depend on the loop index variables of their outer loops. Thus, the execution time of each outermost loop iteration can change drastically. In this dissertation, an application consists of multiple phases, and each phase has a different performance characteristic. Therefore, the execution strategy (e.g., the number of threads used for execution) of one phase is potentially different from that of another phase.

Due to the heterogeneity and many-core nature of a recent HPC system, parallel programming is essential to exploit the computing power of such a system. Nowadays, several directive-based parallel programming models, such as OpenMP [5] and OpenACC [7], have been developed. Among the programming models, OpenMP is the de facto standard for shared memory parallel programming.

The most basic execution model of OpenMP is called the *fork-join* model, as illus-

```
#pragma omp target
#pragma omp teams num_teams(4) thread_limit(60)
#pragma omp parallel for
for(i = 0; i < X; i++){
  /*Loop body*/
}
```

Figure 1.3: A code example using *target* and *teams* constructs.

trated in Figure 1.2. When a thread encounters a *teams* construct, it creates master threads, each of which is corresponding to one *thread team*. Each master thread creates a team of threads when a *parallel* construct is encountered (*Fork*). When a thread finishes its work, it waits for the other ones at an implicit barrier at the end of the parallel region. When all threads arrive at the barrier, they join the master thread of the team (*Join*). The OpenMP specification introduces *thread league* and *thread team* for hierarchical parallelism. A thread league is a set of thread teams. OpenMP specification version 4.0 [6] or later allows programmers to use *target* constructs for offloading computations to accelerators. Moreover, it allows programmers to use *teams* and *parallel* constructs for creating multiple thread teams on the accelerators. A *teams* construct must be contained within a *target* construct. One thread team works independently from other spawning threads and their teams. 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. A code example using *target* and *teams* constructs are shown in Figure 1.3.

In this dissertation, the hotspot of an application is executed either on a CPU or an accelerator. OpenMP *target* constructs are used to offload a hotspot to an accelerator. A novel point of this dissertation is to use multiple thread teams to execute an irregular application. The reason of using multiple thread teams is that using multiple thread teams can reduce thread synchronization overhead compared to using a single thread team. In general, the synchronization overhead increases with the number of joining threads [69, 90]. Since the threads in a team need to synchronize while the threads across

teams do not need to synchronize, using multiple thread teams can reduce the number of threads joining the synchronization, and thus reduce the synchronization overhead. Moreover, irregular applications may require frequent thread synchronization during parallel execution. Therefore, it is better to use multiple thread teams rather than a single one to execute irregular applications.

1.2 Research problem and objective

A research problem is that advanced mechanisms are often written in HPC applications, which messes up the applications and makes the applications difficult to maintain. This dissertation focuses on two types of mechanisms. One is a runtime processor selection mechanism and the other one is a load balancing mechanism. Processor selection and load balancing mechanisms are obviously important because a processor selection mechanism can deal with heterogeneity of an HPC system, and a load balancing mechanism can keep all available cores of an HPC system busy. Here, a load balancing mechanism employs either thread management or workload management. Thread management is to adjust the number of threads in a thread team, and workload management is to adjust the number of iterations assigned to each thread team.

In the current OpenMP specification (version 5.0), selecting a processor to execute a hotspot is left to programmers. The programmers might need to severely modify an application while keeping a particular processor configuration in mind so that the application can select an appropriate processor at runtime. Particularly, an irregular application may have several phases, and each phase is suitable for a different processor to be executed. It means that an irregular application may require a different processor for a different phase. This may further complicate the application. As a result, the application itself is messed up, and thus the maintainability of the application would be severely degraded. Accordingly, this dissertation discusses runtime processor selection without major modifications of the original application.

In the case of execution using multiple thread teams, the current OpenMP specification has two notable behaviors. First, by default, a number of running threads are equally divided and assigned to each thread team, and thus each team has an equal team size. The team size must be specified before executing an application and stays constant until the end of the execution. Second, a loop is parallelized by evenly dividing the loop length and assigning the iterations to each thread team. This kind of division is effective when each iteration of the loop has the same execution time. However, this is not the

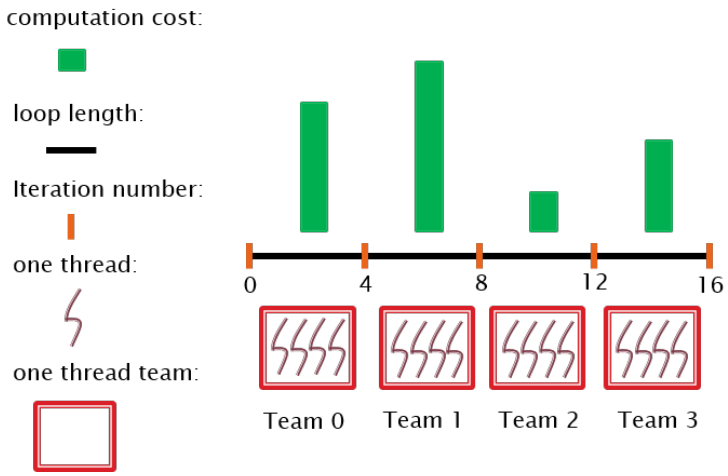
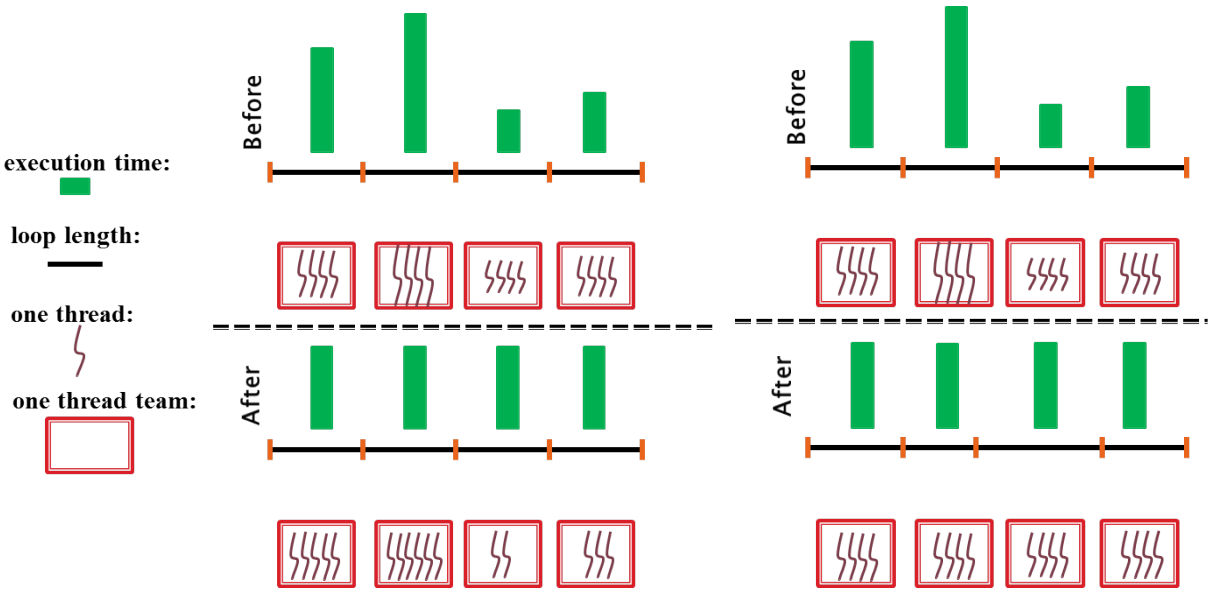


Figure 1.4: The occurrence of load imbalance when executing one phase of an irregular application with multiple thread teams. The numbers represent the iteration number.

case for irregular applications. Since each iteration has a different execution time, load imbalance across thread teams inevitably occurs. Figure 1.4 illustrates how equal division can cause the load imbalance in one phase of an irregular application. In this figure, each thread team has four threads, and the loop length is equally divided and assigned for each team. Let's assume that iterations 4-7 and 8-11 are the ones with the highest and lowest execution times, respectively. Since team sizes are the same across thread teams, load imbalance severely occurs between teams 1 and 2.

In order to deal with the load imbalance across thread teams, this dissertation presents two approaches. The first one is thread management, and the second one is workload management. Each of the two approaches has its own advantages and disadvantages, as illustrated in Figure 1.5. Current OpenMP divides a loop length evenly among thread teams. In the case of irregular applications, such a kind of loop length division leads to a different execution time of each thread team. Thread management aims to balance the execution times of thread teams by adjusting the number of threads in each team. It introduces a small runtime overhead as shown later in this dissertation. However, it cannot finely adjust the the number of iterations assigned to each thread team. In addition, the execution time does not necessarily reduce even if the number of threads increases. Hence, adjusting the number of threads in each thread team may not lead to load balancing among thread teams. On the other hand, workload management aims to balance the execution



(a) Thread management (b) Workload management

Figure 1.5: The differences between thread management and workload management. Thread management adjusts the number of threads in each thread team, while workload management adjusts the number of iterations assigned to each thread team.

times of thread teams by finely adjusting the number of iterations assigned to each team while keeps the thread team size unchanged. However, it may introduce a large runtime overhead. Therefore, if the runtime overhead of workload management is smaller than the performance gain, workload management can be used to tackle load imbalance across thread teams. If the runtime overhead of workload management is so large that it kills performance gain, thread management should be used to tackle the load imbalance.

The processor selection and load balancing mechanisms directly written in an application severely mess up the application, and make the application difficult to maintain. The objective of this dissertation is to separate processor selection and load balancing mechanisms from HPC applications. The research approach is to move processor selection and load balancing mechanisms out of the applications, and to integrate those mechanisms into the OpenMP specification. In this way, the applications become easy to maintain in long-term software development because they do not explicitly contain the two complex mechanisms. Consequently, this dissertation discusses three topics that are processor selection, thread management, and workload management.

First, this dissertation discusses runtime processor selection for heterogeneous systems. Suppose that one application is available for execution, and it can be executed either on CPU or accelerator, using OpenMP directives. Then, the processor appropriate for the execution depends on the application features. One example of application features is the problem size that is strongly correlated with the loop length of a hotspot in many cases. It is assumed that an irregular application may have different code versions, each of which has a different problem size. Generally, each code version may require a different processor. If the problem size is large, an accelerator should usually be used. Otherwise, CPU is usually used. In the case where the problem size varies, programmers need to manually adjust the problem sizes and maintain all code versions. In order to select an appropriate processor at runtime, this dissertation proposes directive customization to generate various code versions, each of which is suitable for a different processor. It is assumed that OpenMP directives have already been inserted into a given application, and thus directive customization can use the existing directives and change the behaviors of the directives. To customize existing directives and define new ones, an XML format is proposed to express the specification of the directive, such as directive name and clause name. A code translation framework, Xevolver [9,10], as well as a directive parser, is used to transform source code associated with customized directives to various code versions. In this dissertation, a runtime processor selection mechanism based on problem size is discussed as an example of processor selection. Therefore, this dissertation discusses the necessity of directive customization for runtime processor selection.

Second, this dissertation clarifies the benefits of dynamic thread management for OpenMP programming on many-core systems. When an application is executed using many threads, the overhead of synchronization among threads often becomes non-negligible. Usually, the synchronization overhead increases with the number of threads. In this dissertation, multiple thread teams are used for executing an irregular application. The current OpenMP specification allows static adjustment of the number of threads in each thread team, and all thread teams have the same number of threads. A conventional approach is to assign the maximum number of threads that allows a given system to

execute an application in order to maximize the parallelism. However, the conventional approach might be ineffective because synchronization overhead is not taken into account. If the overhead overwhelms the performance gain from using more threads, the performance degrades by using more threads. Thus, there exists an optimal number of threads that leads to the best performance. This dissertation shows that it is better to adjust the number of threads in each team individually for irregular applications. It means that the optimal number of threads in each team can be different. This is because there is load imbalance across thread teams in the case of irregular applications, and one solution to tackle the load imbalance is to use a different thread team size. In this dissertation, a static thread team size adjustment method is used to emulate a dynamic one. The evaluation results show that adjusting the number of threads in each team individually achieves a higher performance than the conventional approach by considering runtime overhead.

Third, this dissertation clarifies the advantages of dynamic workload management among OpenMP thread teams. One more solution to tackle load imbalance across thread teams is workload management. This dissertation uses a static workload management mechanism to mimic a dynamic one, so as to quantitatively discuss the upper bound of performance gain of dynamic workload management excluding real scheduling overhead. It also estimates the scheduling overhead according to two assumptions. The first one is that inter-team scheduling overhead can be estimated using the intra-team scheduling overhead. The second one is that the inter-team scheduling overhead increases exponentially with the number of thread teams. By considering the estimated scheduling overhead, the evaluation results show that dynamic workload management across thread teams achieves a better performance in comparison with static workload management.

Accordingly, the contributions of this dissertation are threefold. First, it provides a mechanism for runtime processor selection on heterogeneous systems. Second, it clarifies the benefits of dynamic thread management across OpenMP thread teams. Third, it clarifies the advantages of dynamic workload management across OpenMP thread teams.

1.3 Organization of the Dissertation

The rest of the dissertation is organized as follows. Chapter 2 proposes an approach for runtime processor selection using directive customization. Chapter 3 discusses thread management across OpenMP thread teams. Chapter 4 discusses dynamic workload management across OpenMP thread teams. Chapter 5 gives concluding remarks of this dissertation.

Chapter 2

Runtime Processor Selection on Heterogeneous Systems

2.1 Introduction

Recent years, HPC systems are becoming more and more heterogeneous. They are often coupled with various types of processors. For example, one of the top 500 supercomputers [11], Tianhe-2 [12], is equipped with Intel Xeon Processors and Xeon Phi Processors. In such an HPC system, a different processor has a different performance characteristic.

The problem settings of processor selection are described as follows. First, this chapter assumes that different processors (CPUs and accelerators) are available to execute an irregular application, and the problem size is determined when executing the application. An irregular application may have different code versions, each of which is written for a different problem size. In general, each code version is suitable for a different processor. Second, this chapter assumes that OpenMP directives have already been inserted into the application for parallelizing the application.

A conventional approach to processor selection is to allocate data-parallel computation on accelerators while allocating non data-parallel computation on CPUs. It is not always the best solution since it underestimates the computing capability of CPUs. CPUs might achieve comparable performances with accelerators even for some data-parallel computa-

```
if (USE_CPU == 1){
  #pragma omp parallel for
  for(Index_t gnode = 0; gnode < numNode; ++gnode){
    /*Loop body*/
  }
}
else if (USE_GPU == 1){
  #pragma omp target data map(to: nodeElemStart[:len1])
  \ map(to: nodeElemCornerList[:len2])
  {
    #pragma omp target teams num_teams(TEAMS)
    \ thread_limit(THREADS)
    # pragma omp distribute parallel for
    for(Index_t gnode = 0 ;gnode < numNode; ++gnode){
      /*Loop body*/
    }
  }
}
```

Figure 2.1: An example where processor selection mechanism is written in the application.

tions.

Processor selection often needs various code versions of a given application, because each processor requires a different code version. Moreover, code transformation is required to generate various code versions. It is because a given application needs to be transformed in various ways to adapt for various processors. Many code transformations are provided as the forms of compiler directives, because using compiler directives does not severely mess up a given application. Usually, a compiler directive is associated with one code transformation, and the transformation behavior can be changed by changing the parameter values in the directive. Overall, to generate various code versions for processor selection, various code transformations are required, and thus various compiler directives are also required. One technical challenge is how to generate various code versions without using various compiler directives, but using only one directive. The merit of using only one directive is that programmers do not need to maintain all code versions.

The research problem of this chapter is that programmers need to implement a processor selection mechanism in their applications, which messes up the applications and

makes the applications difficult to maintain. Figure 2.1 shows an example where a processor selection mechanism is written in the application. In the example, the code sections of CPU and GPU are explicitly separated by using an if-else statement. An *omp parallel for* directive is used in the CPU section, and an *omp target teams* directive is used in the GPU section. In this way, the loop body is duplicated, and thus the number of code lines is almost doubled. Therefore, a processor selection mechanism written in a source application makes the application difficult to maintain.

The objective of this chapter is to avoid severely modifying an application for runtime processor selection. To this end, this chapter proposes directive customization. The main idea behind the proposed approach is that directive customization can customize the existing compiler directives in a given application, and change their behaviors of the existing directives to generate various code versions without messing up the application itself.

The current OpenMP specification does not provide any mechanism for processor selection. Note that this chapter does not extend the OpenMP programming model itself. However, this chapter extends the processor selection mechanism in the current OpenMP specification. In the current OpenMP specification, programmers need to decide which processor to be used to execute a computation kernel. In this chapter, a source code can be transformed into various code versions by using customized directives, and a code version associated with an appropriate processor is selected at runtime.

Directive customization basically requires two key components. The first component is a directive format to express the directive specification, such as directive names and clause names. This chapter proposes an XML format to express the directive specification. The second component is a translation framework. This chapter uses Xevolver [9, 10] and a customizable compiler directive parser (CCDP) to translate a source code annotated with customized directives into various versions, based on translation rules written in external files. The generated code versions can be executed on various processors. More details about the proposal can be found in Section 2.4.

The contributions of this chapter are threefold. First, this chapter presents an ap-

proach to runtime processor selection using customized compiler directives. Second, this chapter proposes an XML format to express directive specifications. Third, this chapter introduces a directive parser that can parse a directive string into XML elements.

The rest of this chapter is organized as follows. Section 2.2 describes the related work. Section 2.3 shows a motivating example for runtime processor selection. Section 2.4 presents the proposed approach to runtime processor selection. Section 2.5 shows the evaluation results, and Section 2.6 gives concluding remarks of this chapter.

2.2 Related work

This section describes the related work. It includes three parts. The first part discusses related work with directive-based programming models. The second part presents related work with code transformation, and the third part discusses some approaches to processor selection.

2.2.1 Directive-based programming models

To ease the programming of HPC systems, directive-based programming models become popular. Widely-used directive-based models include OpenMP [5] and OpenACC [7]. XcalableMP [15, 16] is a directive-based parallel programming language for distributed-memory systems. It provides global-view and local-view programming models to parallelize a sequential program, and also supports OpenMP + MPI [17] hybrid programming. The OmpSs [18, 19] programming model is to extend OpenMP with new directives and clauses to support asynchronous parallelism and heterogeneity. Though the latest version of OpenMP has already supported such features, OmpSs can be considered as a precedent of heterogeneity support.

Several researches have extended existing compiler directives to new ones. Lee et al. [20] have extended the OpenMP directives for programming and tuning for GPUs, which are called OpenMPC directives. XcalableACC [21] is a directive-based language extension which is a combination of XcalableMP and OpenACC. OpenMPD [22] is another directive-based extension of a data parallel language. It supports typical parallelization patterns based on data parallel paradigm and work sharing.

2.2.2 Code transformation

ROSE [23] [24] provides a common infrastructure to support user-defined tools, so that the users do not need to develop complicated implementations of code analysis and transformation operations. It provides various high-level helper functions to do user-defined code transformations by manipulating the Abstract Syntax Tree (AST), which describes

how to handle loops, statements and expressions. For example, those functions can help users set the lower and upper bounds of a loop, and insert, remove or replace an expression in the original code. As a result, ROSE users can implement their own translators using those helper functions. In addition, the ROSE compiler allows users to create new compiler directives. However, they need to develop and maintain their own directive parsers in C++. Therefore, if users create their own directives using the ROSE compiler, additional programming efforts and maintenance costs are required for such a directive parser in addition to the application code.

Several script languages [26–29] have been proposed to support code transformation. One of the noticeable script languages is the POET [25] language. POET stands for Parameterized Optimizations for Empirical Tuning. The POET framework is essentially a POET interpreter coupled with some transformation libraries. It can transform the input codes in various ways, according to parameter values and the translation rules written in external POET script files. The POET script language is used to describe the translation rules of the transformations in the POET framework.

Although POET provides the POET language [28] [30] to describe the customized translation rules, it does not support the definition of custom compiler directives. Therefore, even if programmers define a special transformation rule, they cannot use a special compiler directive to annotate the code to be transformed.

Xevolver [9, 10] is an extensible programming framework that can expose an AST as XML data to programmers in a text format. The original Xevolver implementation is built on top of the ROSE compiler infrastructure, and achieves inter-conversion between ROSE Sage III ASTs and XML ASTs. Thus, Xevolver can use user-defined code transformations as well as a large number of code transformations provided by the ROSE compiler. Figure 2.2 shows the workflow of the Xevolver framework to transform the source code. Firstly, a source code written in C or Fortran is parsed into ROSE Sage III ASTs using the ROSE parser. Then, Sage III ASTs are converted to XML ASTs. After that, XSLT (eXtensible Stylesheet Language Transformations) [13] [14] rules are applied to the XML ASTs to convert them into new XML ASTs. XSLT is used to describe the translation

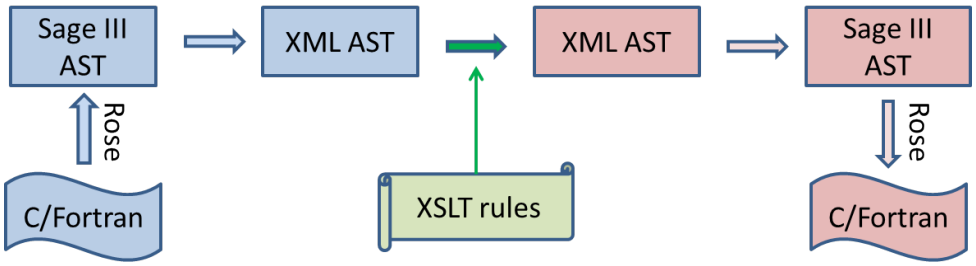


Figure 2.2: The workflow of the Xevolver framework.

rules in external files. The new XML ASTs are then transformed back into ROSE Sage III ASTs, which are finally unparsed into optimized versions of the source code using the ROSE unparser. Different XSLT translation rules would result in different optimized versions of the source code. This chapter uses the Xevolver framework to generate different code versions that are suitable for execution on different processors.

2.2.3 Processor selection

Some studies [31] [32] have proposed methods for selection of embedded processors. Martin-del-Brio et al. [31] have presented an approach to embedded processor selection. Their target systems are embedded systems. Their approach is based on self-organizing maps [33, 34], a neural network commonly used for visualizing large databases using the form of two-dimensional maps. The resulting 'microprocessor maps' is a valuable tool for the design engineer to select a specific microprocessor or micro-controller device for a particular embedded application. On the other hand, although this chapter basically assumes a heterogeneous HPC system with many cores, the proposed approach can be applied not only to HPC systems but also to embedded systems.

Mastronarde et al. [35] have presented a processor selection scheme for video-decoding applications. Their purpose of selecting a proper processor is to minimize the *queuing delays* of multimedia applications. The queuing delays are important for delay-sensitive multimedia applications because the derived video quality depends on processing jobs before their deadlines. Their approach is limited to multimedia applications, while the processor selection approach introduced in this chapter does not have such a limitation.

Takizawa et al. [36] have proposed a runtime processor selection mechanism for energy-aware computing. Their purpose is to maximize the energy efficiency. They can generate various code versions using the SPRAT framework [37]. Each code version is corresponding to one computing engine. According to the runtime behavior of a code, SPRAT can dynamically switch the computing engine for executing each code so as to minimize the energy consumption. Unlike the approach in [36], this chapter uses directive customization to generate multiple code versions. Each code version is suitable for execution on a particular type of processor.

Table 2.1: The experimental environment for motivating example.

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

2.3 Motivation

A motivating example of processor selection is shown in this section. An irregular application is used for preliminary evaluation. As a typical example of irregular applications, a ray tracing code is used as a target code. The experimental environment is shown in Table 2.1. The ray tracing algorithm [93] [94] 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. Figure 2.3 shows an example of the hotspot of the ray tracing algorithm, which is based on the code available at [95], and modified to render multiple images by adding the outermost loop and changing the camera position and the sampling rate. In the example, all objects in the scene are fixed, and the camera position is changed to generate a movie. An iteration of the outermost loop generates one frame of the movie. The computational cost of tray tracing depends on the camera position, and hence the execution time of each iteration of the outermost loop drastically changes. Accordingly, the modified version of the ray tracing program is an irregular application.

In this example, the performances of executions on a CPU and KNC (Intel Xeon Phi, Knights Corner series) are compared, while the image sizes are changed from 4×4 pixels to 512×512 pixels. The execution on CPU uses a single thread, while the execution on KNC uses 240 threads. An OpenMP *target* construct is used to offload the hotspot to the KNC. The results are shown in Figure 2.4. The horizontal axis represents the image sizes, and the vertical axis represents the speedup ratio against the CPU version.

The results in Figure 2.4 show that the CPU works faster than the KNC when the image size is smaller than 16×16 pixels, while the KNC works faster when the image size is greater than 16×16 pixels. It means that different processors are suitable for executing different problem sizes. However, it is difficult to appropriately select an a processor if

```
#pragma omp target data map(alloc:c[0:w*h],temp[0:N][0:w*h])
for(int t=0;t<N;t++){
  #pragma omp target teams num_teams(4) thread_limit(60)
  {
  Ray cam = change_camera_position_here (t);
  for (int y=0; y<h; y++){
    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++){
        for (int sx=0; sx<2; sx++, r=Vec()){
          for (int s=0; s<samps; s++){
            /*Calculate radiance */
            /* It depends on materials of objects*/
            /* It also depends on camera position &
               direction*/
          }
          /*Accumulate radiance */
        }
      }
    }
  }
  }
}
```

Figure 2.3: A code example of ray tracing.

the image size is determined at runtime. Therefore, this example motivates a runtime processor selection mechanism to efficiently run the application for any image size.

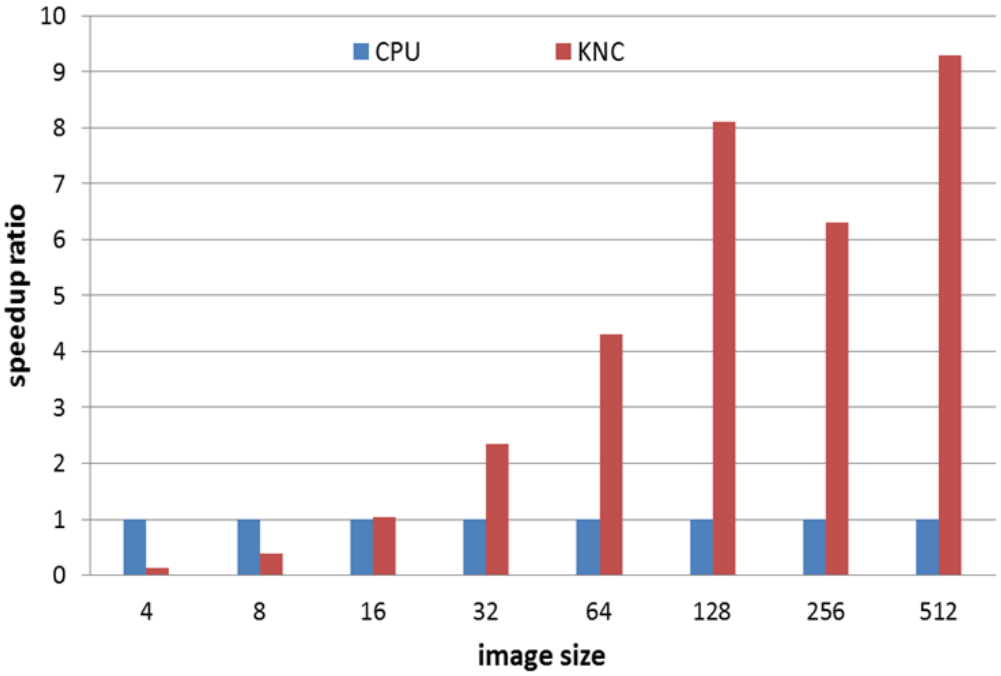


Figure 2.4: Performance comparison of CPU and KNC.

2.4 Runtime processor selection using directive customization

This section discusses runtime processor selection using directive customization. First of all, an XML-based approach to directive customization is described. The XML-based approach is used because it is easy to collaborate with the code translation framework, Xevolver, which also uses XML-based ASTs. Then, processor selection based on directive customization is discussed.

2.4.1 An XML-based approach to directive customization

This subsection presents an XML-based mechanism that allows programmers to customize existing compiler directives. The mechanism consists of the CCDP and an XML format. The CCDP can convert a directive string into XML elements, so that translation rules in XSLT can easily handle parameters in the directive string. The XML format is used to describe the specification of the customized directives, such as directive names, clauses, and their arguments.

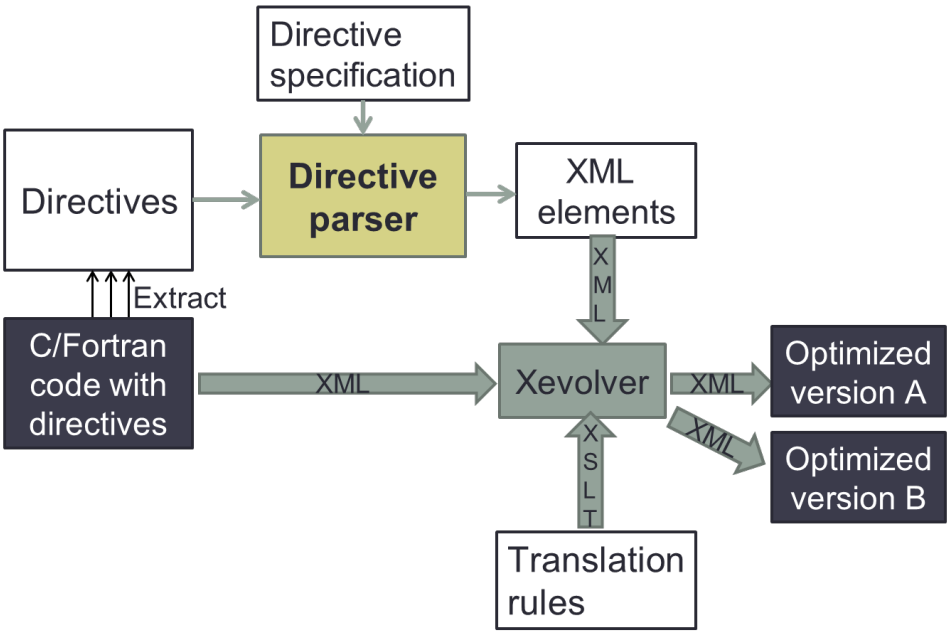


Figure 2.5: The overview of the Xevolver-CCDP framework.

2.4.1.1 The overall framework for customization of compiler directives

The programming framework discussed in Section 2.4.1.1 is constructed by collaboration between CCDP and Xevolver. Figure 2.5 illustrates the overall programming framework, called the Xevolver-CCDP framework. In the Xevolver-CCDP framework, a directive parser as well as C and Fortran parsers is provided. In addition, the specifications of user-defined directives are described in an XML format. Then, the directive parser can parse the directive strings and convert them into XML elements defined by the XML format. Finally, the XML elements are used by Xevolver to configure its translation rules. Note that the source code is also converted into an XML document, i.e., an XML AST. As a result, it is expected that the translation rules in XSLT can handle the directives and the source code in the same manner. Therefore, the Xevolver-CCDP framework can transform the source code into various code versions that are suitable for various types of processors.

By using the proposed XML format to describe the directive specification, the Xevolver-CCDP framework can separate the directive definition from the translation rules. It leads to two advantages as follows: (1) The specification of a directive, such as its directive

name, clauses, and parameters, can be clearly seen in the XML format. (2) The translation rules in the Xevolver-CCDP framework become reusable because analysis of a directive string is given to the directive parser. The programmers do not need to rewrite a new translation rule as long as the transformation rules are unchanged.

The CCDP converts a directive string into XML elements, based on the specification described in an XML format. CCDP assumes that the syntax of every customized directive follows that of an OpenMP directive. By converting the directive string into XML elements, the clauses and parameter values of the directives can be clearly seen in the XML elements. Then, the translation rules in XSLT can easily handle the parameter values that are expressed in XML. The CCDP is responsible to recognize different components in a directive string according to different keywords set in the specification description, and most importantly, to replace the parameter values in the specification with the values in the directive string. This is to ensure that the parameter values written in a directive appear in an output XML document.

2.4.1.2 The XML data format for customizing compiler directives

Suppose that the C language is used in application development. Then, a compiler directive may usually contain the following components: A symbol #, a keyword *pragma*, a keyword indicating the directive type and a directive name. One or more clauses and their parameter lists could be additionally written in the directive. For example, regarding an OpenMP directive *"#pragma omp parallel num_threads (5)"*, the keyword *pragma* is followed by the directive type *omp*, which indicates that it is an OpenMP directive. Then, the directive name *parallel* appears after *omp*, followed by the clause name *num_threads*. Finally, the clause takes a parameter list that is *5* in this case.

In order to describe different components of a customized compiler directive, several keywords are provided in the XML format. The XML format is called the extensible directive definition markup language (XDDML). Assuming that a compiler directive in the C language starts with *"#pragma"*, the specification of a compiler directive requires one or more keywords for directive names, clause names, and parameter lists. Therefore,

```
<DEFS>
  <DIRECTIVE name = "x">
    <CLAUSE name = "y">
      <PLIST>
        <IDENTIFIER name = "z">
          <LI value = "a"/>
        </IDENTIFIER>
      </PLIST>
    </CLAUSE>
  </DIRECTIVE>
</DEFS>
```

Figure 2.6: The XML format to express custom compiler directives.

Table 2.2: The Descriptions of the keywords.

Keyword	Description	Optional or not
DEFS	Indicates the beginning and the end of a directive definition	No. It is mandatory
DIRECTIVE	Indicates the name of a directive	No. It is mandatory
CLAUSE	Indicates the clause name of a directive	No. It remains even if no clause in the directive. In such case, the clause name is blank
PLIST	Indicates the parameter list in a directive. It is used when the number of parameters is unfixed	Yes. It can be omitted if the number of parameters is fixed
IDENTIFIER	Indicates the identifier in a directive	Yes. It can be omitted if no identifier in the directive
LI	Indicates the parameter list item	Yes. It can be omitted if no parameter in the directive

several keywords are provided in the XDDML.

Figure 2.6 shows the XDDML to describe the specification of custom compiler directives. Using XDDML, the definition of a custom compiler directive is described in an XML document with predefined keywords such as DEFS, DIRECTIVE, and CLAUSE. The root element of a compiler directive definition file is expressed with keyword DEFS. Keyword DIRECTIVE indicates the directive name. Keyword CLAUSE indicates a clause that can appear in the directive string. Besides, there are keywords, such as PLIST, IDEN-

TIFIER and LI, which indicate a parameter list, an identifier and a parameter list item, respectively. Table 2.2 lists the descriptions of all the keywords.

```
<DEFS>
  <DIRECTIVE name = "flatten">
    <CLAUSE name = "param">
      <LI value = "a"/>
      <LI value = "b"/>
    </CLAUSE>
  </DIRECTIVE>

  <DIRECTIVE name = "interchange">
    <CLAUSE name = "loop">
      <LI value = "x"/>
      <LI value = "y"/>
    </CLAUSE>
  </DIRECTIVE>
</DEFS>
```

Figure 2.7: Directive definition using XDDML.

```
for(i = 0; i < 50; i++){
  X = i + 1;
  Y = 4 * i;
  #pragma xev flatten param(1, 1000)
  for(j = X; j < Y; j++){
    sum = sum + 3;
  }
}

#pragma xev interchange loop(1, 2)
for(i = 0; i < 100; i++){
  for(j =0; j < 100; j++){
    sum = sum + 3;
  }
}
```

Figure 2.8: Compiler directives in the original code.

The XDDML is designed to define OpenMP-like directives. Figure 2.7 shows an example of the XDDML to describe the custom compiler directives annotated in the source code shown in Figure 2.8. The directive string *"#pragma xev flatten param(1, 1000)"* is

described using XDDML. The directive type is given as *xev*, the directive name is indicated as *flatten*, and the clause name is written as *param*. The directive string contains two parameter values, *1* and *1000*. Thus, in the specification description, two parameters are specified using the keyword *LI*. Note that the parameter values in the specification description can be different from that in the real directive. Accordingly, in Figure 2.7, the parameter values are designated as some variables, such as *a* and *b*, instead of *1* and *1000*, respectively. The parameter values are explicitly listed in the description of the directive specification. It allows XSLT rules to easily extract and even modify the parameter values, because the directive string is expressed using XML elements that can be handled in XSLT rules.

2.4.2 Runtime processor selection based on directive customization

As an example of directive customization, by using the XDDML, an OpenMP directive *"#pragma omp parallel"* is customized to have a special clause named *"cond"*. The directive definition using XDDML is shown in Figure 2.9. This directive is used to automatically generate three code versions, which are a single-thread version, a multiple-thread version, and a KNC version, respectively. The generated code versions are illustrated in Figure 2.10. The single-thread version contains no OpenMP directives, which means the code is executed on CPU using a single thread. The multiple-thread version contains the directive *"#pragma omp parallel for num_threads(auto)"*, which means the code is executed on CPU using multiple threads. The KNC version includes an OpenMP *target* directive *"#pragma omp target"*, which means the code is offloaded to an accelerator for execution. The code transformation rule associated with the directive *"#pragma omp parallel cond(N1, N2)"* is implemented using XSLT. The directive string contains two parameter values, *"N1"* and *"N2"*, which describe two thresholds. The thresholds are given by programmers, and they can be obtained by offline tuning.

By comparing the two thresholds with the problem size N of a given application,

```
<DEFS>
  <DIRECTIVE name = "parallel">
    <CLAUSE name = "cond">
      <LI value = "N1"/>
      <LI value = "N2"/>
    </CLAUSE>
    <CLAUSE name = "num_threads">
      <LI value = "a"/>
    </CLAUSE>
  </DIRECTIVE>
</DEFS>
```

Figure 2.9: The definition of the customized OpenMP directive.

one can select an appropriate processor to execute the application. Figure 2.10 shows a simple example of selecting a code version for execution. If the problem size N is smaller than the threshold *N1*, the single-thread version is selected. Else if the problem size N is smaller than the threshold *N2*, the multi-thread version is selected. Otherwise, the KNC version is selected. The three code versions are listed using *"if-else"* statements as shown in Figure 2.10. Therefore, only one of the three versions will be selected for execution at runtime. The two thresholds *N1* and *N2* are important factors to properly select one type of processors to execute the application. The values of the thresholds can be obtained after one trial run of the given application.

In Section 2.4.2, processor selection based on the problem size is discussed as an example of processor selection. In the case of selecting processors based on other features, different customized directives would be required.

```
if(N<N1){
  /*single-thread version executed on CPU*/
  for(i=0;i<N;i++){
      /*Loop body*/
  }
}
else if(N<N2){
  /*multi-thread version executed on CPU*/
  #pragma omp parallel for num_threads(X)
  for(i=0;i<N;i++){
      /*Loop body*/
  }
}
else{
  /*KNC version executed on accelerator*/
  #pragma omp target
  #pragma omp paralle for
  for(i=0;i<N;i++){
      /*Loop body*/
  }
}
```

Figure 2.10: Three code versions generated by the customized OpenMP directive using *"if-else"* statements.

2.5 Evaluations

2.5.1 Experimental setup

The system used for evaluation is the same as in Section 2.3. In the evaluation, three target applications are used. They are ray tracing, Sieve of Eratosthenes (SoE), and n-body simulation applications.

The SoE [38, 39] is a simple algorithm to calculate prime numbers that are smaller than a given number. It does so by iteratively eliminating the multiples of each prime number. The kernel of the SoE code is shown in Figure 2.11, which is based on the code available at [40]. It is an irregular code because the loop length of the inner loop depends on the loop index of the outer loop. Therefore, each iteration of the outer loop costs a different execution time.

The n-body simulation [41] is a simulation of particles' movement under the influence

```
for(i = 2; i < size; i++){
  if(prime[i] = PRIME){
    for(j = i + i; j < size; j += i){
      prime[j] = NONPRIME;
    }
  }
}
```

Figure 2.11: The kernel loop of SoE.

Table 2.3: Some parameters used in n-body code.

TIME_LIMIT	TIME_STEP	Dataset
316	1	Dubinski

of physical forces, such as gravitational force. The Barnes-Hut [42–44] algorithm is a typical method used in n-body simulation to reduce the computational complexity. In the Barnes-Hut algorithm, the space of particles is divided into small cells, and only interactions between particles from nearby cells need to be treated individually. Particles in distant cells are treated as a large particle whose mass center is the center of the distant cell's center of mass. The cells are also refined to smaller ones in the case of many particles per cell. In this way, the number of particle pair interactions is drastically reduced, and thus the computational complexity is drastically reduced. Figure 2.12 shows an example of n-body codes using the Barnes-Hut algorithm, which is based on the code at [45]. In this code, an octree [46–48] is firstly constructed, and each node of the octree represents a cell of particles. Then, the mass center of each node is calculated. Finally, the forces are calculated and the particles are updated by applying the forces. The n-body (Barnes-Hut) code is irregular because the particles' distribution is irregular, and thus traversing the octree costs a different time for a different node. Furthermore, it uses approximation to calculate forces, and thus calculating forces costs a different time for a different node. Some parameters used in the evaluation of n-body code are shown in Table 2.3. Note that the dataset of particles used in the evaluation is Dubinski [49, 50] dataset, which is a typical dataset in n-body simulation.

Three code versions of each target application are generated. The single-thread version

```
for(time =0.0; time <TIME_LIMIT; time += TIME_STEP)
{
  // Construct octree
  for (long i = 0; i < N; i++) {
    octree_insert(&roots[idx], particles[i],
    \ &node_count[idx]);
  }
  // Calculate the mass center
  for (long i = 0; i < 8; i++) {
    octree_transform(microot, &micmemptrs[i], &roots[i]);
  }
  // Calculating forces and updating particles
  #pragma omp target map(tofrom:particles[0:N])
  \ map(to:microot[0:total_nodes])
  #pragma omp teams num_teams(2) thread_limit(120)
  \ shared(N,max_lim, min_lim)
  #pragma omp parallel for schedule(static)
  for (long i = 0; i < N; i++) {
    physics_calc_force_mic(microot, microot, &particles[i]);
    physics_apply_force(&particles[i], max_lim, min_lim);
  }
}
```

Figure 2.12: An example of n-body codes using Barnes-Hut algorithm.

is executed on CPU using one thread. The multi-thread version is executed on CPU using multiple threads. OpenMP *parallel for* directive is used for parallel execution, and *num_threads* clause is used to assign the number of threads. The KNC version is executed on Intel Xeon Phi using OpenMP *target* directive with 240 threads.

2.5.2 Evaluation results

The evaluation results of the ray tracing code are shown in Figure 2.13. The horizontal axis represents image sizes, and the vertical axis represents the speedup ratios against the single-thread version. The results show that the 8-thread version achieves the best performance when the image sizes are smaller than 64×64 pixels, and the KNC version achieves the best performance when the image sizes are larger than 128×128 pixels. The 8-thread version achieves up to 6.4x speedup against the single thread version when the

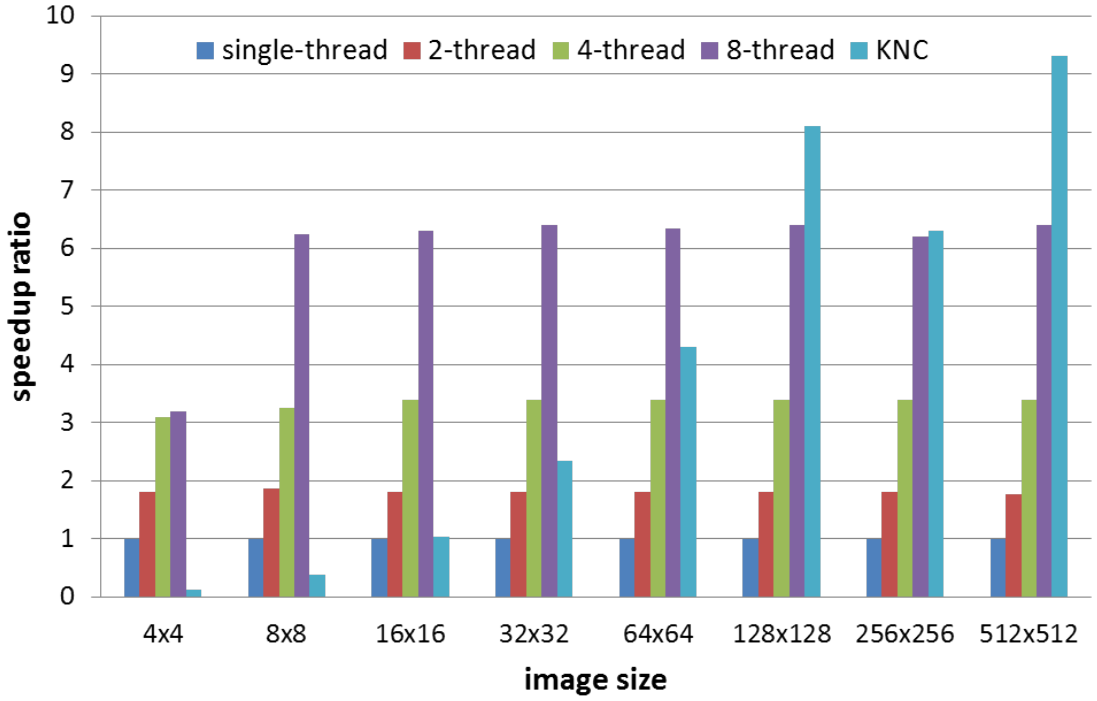


Figure 2.13: Performance comparisons of different code versions of ray tracing.

image size is less than 64×64 pixels. The KNC version achieves up to 9.3x speedup against the single-thread version when the image sizes are larger than 128×128 pixels. This indicates that the CPU instead of the KNC should be selected when the problem size is not large, because CPU provides enough parallelism for small and medium problem sizes. On the other hand, the KNC should be selected when the problem size is large, because the KNC instead of the CPU can provide higher parallelism for a large image size. For a certain image size, the performances of single-thread, 2-thread, 4-thread, and 8-thread versions are saturated because the maximum parallel efficiency is achieved on CPU for those code versions. In this case, *N1* can be equal to 5, and *N2* can be equal to 65.

The evaluation results of SoE are shown in Figure 2.14. The horizontal axis represents size, and the vertical axis represents the speedup ratios against the single-thread version. The performance results show that CPU performs better than KNC when the size is 1000, while KNC performs better than CPU when the size is larger than 10000. In this case, *N1* can be equal to 0, and *N2* can be equal to 1001.

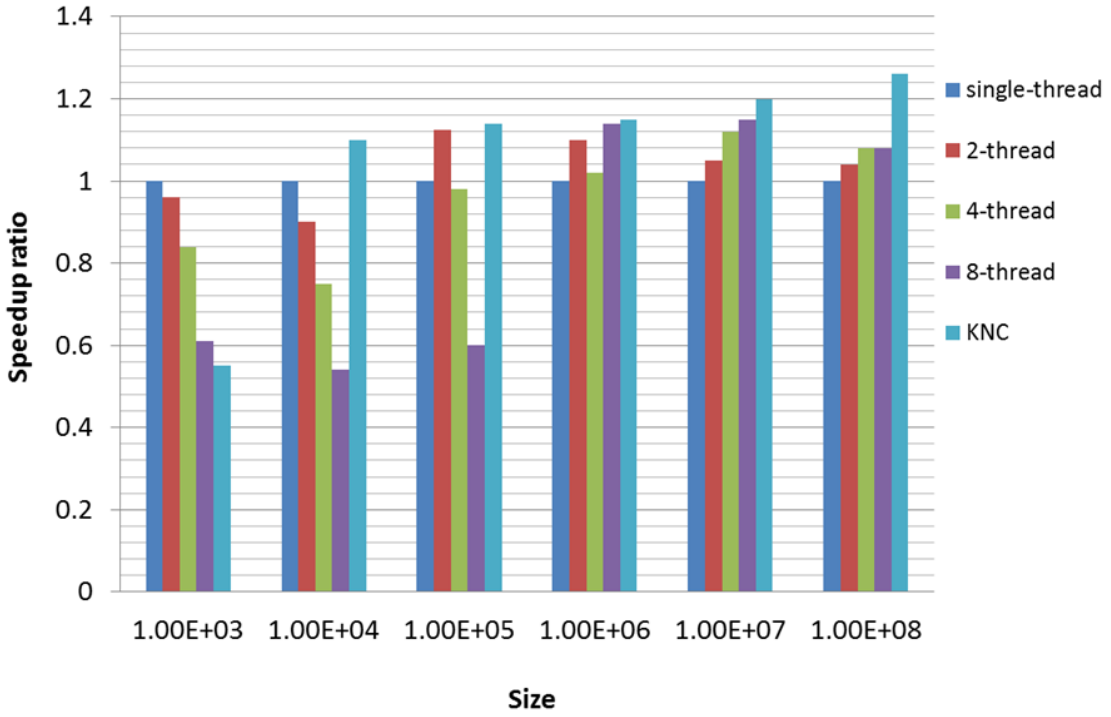


Figure 2.14: Performance comparisons of different code versions of SoE.

The evaluation results of n-body are shown in Figure 2.15. The horizontal axis represents the number of particles, and the vertical axis represents the speedup ratios against the single-thread version. The performance results show that different code versions lead to different performances. When the number of particles is medium such as 2048, CPU achieves a better performance than KNC. If the number of particles is large such as 32768, KNC performs better than CPU because KNC provides higher parallelism for a large number of particles. In this case, *N1* can be equal to 129, and *N2* can be equal to 8193.

Note that the two thresholds introduced in Section 2.4.2 can be designated by programmers. In the evaluations, the thresholds *N1* and *N2* need to be set properly. In order to find the two thresholds, *N1* and *N2*, programmers need to do offline tuning. With proper settings of the two thresholds, programmers can use directive customization to select an appropriate processor (CPU or KNC) at runtime for the target codes with various problem sizes.

The evaluation results show that *N1* and *N2* are not sensitive to affect performance

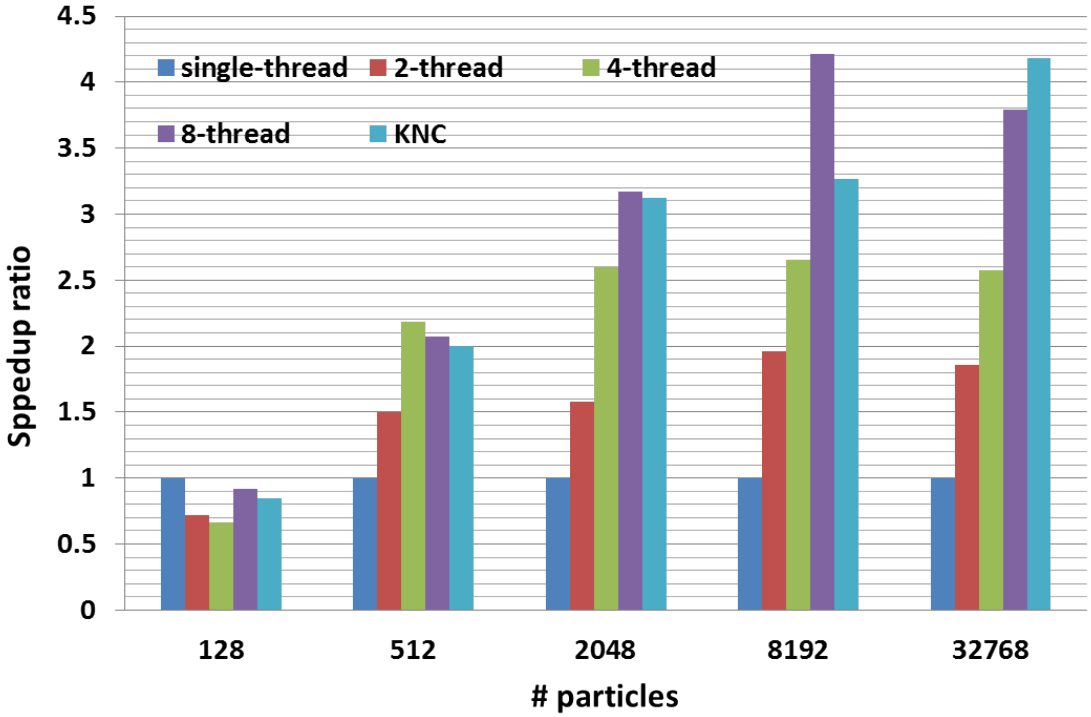


Figure 2.15: Performance comparisons of different code versions of n-body.

if the values of *N1* and *N2* do not exceed a range. It is because *N1* and *N2* are not sensitive to affect processor selection decision. As long as the processor selection decision is unchanged, the performance is unchanged.

If directive customization is used, a source program can be transformed into various code versions without messing up the program itself. Conversely, if directive customization is not used, programmers need to severely modify the program by duplicating code sections as shown in Figure 2.1. In terms of the number of code lines, there is a big difference between the manual code modification and the proposed directive customization approach. Figure 2.16 shows the difference of the number of code lines for each target code. The figure shows that the number of code lines is significantly reduced by using directive customization. It implies that, by using directive customization, a source program can be kept clean and easy to maintain.

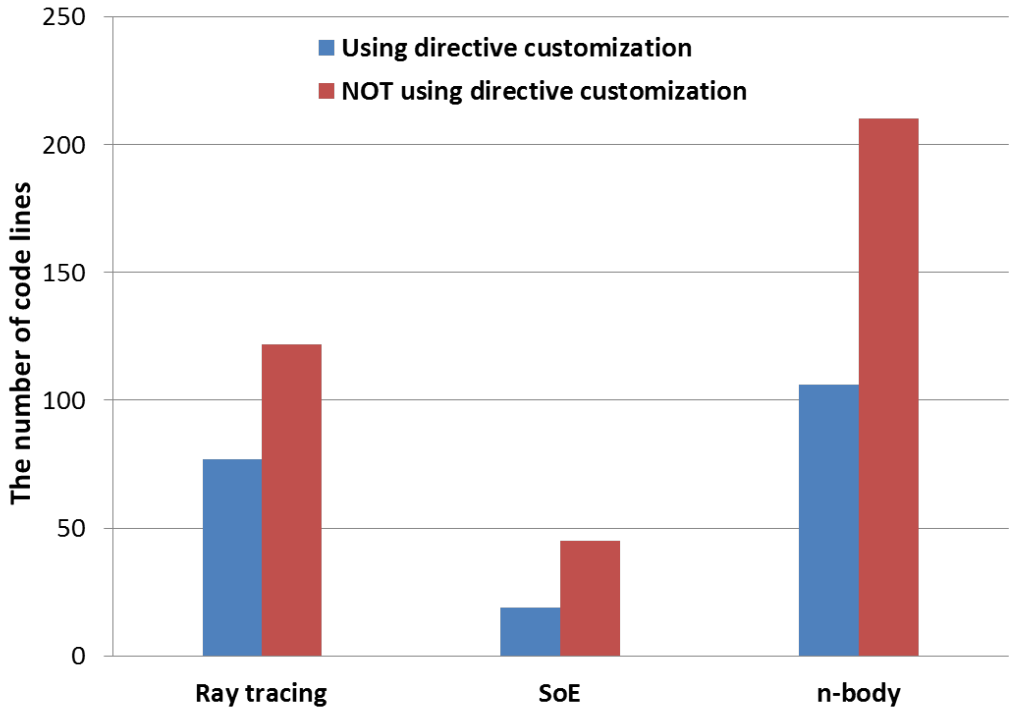


Figure 2.16: The difference of the number of code lines for each target application.

2.6 Conclusions

The objective of this chapter is to avoid severe code modifications for runtime processor selection. To this end, this chapter proposes directive customization. An OpenMP directive is customized, and three code versions of a given application can be generated using the Xevolver-CCDP framework. The code versions include a single-thread version, a multi-thread version and a KNC version. They are listed one by one using *if-else* statements. By comparing two given thresholds with the problem sizes of a given application, an appropriate processor can be selected at runtime to execute the application.

This chapter demonstrates that combining code transformation and directive customization can satisfy the requirements for runtime processor selection of existing irregular OpenMP codes. In addition, directive customization can help programmers to reduce code modifications for runtime processor selection, and thus a given application can be kept clean and easy to maintain.

Chapter 3

Dynamic Thread Management for Irregular Applications

3.1 Introduction

Nowadays, the architecture of HPC systems is becoming complex with more and more cores integrated into the systems. Therefore, parallel programming is essential to exploit the computing power of the HPC systems. However, parallel programming for HPC systems is becoming more challenging because programmers often need to exploit massive thread-level parallelism on the system.

Fortunately, the OpenMP programming model provides various compiler directives and runtime library routines to help programmers to exploit thread-level parallelism. The number of threads used for executing an HPC application is an important factor for thread-level parallel execution. This is because a different number of threads leads to a different parallel efficiency. Thus, programmers often need to adjust the number of threads in order to achieve a high parallel efficiency, and thus a high performance.

Tuning the number of threads to execute an application remains challenging. This is because it is difficult to know the optimal number of threads before execution. A conventional approach is to use the maximum number of threads that allows a given system to exploit the parallelism of an application as much as possible. However, it is not

always a good solution. The synchronization overhead usually increases with the number of threads joining the synchronization. If the overhead overwhelms the performance gain from using more threads, the performance degrades by using more threads. Thus, there exists an optimal number of threads that leads to the best performance.

This chapter focuses on irregular applications. It is difficult to efficiently parallelize an irregular application with many threads, because load imbalance is inevitable and also thread synchronization becomes expensive. This chapter demonstrates that the use of multiple thread teams to execute an irregular loop is beneficial because it can reduce the thread synchronization overhead at a certain level. However, use of multiple thread teams does not completely solve the load imbalance among thread teams. Conversely, it might exacerbate the load imbalance problem.

The research problem of this chapter is load imbalance across thread teams for irregular applications. To solve the load imbalance across thread teams, this chapter discusses dynamic thread team size adjustment, which is to adjust the number of threads in each thread team at runtime.

The objective of this chapter is to investigate the benefits of dynamic team size adjustment on the performance of irregular kernels, under the assumption where the synchronization overhead is large and use of multiple thread teams is thus beneficial to attain efficient execution. To this end, this chapter uses a static team size adjustment method to emulate a dynamic one. The upper bound of performance gain by introducing dynamic thread team size adjustment is estimated by exploring the performances of many possible team size combinations in each phase and accumulating the best performance of each phase. Based on the upper bound of performance gain, the practical performance gain is also discussed by considering the runtime overhead required for dynamic thread team size adjustment.

The contributions of this chapter are twofold. First, this chapter presents a method to emulate dynamic thread team size adjustment. Second, this chapter introduces a method to estimate the upper bound of runtime overhead of dynamic thread team size adjustment.

The rest of this chapter is organized as follows. Section 3.2 reviews the related work.

Section 3.3 gives motivating examples of dynamic thread team size adjustment. Section 3.4 presents a way of estimating the performance gain from dynamic thread team size adjustment. Section 3.5 shows the performance evaluation results, and Section 3.6 concludes this chapter.

3.2 Related work

Schonherr et al. [54] have proposed an approach to dynamic thread teams in OpenMP programming. They categorize threads in a thread team into active and idle threads. A dynamic thread team is a team of which the number of active threads can be dynamically adjusted. In their approach, they maintain the overall number of threads in a team constant, and dynamically change the number of active threads while putting idle threads into sleep. In this way, they can dynamically adjust the number of active threads in a thread team without violating OpenMP specification. Their work considers adjusting the number of threads in a thread team for multi-core systems. The purpose of their research is to solve undersubscription and oversubscription problems. On the other hand, this chapter aims to solve the load imbalance across multiple thread teams by adjusting the thread team size. The idle threads in a thread team are migrated to other teams instead of putting idle threads into sleep. This chapter discusses the benefits of extending the OpenMP specification for dynamically changing the number of threads in multiple thread teams.

Several studies [55–58] have provided various ways to adjust the number of threads for many-core systems such as GPUs and Xeon Phis. They use different programming models, either in CUDA [59] or OpenCL [60]. Wang et al. [55] have presented a dynamic thread block launch mechanism in CUDA programming. This mechanism allows a thread to dynamically launch a thread block, and thus supports nested launching of thread blocks. Lashgar et al. [56] have presented an approach to dynamically adjust the number of threads in a warp for GPUs. Hsu et al. [57] have implemented a runtime optimization mechanism that can adjust work-group size and merge work-items in OpenCL programming. Yang et al. [58] have provided a way to manage parallelism from compiler's perspective. They designed a CUDA compiler that allows two thread blocks to merge into one, as well as two threads merge into one. By doing so, the number of expensive global memory accesses is expected to reduce, and data reuse is expected to increase, which leads to performance gain. All those researches use programming models different from

OpenMP. This chapter focuses on adjusting the number of threads in OpenMP thread teams.

Table 3.1: The system for preliminary evaluations.

OS	CentOS Release 6.7
Host CPU	Intel Xeon CPU E5-2690
Accelerator	Intel Xeon Phi 5110P, 240 threads maximum
Compilation	Intel compiler version 16.0.2 with -O3 option

3.3 Motivating examples

Two motivating examples are shown in this section. The first example shows the importance of using multiple thread teams, and the second example shows the importance of adjusting thread team size. Two irregular applications are used for the examples. One is a ray tracing code that is the same as in Section 2.3, and the other is a SoE code that is the same as in Section 2.5. The system used is shown in Table 3.1.

3.3.1 The importance of using multiple thread teams

This subsection shows that multiple thread teams rather than a single one are required to efficiently execute an irregular code. Multiple teams in OpenMP are originally introduced for nested parallelism. However, in this experiment, the purpose is not to provide such nested parallelism. This subsection investigates the performance changes according to the number of thread teams. Therefore, in the experiment, the execution times for different image sizes are measured 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. Although the number of teams varies from 1 to 60, the overall thread count is maintained as 240 regardless of how many teams are created.

The results of the experiments are shown in Figure 3.1. 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. Figure 3.1 illustrates that a different number of teams lead to a different performance even if the image size is unchanged. 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

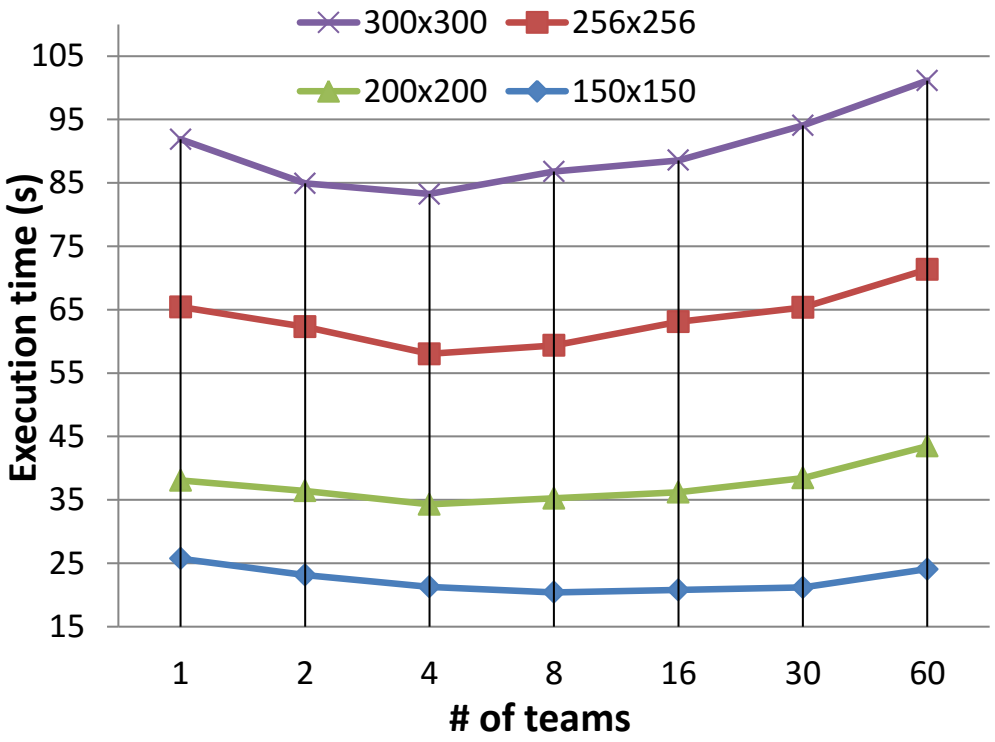


Figure 3.1: Performances with different numbers of thread teams across various image sizes.

too few threads in each team hurt the parallelism in a team. Thus, the above motivating experiment indicates that using multiple teams is potentially better than using a single team for irregular loop nests.

3.3.2 The importance of adjusting thread team size

This subsection shows the importance of adjusting thread team size. The ray tracing and SoE codes are executed using four thread teams. The image size is set to 256×256 pixels for ray tracing, and the problem size of the SoE is set to 1.0×10^7. A thread team size can be designated using a *thread_limit* clause in a *teams* construct. The execution time is measured, while the team size is changed from small to large. Here, all thread teams have the same team size.

The performance evaluation results of ray tracing and SoE are shown in Figures 3.2 and 3.3, respectively. The horizontal axis represents the thread team size, and the vertical axis represents the execution time, in both figures. Figures 3.2 and 3.3 show that different

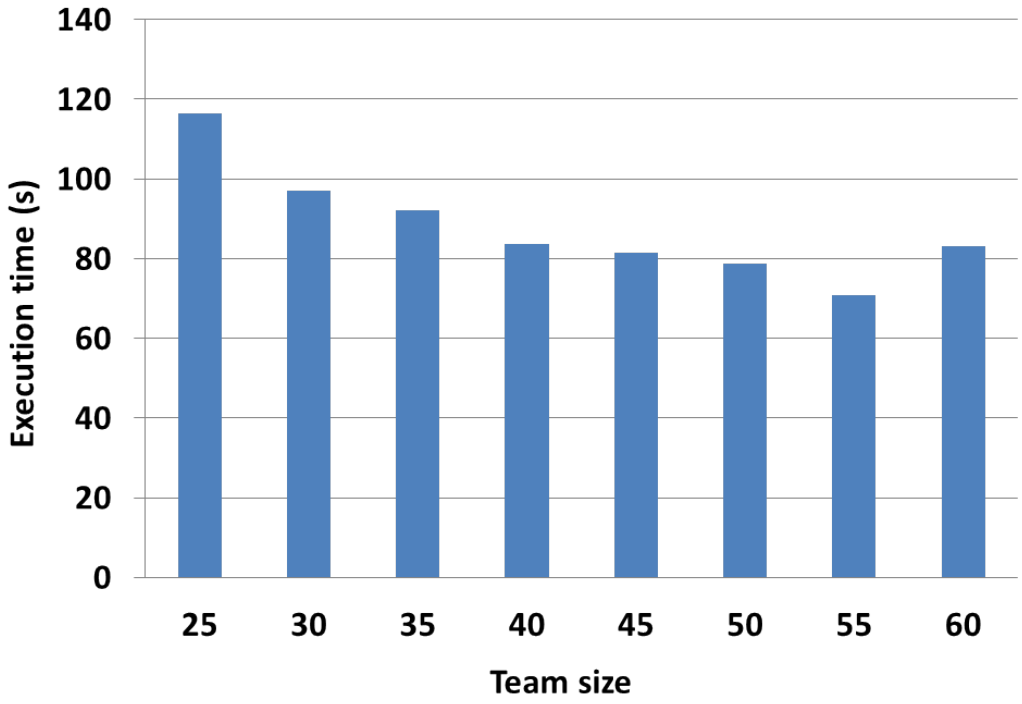


Figure 3.2: Performances with different team sizes for ray tracing. Four thread teams are used for execution.

team sizes lead to different performances. The best team sizes for ray tracing and SoE are 55 and 60, respectively. The figures also indicate that using a larger team size may not always result in a better performance. This is because using more threads may introduce larger synchronization overheads. There can be a transition point when the synchronization overhead overwhelms the performance gain from using more threads. The results in Figures 3.2 and 3.3 imply that the thread team size needs to be adjusted. However, the best team size is unknown in advance of an application execution. Therefore, this chapter discusses dynamic adjustment of thread team size for irregular applications.

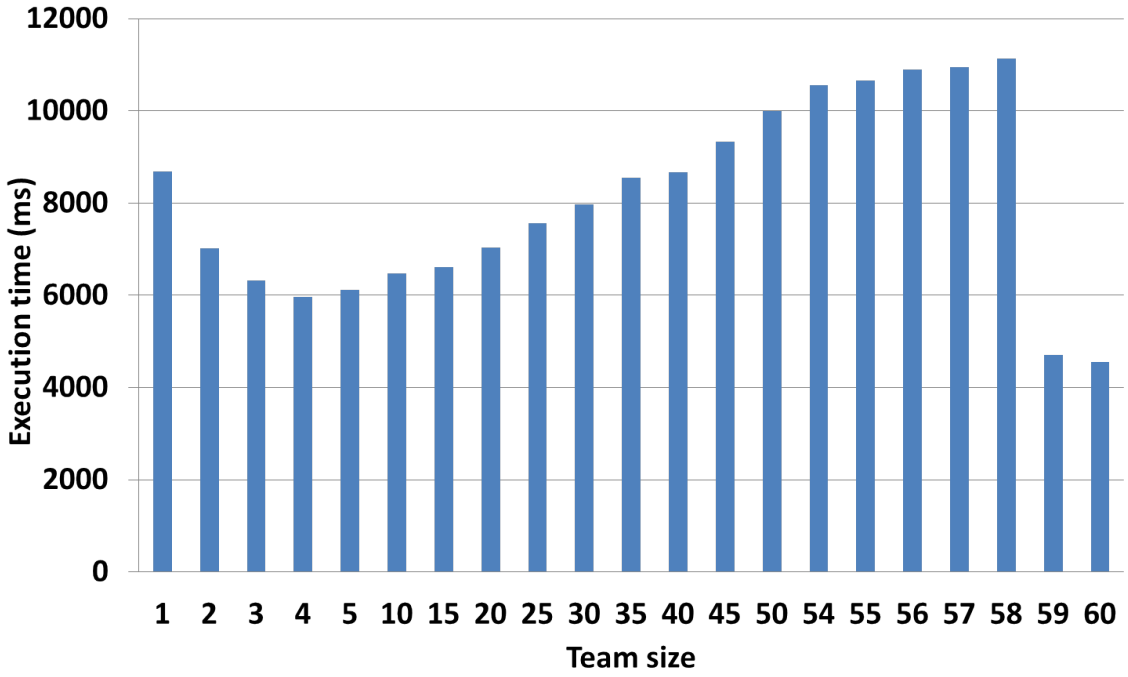


Figure 3.3: Performances with different team sizes for SoE. Four thread teams are used for execution.

3.4 Performance estimation methodology

This section discusses a methodology for estimating the upper bound of performance gain and the runtime overhead of dynamic thread team size adjustment.

3.4.1 Methodology for estimating upper bound of performance gain from dynamic thread team size adjustment

This subsection introduces a method to estimate the upper bound of performance gain from dynamic thread team size adjustment. The method is summarized as follows.

- A given application is assumed to be executed using multiple thread teams. The application execution consists of multiple phases. The performance characteristic of the application changes from phase to phase, and hence the thread teams size should be adjusted for each phase. The numbers of thread teams and phases are denoted as M and N, respectively.

- The execution time of each team for every phase is measured while the team size

is changed. The thread team size for team j is denoted as s_j ($j = 0,\ M - 1$). A thread team size vector v is defined as a vector of the thread team sizes, as shown in Eq. (3.1).

$$v = (s_0, s_1, ..., s_{M-1}). \tag{3.1}$$

The current OpenMP specification supports static adjustment of thread team size vector. The thread team size vector is designated in advance of execution. Thus, with a fixed thread team size vector, the application is executed from beginning to end while measuring the execution time of each phase. The execution is repeated while changing the thread team size vector. As a result, one can obtain the best thread team size vector for each phase that can minimize the execution time of the phase.

- The best execution times of all phases are accumulated to estimate the overall execution time d that can be achieved if dynamic thread team size adjustment works perfectly without any runtime overhead. In other words, d is the upper bound of performance gain due to dynamic thread team size adjustment.

Let $t_i(v)$ be the execution time of phase i ($i = 1,\ N$) when the thread team size vector is v. The execution time d of an application using dynamic thread team size adjustment can be estimated using Eq. (3.2). In Eq. (3.2), the shortest execution time of each phase is accumulated to estimate the overall execution time d, without considering any runtime overhead. Thus, this equation can be used to estimate the upper bound of performance gain from a dynamic thread team size adjustment mechanism.

$$d = \sum_{i=1}^{N} \min_{v} t_i(v). \tag{3.2}$$

To put it simple, Figure 3.4 illustrates the idea of estimating the upper bound of performance gain due to dynamic team size adjustment. Suppose that an irregular application has N phases. In each phase, the execution time changes with the number of threads. There exists an optimal number of threads that leads to the shortest execution time in

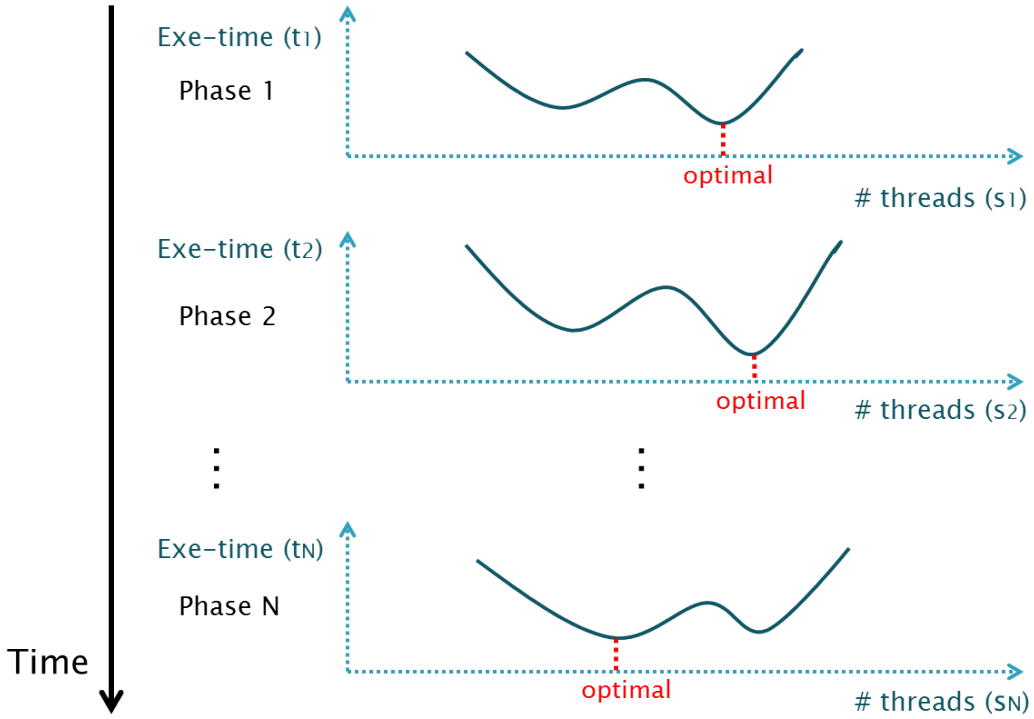


Figure 3.4: Illustration of estimating the upper bound of performance gain due to dynamic team size adjustment.

each phase. The shortest execution times of all phases are accumulated to estimate the upper bound of performance gain due to dynamic team size adjustment.

3.4.2 Methodology for estimating runtime overhead of dynamic thread team size adjustment

This subsection presents a method for estimating the runtime overhead of dynamic thread team size adjustment.

The main idea of dynamic thread team size adjustment is to dynamically migrate idle threads from one team to another. In this approach, the thread migration is performed by destroying the current thread teams and spawning new ones with different thread team sizes. It is expected that the runtime overhead in such a case is larger than other cases where thread migration is performed without destroying and spawning thread teams.

The performance improvement obtained from such thread migration substantially depends on the algorithm employed for the team size adjustment. To determine the upper

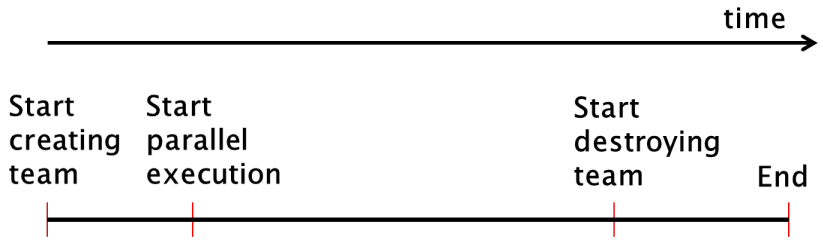


Figure 3.5: Three procedures of a parallel execution.

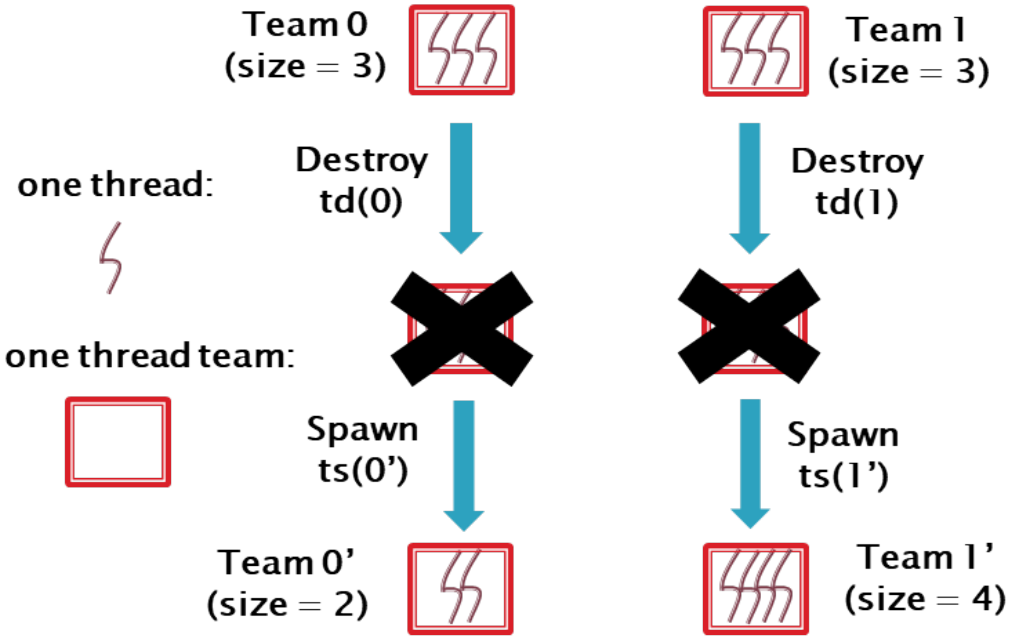


Figure 3.6: Migrating one thread from team 0 to team 1 is performed by destroying teams 0 and 1, and then spawning new teams 0' and 1'.

bound of performance improvement, this chapter assumes an oracle algorithm that can achieve a perfect thread assignment in a single step. Furthermore, this chapter assumes that the procedure of a parallel execution can be divided into three steps, as shown in Figure 3.5. The first step is to start creating thread teams, the second step is to start parallel execution, and the third step is to start destroying thread teams. Last but not least, this chapter assumes that the time of spawning threads in a thread team is proportional to the number of threads, which indicates that spawning more threads costs more time.

Figure 3.6 illustrates the migration of one thread from one thread team to another. In this figure, two thread teams, teams 0 and 1, are considered. Initially, teams 0 and 1 both consist of three threads. To migrate one thread from team 0 to team 1, both teams are

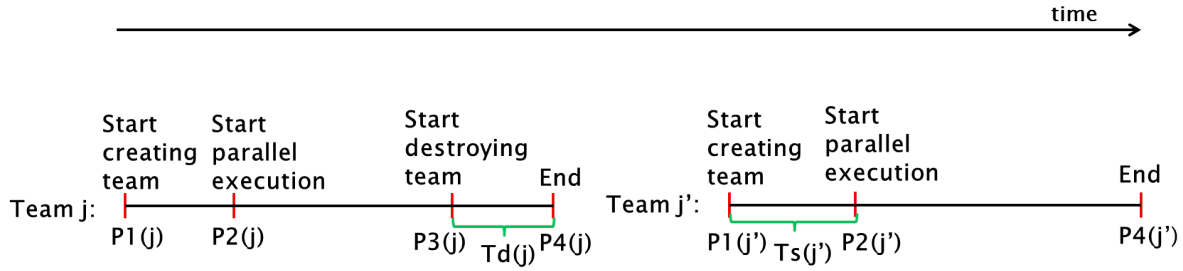


Figure 3.7: Execution procedure of teams j and j'. Team j has a step of destroying the team while team j' does not have such a step.

first destroyed where the time of destroying team j is denoted as $td(j)$, and then teams 0' and 1' are spawned where the time of spawning team j' is denoted as $ts(j')$. Note that an oracle algorithm of dynamic thread team size adjustment is assumed. Therefore, team j' has the best thread team size.

For N phases, the overall runtime overhead is estimated by summing up the runtime overhead of each phase. For a specific phase, the runtime overhead is estimated by choosing the maximum overhead of each thread team. For a specific thread team, the runtime overhead is estimated by summing up the time of destroying the current team and the time of spawning a new team. Eq. (3.3) shows the overall runtime overhead of N phases, and each phase is executed using M thread teams.

$$overhead = \sum_{i=1}^{N} \max_{j,j'} \left(td_i(j) + ts_i(j')\right). \tag{3.3}$$

Here, $td_i(j)$ and $ts_i(j')$ represent the time of destroying team j in phase i and the time of spawning new team j' in phase i, respectively.

During the execution of an irregular application, some threads finish their executions earlier than other threads. As a result, more and more threads become idle in a certain thread team as the execution proceeds. A challenge is to decide when to destroy a thread team. When a thread team is destroyed, all the threads in the team forcefully exit a parallel region and join the master thread of the team. However, destroying active threads requires to unwind the stacks of the threads [63] and deallocate the memories [64]. This is why destroying active threads is more expensive than destroying idle threads.

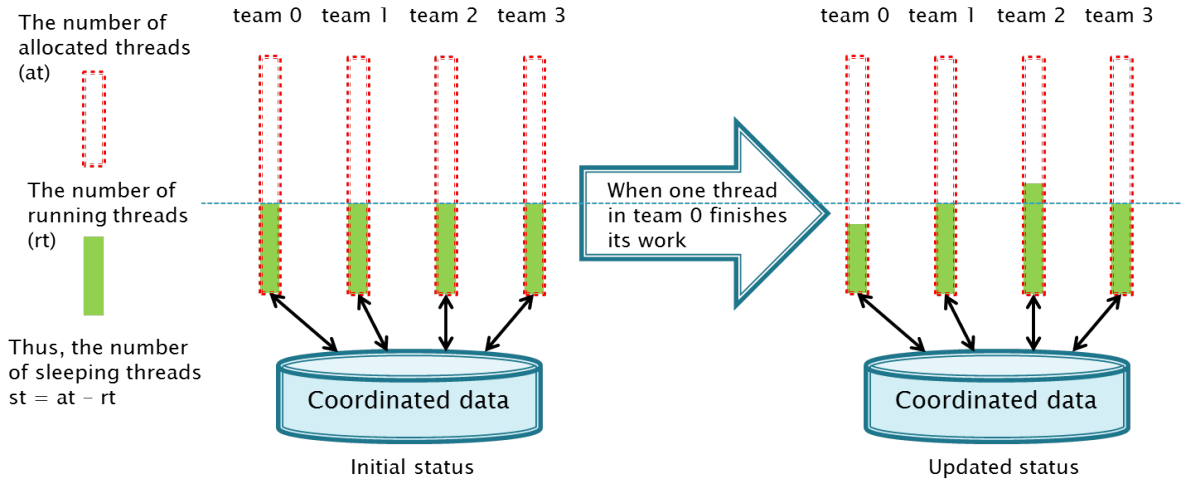


Figure 3.8: The idea of implementation.

Note that thread migration occurs when at least one thread finishes its work and becomes idle. In this approach, all teams are destroyed when the fastest thread in team j is found idle. This ensures that the time of destroying team j is the longest because as more threads in team j become idle, the time of destroying team j becomes smaller. Therefore, the time for destroying a thread team is expected to be the longest if the team is destroyed when the fastest thread in the team becomes idle. As a result, one can estimate the upper bound of the runtime overhead for dynamic thread team size adjustment.

Figure 3.7 shows the execution procedure of teams j and j'. The time stamp of the i-th ($i = 1, 3$) step is recorded as Pi. Moreover, the time stamp is recorded as $P4$ at the end of a parallel execution. Accordingly, the $td(j)$ and $ts(j')$ can be obtained by using Eq. (3.4).

$$\begin{aligned} td(j) &= P4(j) - P3(j), \\ ts(j') &= P2(j') - P1(j'). \end{aligned} \tag{3.4}$$

In addition, since $ts(j')$ is proportional to the number of threads in team j' for a specific application, one can estimate $ts(j')$ of different thread team sizes.

3.4.3 The idea of implementation

The idea of implementation is described as follows. When one thread in one team finishes its work, it is put into sleep, and another thread in another team wakes up and becomes

active. As Figure 3.8 illustrates, initially, each thread team has the same number of running threads and sleeping threads. Each team is allocated to have the maximum number of threads of a given system, since each team has the possibility to reach the maximum number of threads. Some allocated threads are put into sleep. Each sleeping thread will automatically wake up when a certain time period has elapsed. All teams have their own data for thread management. The data are called the coordinated data that are stored in a temporary memory. Subsequently, if one thread in a team finishes its work, it updates a counter of available threads in the thread management data, and also it is put into sleep. One sleeping thread in another team wakes up and checks if the counter is larger than 0. In this case, the sleeping thread becomes active. If the counter is smaller than 0, it is put into sleep again. In Figure 3.8, when one thread in team 0 finishes its work, it updates the thread management data, and it is put into sleep. One sleeping thread in team 2 checks the counter, and becomes active.

In the implementation, a thread can be put into sleep using a function *usleep(t)*. The value of t is a factor to affect the performance. If t is too large, the sleeping threads cannot efficiently wake up when one active thread finishes its work. On the other hand, if the t is too small, the threads frequently access the coordinated data, which may introduce a large overhead. This dissertation assumes that an appropriate value of t is given by users.

3.5 Evaluations

This section shows the evaluation results using the method described in Section 3.4. Since each thread team works independently from the others, the execution time of each team is measured individually when the team size is changed. The OpenMP version 4.5 is used in the evaluations. The OpenMP runtime system uses thread pooling [65, 66] to implement thread destroy and creation. When a thread finishes its work, instead of destroying the thread, it is put into sleep until awaken again. Additional threads will be created when more threads are needed. However, this chapter assumes that the team is totally destroyed. It is more expensive to totally destroy a team than to put it into sleep. This is because destroying a team might need to destroy active threads. Therefore, the overhead of destroying a thread team in this chapter is larger than that in real implementations (e.g., GCC and Intel implementations). Even with considering the larger overhead, dynamic thread team size adjustment can still improve performance as shown later in the evaluation results.

3.5.1 Experimental setup

Three target applications are used for evaluation. They are ray tracing, SoE, and n-body applications, which are the same applications as in Section 2.5.1. The system used for ray tracing and SoE is a KNC system shown in Table 3.2. The system used for n-body is a KNL (Knights Landing) system shown in Table 3.3. Note that the KNC system has maximum of 240 threads, while the KNL system has maximum of 288 threads. The ray tracing and SoE applications are executed using four thread teams, and the n-body application is executed using either two or four thread teams. Eight phases are assumed in the ray tracing code, and each phase renders an image of 256×256 pixels. On the other hand, five phases are created in the SoE, and each phase calculates 1/5 of the problem size of 1.0×10^7. For the n-body application, some parameters are shown in Table 3.4. A typical dataset of Dubinski is used in n-body application, and the number of particles is $81,920$. 316 phases are created. In each phase, an octree is constructed and the particles

Table 3.2: The KNC system.

OS	CentOS Release 6.7
Host CPU	Intel Xeon CPU E5-2690
Accelerator	Intel Xeon Phi 5110P, 240 threads maximum
Compilation	Intel compiler version 16.0.2 with -O3 option

Table 3.3: The KNL system.

OS	CentOS Release 7.4.1708
Processor	Intel Xeon Phi CPU 7290, 288 threads maximum
Compilation	Intel compiler version 17.0.4 with -O3 option

in the octree are updated.

In the evaluation, the number of threads in each team can be different. It can be designated using a *num_threads* clause in a *parallel* construct. The code in Figure 3.9 shows an example to change the thread team size and individually measure the execution time of each team with a different team size.

In the evaluation, the way to adjust a thread team size vector is described as follows. In the case of 4-team execution, for a given thread team size vector $v = (s_0, s_1, s_2, s_3)$, the value of s_i ($i = 0, 1, 2, 3$) is changed while $(s_0 + s_1 + s_2 + s_3)$ is accumulated to the maximum number of threads in KNC or KNL system. The value of s_i changes from 20 to 180 for ray tracing, from 30 to 120 for SoE, and from 40 to 136 for n-body. The granularity of changing s_i is 20, 10, and 4, for ray tracing, SoE, and n-body, respectively. In this way, one can adjust a thread team size vector, and explore as many vectors as possible to find the best vector in each phase.

3.5.2 Performance gain by dynamic thread team size adjustment

Figures 3.10 and 3.11 show the evaluation results of the ray tracing and SoE, respectively. The figures show the execution times of each team for different phases when the team size is changed. The horizontal axis represents the number of threads in a team, and the vertical axis represents the execution time. Different colors represent different phases.

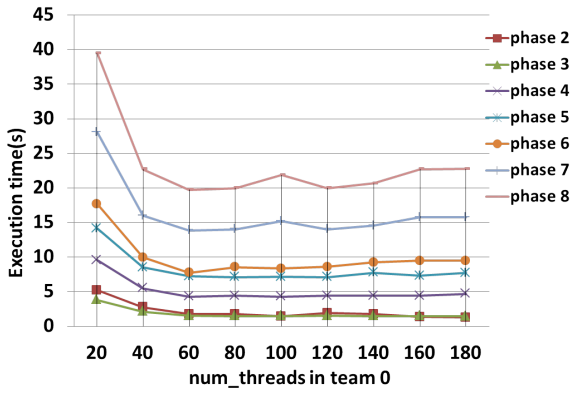
Table 3.4: Some parameters used in n-body code.

TIME_LIMIT	TIME_STEP	Dataset	The number of particles	Phases
316	1	Dubinski	81,920	316

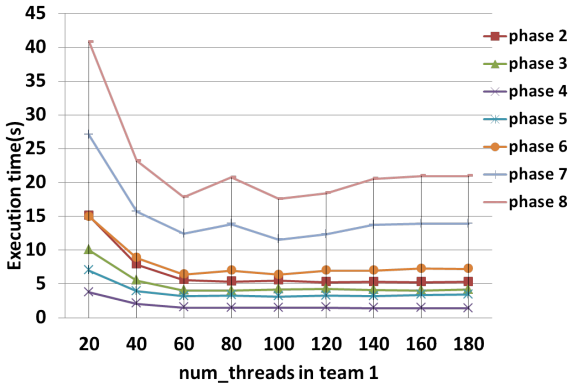
```
#pragma omp target device(0)
#pragma omp teams num_teams(4)
{
 ......
 if(omp_get_team_num() == 0){
   double t00=omp_get_wtime();
   omp_set_dynamic(0);
   #pragma omp parallel for num_threads(100)
   /*For Loop to be measured*/
   double t01=omp_get_wtime();
   double t0 = t01-t00;
 }
 else if(omp_get_team_num() == 1){
    double t10=omp_get_wtime();
    omp_set_dynamic(0);
    #pragma omp parallel for num_threads(80)
    /*For Loop to be measured*/
    double t11=omp_get_wtime();
    double t1 = t11-t10;
 }
  .....
 }// end omp target
```

Figure 3.9: An example to change team size and measure execution time of each phase. The clause *num_teams* is used to assign the number of teams. The routine *omp_get_team_num()* is used to designate the team number. The clause *num_threads* is used to designate the team size.

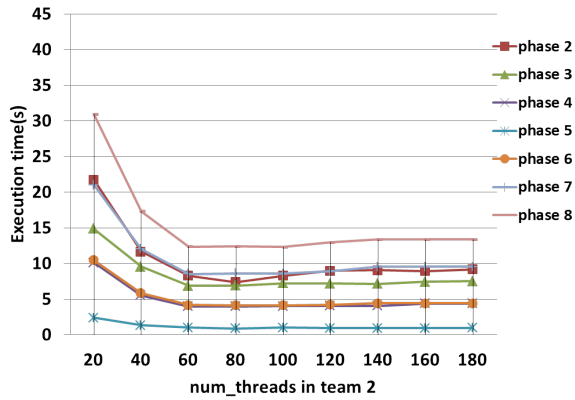
The results in Figures 3.10 and 3.11 show that the execution time varies from phase to phase with a fixed team size. It is because the target applications are irregular. For a particular phase, the execution time changes with the team size. This is because a different team size leads to different thread-level parallelism. However, the execution times of some phases (e.g., phase 4 in Figure 3.10(b)) are almost unchanged even if the team sizes are changed. This is because ray tracing and SoE are not computation-intensive applications so that changing the team size does not have significant impact on performance. In Figures 3.10 and 3.11, the maximum team size does not always achieve



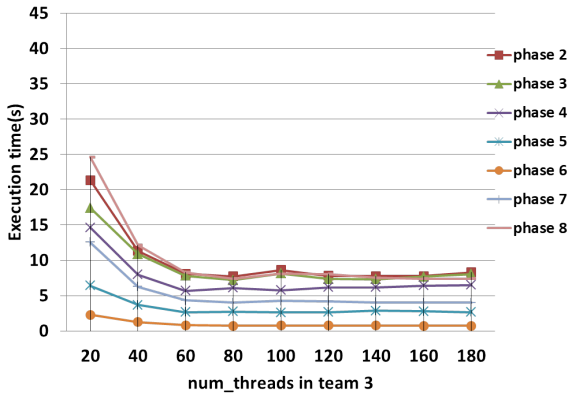
(a) The execution times of team *0* for different phases.



(b) The execution times of team *1* for different phases.

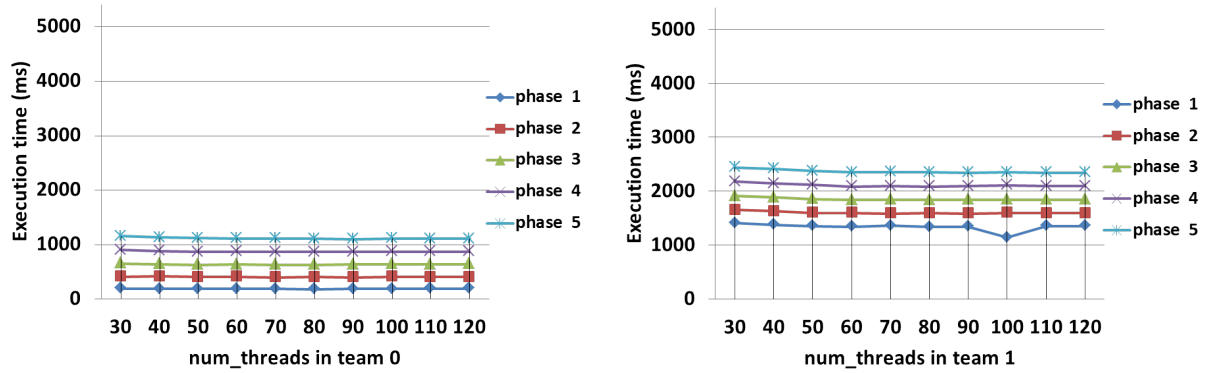


(c) The execution times of team *2* for different phases.

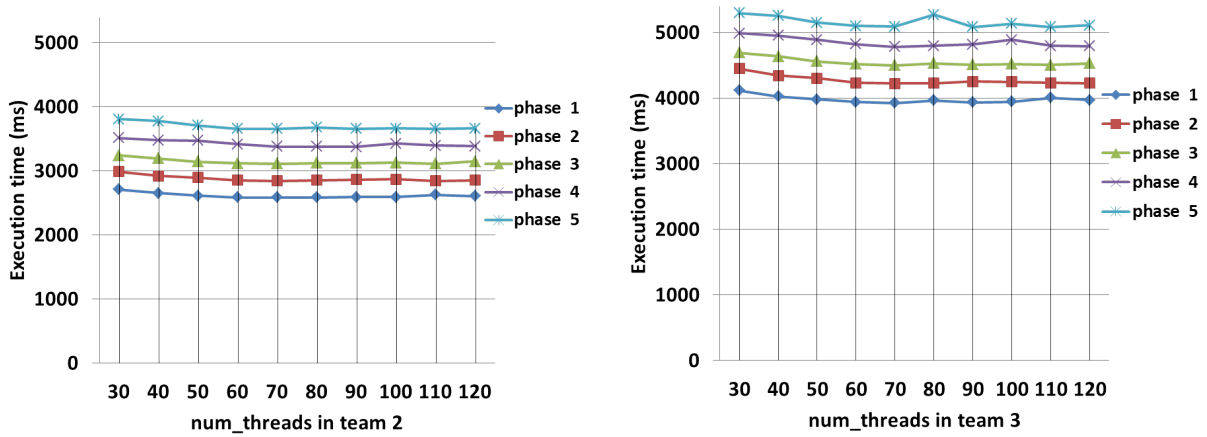


(d) The execution times of team *3* for different phases.

Figure 3.10: The execution times of each team for the ray tracing code, when the team size is changed.



(a) The execution times of team *0* for different phases.(b) The execution times of team *1* for different phases.



(c) The execution times of team *2* for different phases.(d) The execution times of team *3* for different phases.

Figure 3.11: The execution times of each team for the SoE code, when the team size is changed. Five phases are created.

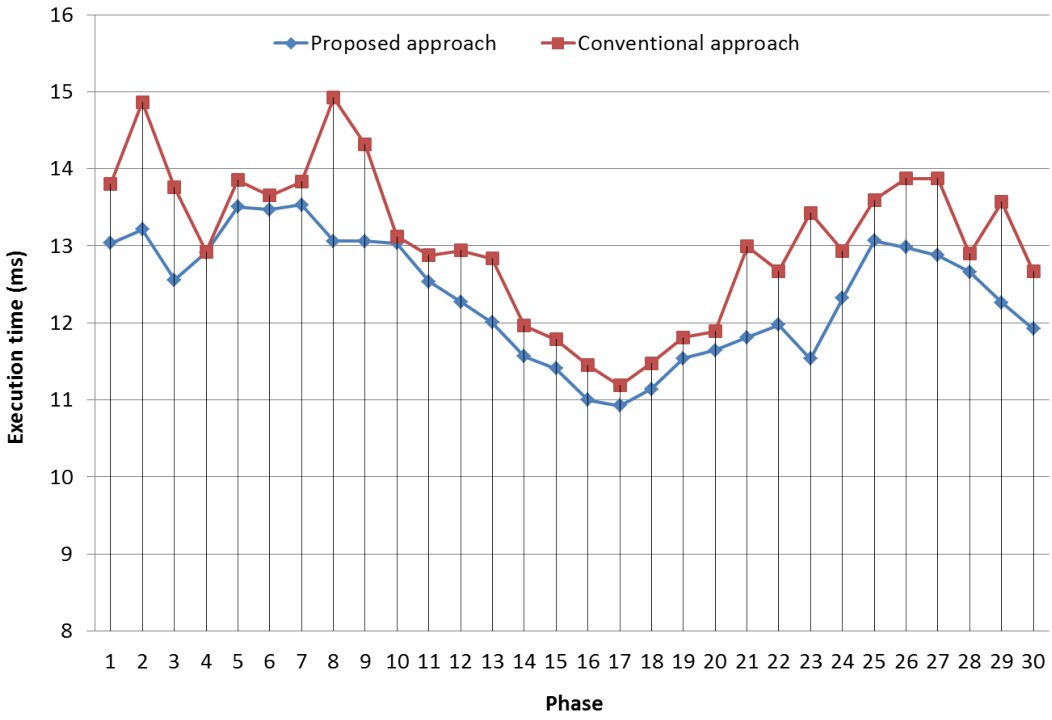


Figure 3.12: The execution times of 30 phases for n-body.

the best performance for most phases. One important point seen in the results is that the best team size changes from team to team for a particular phase. Therefore, the thread team size needs to be adjusted individually for each team.

For n-body code, execution times of 30 phases are shown in Figure 3.12 in the case of 4-team execution. The performance of proposed approach is obtained when the best team size vector is applied in each phase. No runtime overhead is included in the performance of proposed approach. The conventional approach is to divide the maximum number of threads evenly and assign the same number of threads to each team. In Figure 3.12, the horizontal axis represents the phase number, which is from phase 1 to phase 30, the vertical axis represents the execution time in milliseconds. The results show that each phase has a different execution time. It is because the n-body code is irregular. The results also show that the proposed approach is equivalent to or better than the conventional approach for all phases.

By using the method described in Section 3.4, the overall execution time of dynamic thread team size adjustment can be estimated. Figure 3.13 shows the performance of

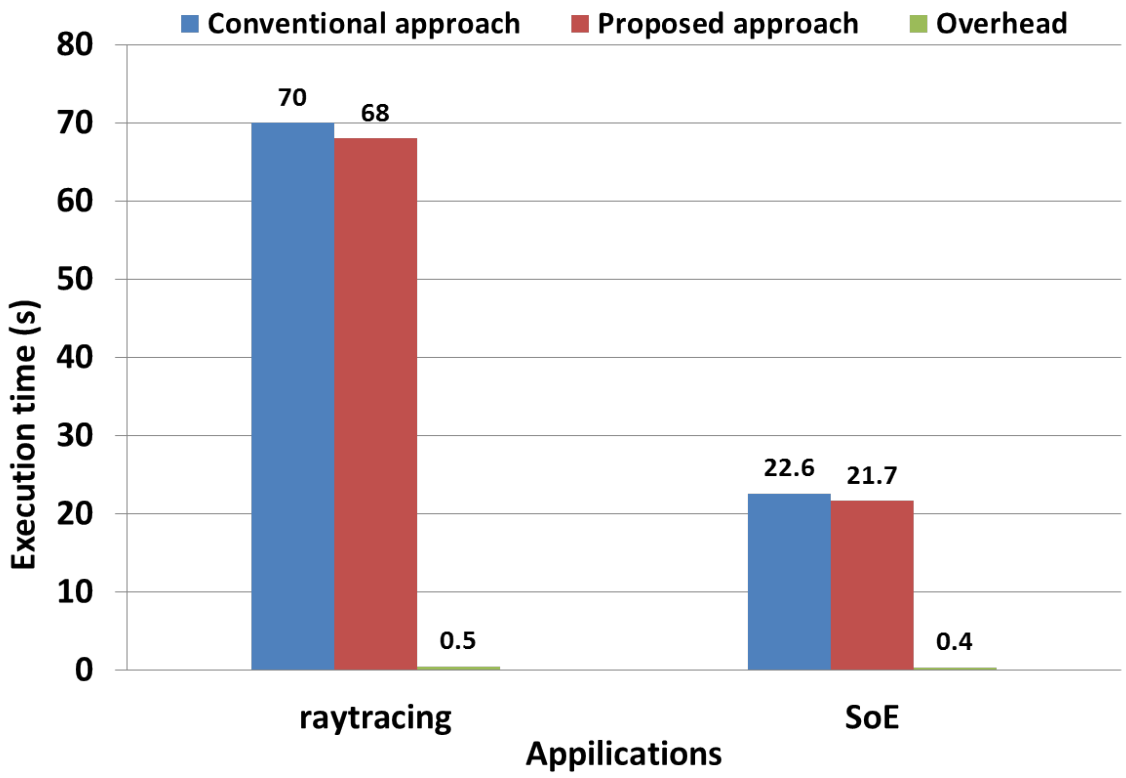


Figure 3.13: Comparison of conventional approach, proposed approach, and runtime overhead, for ray tracing and SoE.

proposed approach against the conventional approach, for ray tracing and SoE. For the ray tracing code, dynamic thread team size adjustment improves performance by about 3%. For the SoE code, the performance improvement is about 4%. The performance improvements are not significant because both ray tracing and SoE are memory-bound applications [67], and thus changing the number of threads does not significantly affects the performance.

Figure 3.14 shows the performance of proposed approach against the conventional approach for n-body code. In the case of 2-team execution, the performance improvement is about 20.5% excluding the runtime overhead. In the case of 4-team execution, the performance improvement is about 23.5% excluding the runtime overhead. The performance gain of n-body is significant because adjusting the number of threads significantly affects the performance.

After team size adjustment, the best team size vectors are shown in Tables 3.5, 3.6, and 3.7, for ray tracing, SoE, and n-body, respectively. The tables show that the number

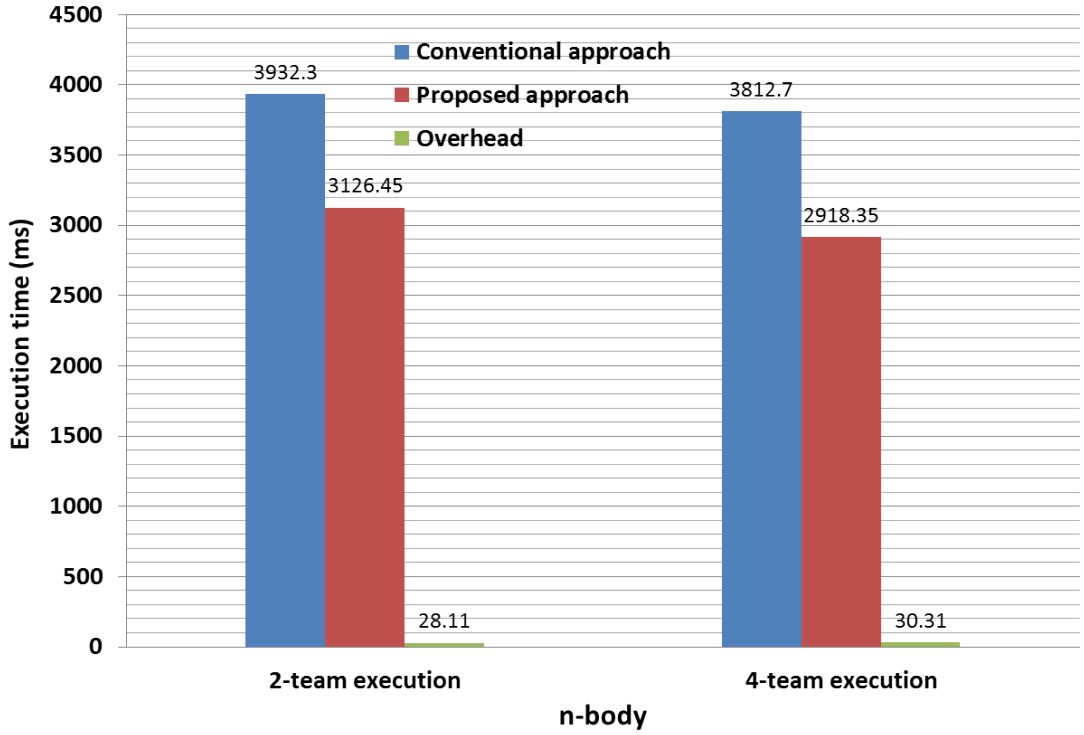


Figure 3.14: Comparison of conventional approach, proposed approach, and runtime overhead, for n-body.

of threads in each team can be different after team size adjustment, which is different from the standard OpenMP specification. Therefore, the OpenMP specification should support a different thread team size in each team in the case of using multiple teams, in order to balance the execution times of teams.

3.5.3 Runtime overhead

This subsection presents the results of estimated runtime overhead of dynamic thread team size adjustment by using the method described in Section 3.4.2. To destroy a thread team, the OpenMP *cancellation* constructs are used.

The runtime overheads of ray tracing and SoE are shown in Figure 3.13, along with the performances of proposed approach and conventional approach. The runtime overheads of ray tracing and SoE are 0.5 seconds and 0.4 seconds, respectively. The runtime overhead of n-body is shown in Figure 3.14. In the case of 2-team execution, the overhead is 28 milliseconds. In the case of 4-team execution, the overhead is 30 milliseconds. The

Table 3.5: The best thread team size vector of the ray tracing for all phases.

	Thread team size vector
Phase 2	(20,60,80,80)
Phase 3	(40,40,60,80)
Phase 4	(60,40,60,60)
Phase 5	(120,40,20,60)
Phase 6	(60,60,40,40)
Phase 7	(60,60,40,40)
Phase 8	(60,100,40,40)

Table 3.6: The best thread team size vector of the SoE for all phases.

	Thread team size vector
Phase 1	(70,40,60,70)
Phase 2	(50,70,50,70)
Phase 3	(50,50,70,70)
Phase 4	(40,50,80,70)
Phase 5	(40,90,40,70)

overhead of n-body is obviously smaller than that of ray tracing and SoE, because the n-body code is run on the KNL system but not on the KNC system. The results demonstrate that the overhead can be negligible in comparison with the performance of proposed approach, for the three target applications.

The evaluation results show that, even with overestimating the runtime overhead, dynamic team size adjustment can improve performance of irregular applications. However, the performance improvement is small in the case of memory-bound applications.

Table 3.7: The best thread team size vector of the n-body for 10 phases in the case of 4-team execution.

	Thread team size vector
Phase 1	(64,72,72,80)
Phase 2	(60,80,80,68)
Phase 3	(68,72,76,72)
Phase 4	(68,68,80,72)
Phase 5	(76,76,68,68)
Phase 6	(72,76,72,68)
Phase 7	(72,76,72,68)
Phase 8	(68,76,72,72)
Phase 9	(68,68,80,72)
Phase 10	(64,76,76,72)

3.6 Conclusions

The objective of this chapter is to clarify the benefits of dynamic thread team size adjustment on the performance of irregular applications. The motivating examples show that the thread team size needs to be carefully adjusted to achieve a higher performance. In this chapter, the methods to estimate performance gain and runtime overhead of dynamic thread team size adjustment are presented.

The evaluation results show that, even if the runtime overhead is considered, dynamic thread team size adjustment can still improve the performance of irregular applications in comparison with the conventional approach. Moreover, the runtime overhead of thread management is negligible in comparison with the total execution of an application, which implies that migrating OpenMP threads from one thread team to another introduces a low overhead.

In this chapter, an oracle algorithm for dynamic thread team size adjustment is assumed. Such an algorithm can achieve a perfect thread team size adjustment in a single step, which is not practical. Thus, the future work includes research on developing a more practical algorithm for dynamic thread team size adjustment.

Chapter 4

Dynamic Workload Management for Irregular Applications

4.1 Introduction

As in Chapter 3, this chapter assumes that the hotspot of an irregular application is executed on a KNC using multiple thread teams. However, unlike Chapter 3, this chapter assumes that each thread team always has the same number of threads, and the number of threads of each team is unchanged during the execution. For load balancing, an application is parallelized by unevenly dividing the loop length into chunks, each of which is assigned to one thread team for execution. Since load imbalance occurs across thread teams for irregular applications if a loop length is evenly divided for parallelization, this chapter deals with the load imbalance by using workload management, which is to adjust the number of iterations assigned to each thread team.

The current OpenMP specification has already supported both static and dynamic loop scheduling mechanisms that are within a thread team, and also supports a static scheduling mechanism across thread teams. In this chapter, a scheduling mechanism is a workload management mechanism to adjust loop iterations. When multiple thread teams are used to execute an application, it is strongly required to have dynamic workload management across thread teams to handle the load imbalance across teams. Note that

this chapter does not extend the OpenMP programming model itself. However, this chapter extends the way of dividing loop length in the current OpenMP specification. In the current OpenMP specification, the loop length for each team is evenly divided. In this chapter, the loop length for each team is dynamically changed.

This chapter has two general assumptions. First, for a specific application, the overhead of inter-team dynamic scheduling can be estimated using the overhead of intra-team dynamic 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, it is expected 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 [69] 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. In this chapter, 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 task pooling [70, 71] algorithms, which allow a scheduler to assign 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 improve performance. Usually, the overhead of dynamic scheduling decreases as the chunk size increases, as discussed in [69]. 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 further load imbalance. Thus, the chunk size needs to be tuned with considering a trade-off between the effects of workload management and the scheduling overhead.

The objective of this chapter is to clarify the benefits of dynamic workload management across thread teams. In this chapter, a static workload management method is used to emulate a dynamic one. Moreover, the inter-team scheduling overhead is estimated by using the intra-team scheduling overhead.

The contributions of this chapter are twofold. First, this chapter clarifies the benefits of dynamic workload management across thread teams for irregular applications. Second, this chapter discusses the overhead of inter-team dynamic workload management under the aforementioned assumptions.

The rest of this chapter is organized as follows. Section 4.2 describes the related work. Section 4.3 discusses the emulation method of inter-team dynamic workload management. It also discusses a method to measure the overhead of intra-team dynamic workload management. Section 4.4 shows the evaluation results, and Section 4.6 gives concluding remarks.

4.2 Related work

This section describes the related work of dynamic workload management. Unlike the related work in Section 3.2 that mainly focuses on adjusting the number of threads, the related work in this section mainly focuses on scheduling the number of iterations.

4.2.1 Scheduling methods in OpenMP

Currently, the OpenMP specification 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 cannot solve the load imbalance across teams. Thus, this chapter examines the benefits of dynamic scheduling across thread teams.

4.2.2 Co-scheduling for heterogeneous systems

Many researches focus on dynamic workload management for heterogeneous systems. Wang et al. [72] have proposed an asymptotic profiling method to schedule loop iterations between a CPU and a GPU. Ren et al. [73] have proposed a method to test various work distributions between a CPU and a GPU to find the most efficient distribution. Boyer et al. [74] have presented a dynamic workload management mechanism that requires no offline training and responds automatically to performance variability. Other approaches, such as in [75–77], use modest amounts of initial training to distribute computations. Scogland et al. [78] also have proposed a dynamic workload management mechanism across CPUs and GPUs. They consider different computation patterns of the accelerated code region. All those researches only concern about how the computations should be distributed in the heterogeneous systems. They do not care about fine-grained workload management either on CPUs or accelerators. Unlike other researches, this chapter focuses on dynamic scheduling of the iterations across thread teams on an accelerator. Although KNC is used as an example of the accelerator, the dynamic scheduling mechanism is

applicable to other accelerators.

4.2.3 Scheduling for irregular applications

Some related studies use dynamic scheduling for irregular applications. Durand et al. [79] have proposed a new OpenMP loop scheduler that is able to perform dynamic workload management while taking memory affinity into account for irregular applications. Min et al. [80] 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 accelerator constructs for execution on a target accelerator. They mainly schedule the iterations among different threads or different cores. On the other hand, this work uses high-level OpenMP accelerator constructs for offloading the computations to an accelerator. As in Chapter 3, this chapter uses multiple thread teams to execute the codes. However, unlike Chapter 3, this chapter discusses workload management across thread teams instead of thread management.

4.2.4 General-purpose scheduling and domain-specific scheduling

In addition to some centralized scheduling methods [72, 81] for dynamic workload management, work stealing [82–85] is a popular scheduling method. It is proved that work stealing is efficient for scheduling some multi-threaded executions. Prokopec et al. [86] have presented a work-stealing algorithm for irregular data-parallel applications. Their method allows workers to decide the computation distribution in a lock-free, computation-driven manner. However, work stealing requires data synchronization whenever a steal happens. Frequent steals might introduce huge overhead, and thus degrade performance. In addition to general-purpose scheduling methods, some studies have proposed domain-specific scheduling mechanisms for dynamic workload management. Their domains vary from fluid dynamics [87] to linear algebra [88]. None of those works addresses dynamic workload management across OpenMP thread teams.

4.2.5 Overhead in OpenMP

Extensive studies [69, 89–92] have been performed to measure the overheads of OpenMP work-sharing and mutual exclusion directives in the literature. Bull [90] 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. This chapter uses the idea in [90] to measure the intra-team dynamic scheduling overhead. More details are described in Section 4.3.2. Fredrickson et al. [69] 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 [69] have shown that intra-team dynamic scheduling overhead is likely to increase exponentially with the number of threads. Based on the results, this chapter makes a similar assumption that inter-team dynamic scheduling overhead exponentially increases with the number of thread teams.

4.3 Methodology

4.3.1 Emulation of inter-team dynamic workload management

This subsection presents a methodology to emulate a dynamic workload management mechanism across thread teams.

Since inter-team dynamic scheduling has not been supported by the current OpenMP specification, a static workload management mechanism is used to mimic a dynamic one. This chapter assumes that a given application is executed through multiple phases (e.g., N phases, from phase 1 to phase N). A workload ratio rj (j = *0, M-1* and M is the number of thread teams) is defined using Eq. (4.1), where e_j is the number of iterations executed by team j, and e_{all} is the total number of iterations executed by all teams.

$$r_j = \frac{e_j}{e_{all}}. \tag{4.1}$$

A workload vector v is defined as a vector of workload ratios, as shown in Eq. (4.2).

$$v = (r_0, r_1, ..., r_{M-1}). \tag{4.2}$$

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 dynamic adjustment of the workload vector. Thus, with a fixed workload vector, the application is executed from beginning to end while the execution time of each phase is measured. The execution is repeated while changing the workload vector. As a result, it is possible to 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, it is possible to obtain the upper bound of performance gain due to dynamic workload management 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 upper bound of performance gain d of an application using

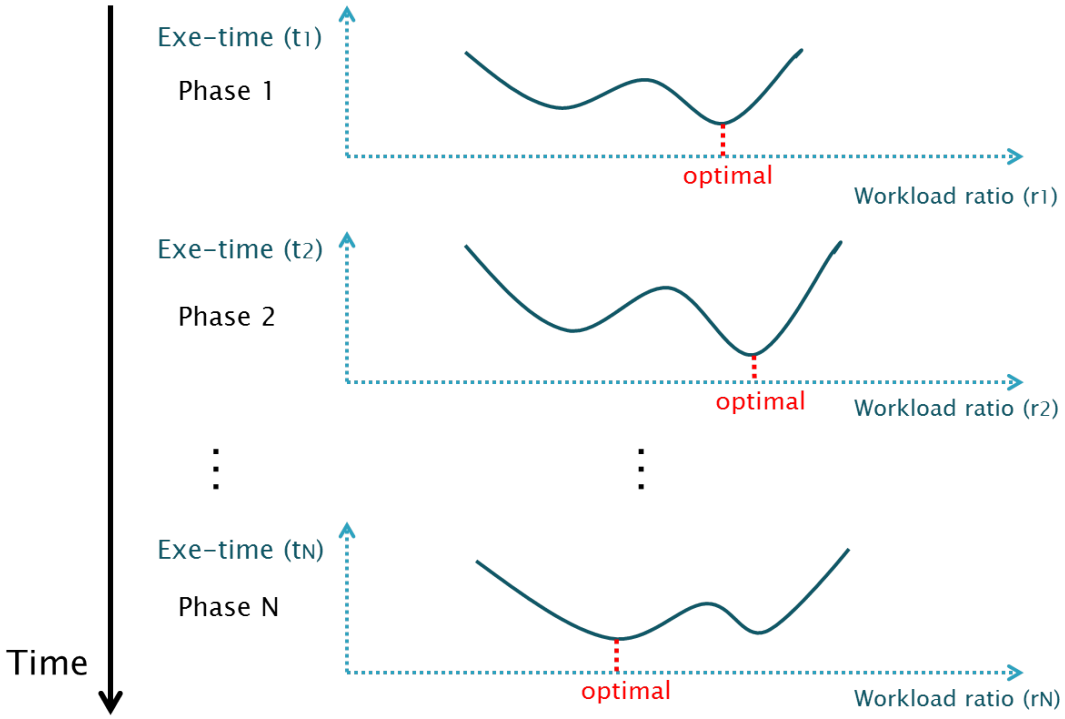


Figure 4.1: Illustration of estimating the upper bound of performance gain due to dynamic workload management.

dynamic workload management can be expressed using Eq. (4.3).

$$d = \sum_{i=1}^{N} \min_{v} t_i(v). \tag{4.3}$$

To put it simple, Figure 4.1 illustrates the idea of estimating the upper bound of performance gain due to dynamic workload management. Suppose that an irregular application has N phases. In each phase, the execution time changes with the workload ratio. By exploring as many workload ratios as possible, it is possible to find the optimal workload ratio that leads to the shortest execution time in each phase. The shortest execution times of all phases are accumulated to estimate the upper bound of performance gain due to dynamic workload management.

Although a dynamic workload management mechanism has not been implemented for the OpenMP programming model, Eq. (4.3) can be used to estimate the benefits of introducing such a mechanism. Based on the estimation, it is possible to discuss whether the performance gain could be larger than the scheduling 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 4.2: Loop (nest) *L1* parallelized by OpenMP directives.

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

Figure 4.3: Loop (nest) *L2* executed using a single thread.

4.3.2 Scheduling overhead

In this chapter, two assumptions have already been discussed in Section 4.1. Based on the first assumption, the overhead of inter-team dynamic scheduling for two-team execution is estimated using the overhead 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. (4.3) with the estimated scheduling overhead, it is possible to discuss the performance gain of inter-team dynamic workload management with runtime overheads.

A method to evaluate the overhead of intra-team dynamic workload management is described in [90]. This chapter employs the same method to evaluate the intra-team dynamic scheduling overhead. The method in [90] 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 Figure 4.2. A serialized version of the same loop (nest) is also assumed, denoted as *Loop (nest) L2*, as shown in Figure 4.3. The goal is to measure the scheduling overhead of Loop *L1*. First, the execution time *t1* of Loop *L1* can be measured. Then, the execution time *t2* of Loop *L2* using a single thread can be measured. Finally, the scheduling overhead t is given by Eq. (4.4).

$$t = \frac{|t1 - t2|}{repeats}. \tag{4.4}$$

If the loop body shown in Figures 4.2 and 4.3 is replaced with the ray tracing loop nest, the intra-team scheduling overhead of the ray tracing code can be measured. The scheduling type can be designated as *dynamic*. To find the upper bound of the overhead, the chunk size is assigned as *1*. Note that the intra-team scheduling overhead can be used to estimate the inter-team scheduling overhead.

4.4 Evaluations

4.4.1 Experimental setup

Three target applications are used for evaluation. They are ray tracing, SoE, and n-body applications, which are the same applications as in Section 2.5.1. The system used for evaluations is a KNC system shown in Table 3.2. The ray tracing is executed using two teams as well as four teams, the SoE and n-body are executed using four teams. Eight phases are used in the ray tracing code, and each phase renders an image of 256×256 pixels. On the other hand, five phases are created in the SoE, and each phase calculates 1/5 of the problem size of 1.0×10^7. A typical dataset of Dubinski is used in n-body application, and the number of particles is 81920. 316 phases are created in the n-body code. In each phase, an octree is constructed and the particles in the octree are updated. In the evaluation, the overall thread count is maintained as the maximum number of threads regardless of how many teams are created, and each team has the same number of threads. However, the workload ratio of each team ranges from 5% to 95%, and increases by 5%.

The performances of six different mechanisms shown in Table 4.1 are evaluated. 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

Table 4.1: Six mechanisms.

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

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 mechanisms, SS, SD, MSS, and MSD have already been supported in the current OpenMP specification. Their performances can be evaluated in a straightforward way. The intra-team scheduling method provided by OpenMP is employed for SD and MSD. However, MDS and MDD have not been supported by OpenMP. Thus, the methodology described in Section 4.3 is used to estimate the performances of MDS and MDD. In the evaluation, the performances of SD, MSD and MDD already include intra-team scheduling overheads. However, the performance of MDD does not include inter-team scheduling overhead.

4.4.2 Performance gain of inter-team dynamic workload management

4.4.2.1 Performance results of ray tracing

In the evaluation, by changing the camera position and direction from phase to phase, a movie of several images can be created. Two movies are created by moving the camera in two different ways. MDS-1 and MDD-1 are used to represent the two mechanisms for the first movie, while MDS-2 and MDD-2 are used for the second movie. For each movie, the iteration space of loop y is statically divided into M (M is the number of teams) parts. Each part is executed by a corresponding thread team. An example is shown in Figure 4.4 when the number of teams $M = 2$. By varying the workload vector, the execution time with each vector is measured.

The evaluation results using two teams are presented as follows. Table 4.2 lists the

```
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++){
        samps = change_sampling_rate_here (t, y);
        for (int x=0; x<w; x++){
          for (int sy=0, i=(h-y-1)*w+x; sy<2; sy++){
            for (int sx=0; sx<2; sx++, r=Vec()){
              for (int s=0; s<samps; s++){
                 /*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++){
        samps = change_sampling_rate_here (t, y);
        for (int x=0; x<w; x++){
          for (int sy=0, i=(h-y-1)*w+x; sy<2; sy++){
            for (int sx=0; sx<2; sx++, r=Vec()){
              for (int s=0; s<samps; s++){
                 /*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 4.4: Statically dividing the iteration space of loop y when M = 2.

best workload vectors in each phase of MDS and MDD, for both movies. In Table 4.2, the best workload vector in each phase is different from (0.5, 0.5), except in Phase 8. It means that load imbalance exists in almost all the phases. Figures 4.5 and 4.6 show the execution times for phases in various mechanisms. Figures 4.5 and 4.6 show the results for the first movie and second movies, respectively. In these figures, different curves represent different mechanisms. MDS and MDD can be either optimal or suboptimal. The optimal MDS

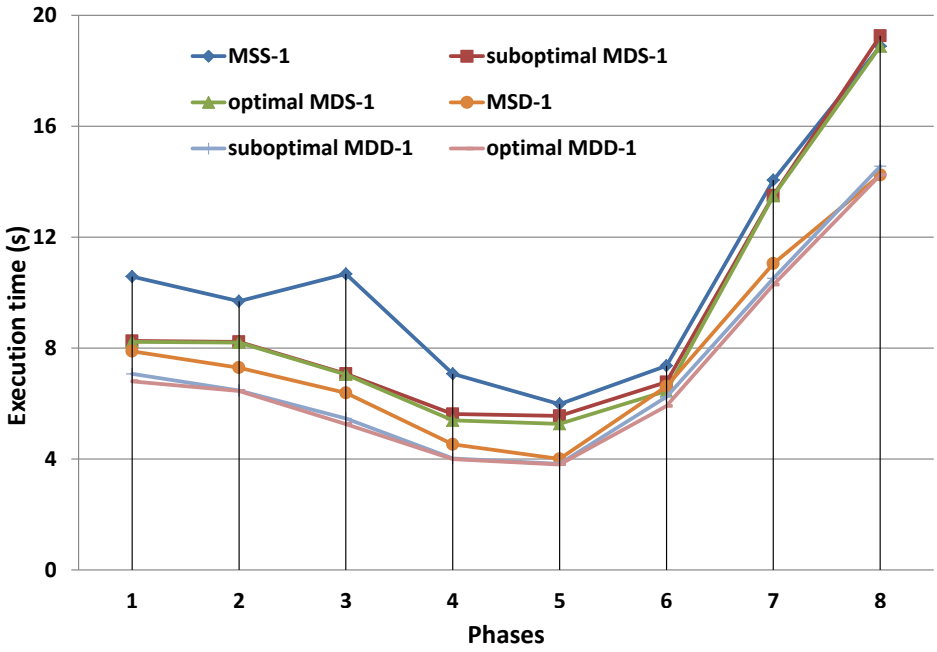


Figure 4.5: The execution times for phases of the first movie. Two teams are used for the execution. MDS and MDD can be either optimal or suboptimal. The optimal MDS means that the best workload ratio in each phase is used, and the suboptimal MDS means that the second-best workload ratio in each phase is used to discuss the performance gain with imperfect prediction.

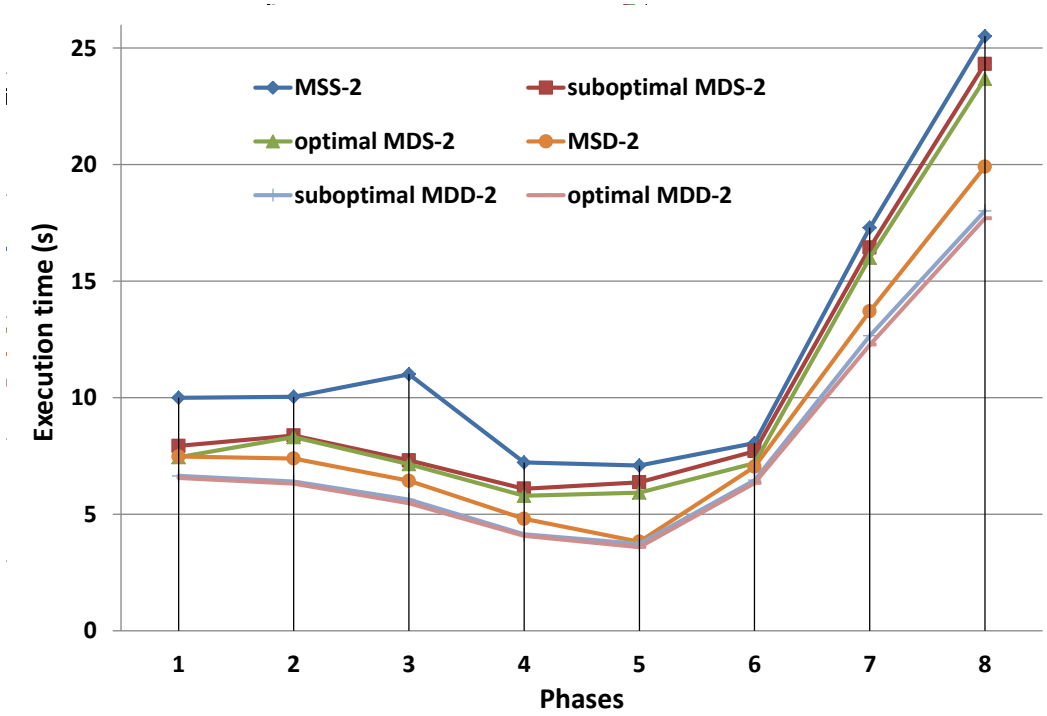


Figure 4.6: The execution times in each phase of various mechanisms for the second movie. Two teams are used for the execution.

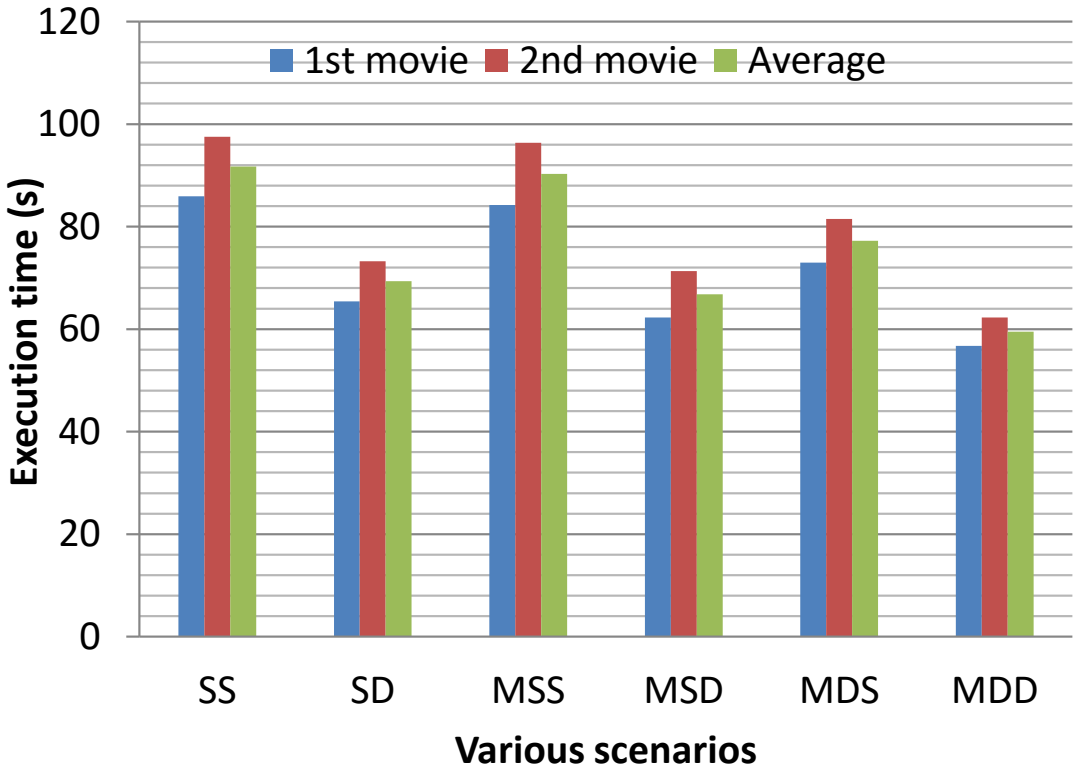


Figure 4.7: Performance comparison of six mechanisms for both movies when two teams are used for execution.

(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 Figures 4.5 and 4.6, execution times vary from phase to phase for a specific mechanism. This is because the ray tracing code is an irregular application. Moreover, the optimal MDD is the best mechanism among the six mechanisms, for both movies. In the rest of this chapter, unless otherwise stated, MDS (MDD) represents the optimal MDS (MDD).

Figure 4.7 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. Figure 4.7 shows 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 MSD. Thus, dynamic

Table 4.2: 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)

scheduling across thread teams improves performance in comparison with static scheduling when two teams are used for execution. This is because the execution times across thread teams are more balanced.

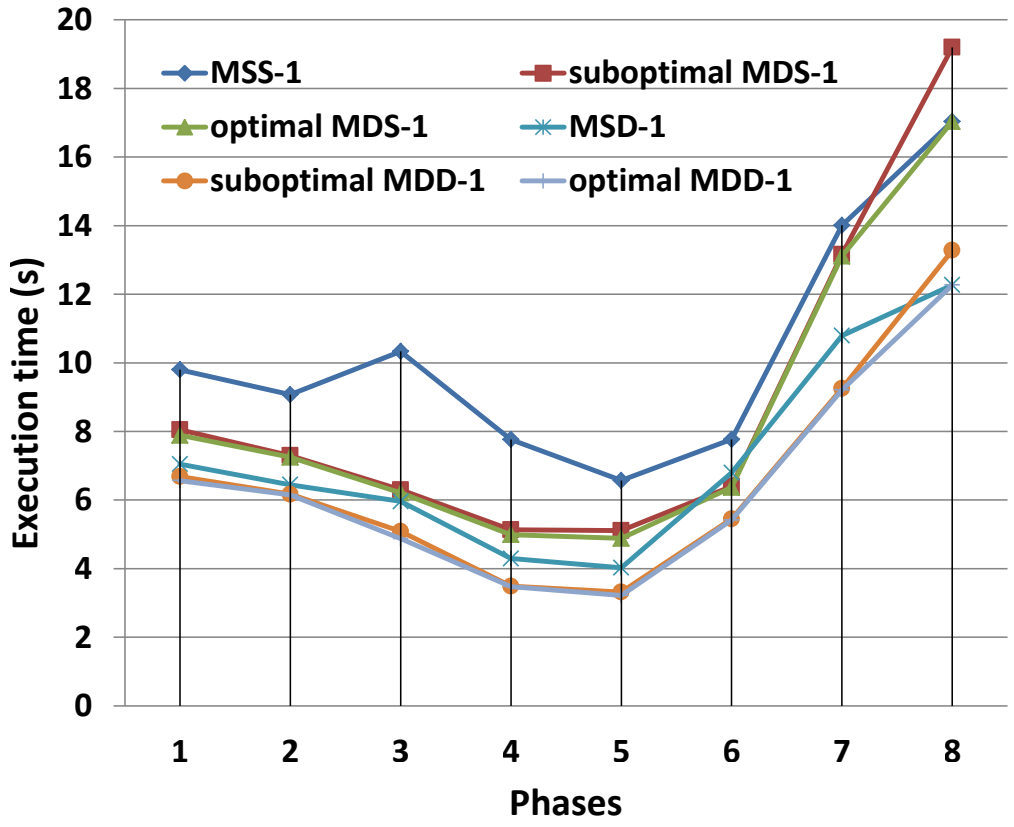


Figure 4.8: The execution times in each phase of various mechanisms for the first movie. Four teams are used for the execution.

The evaluation results using four teams are presented as follows. The best workload vector in each phase is summarized in Table 4.3, for both movies. Figures 4.8 and 4.9

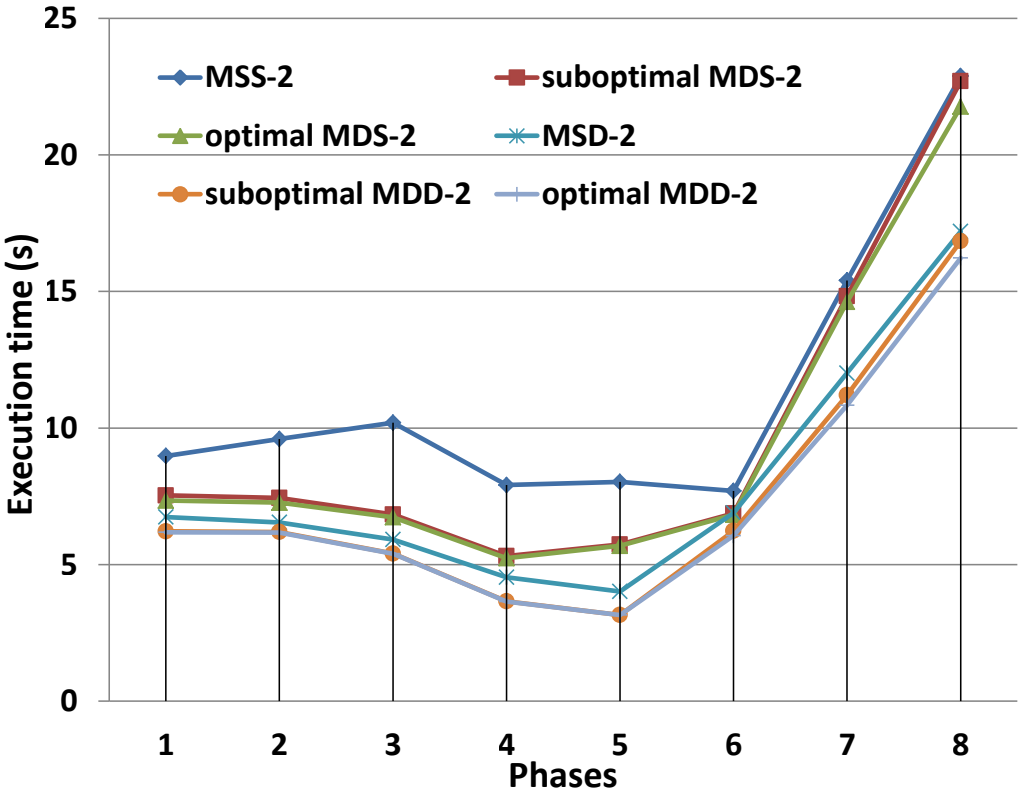


Figure 4.9: The execution times in each phase of various mechanisms for the second movie. Four teams are used for the execution.

show the execution times of phases for the first movie and second movie, respectively.

Figure 4.10 shows the total execution time of each mechanism for both movies when four teams are used for execution. The average execution times of both movies are also shown. Figure 4.10 shows 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.

Table 4.3: The best workload vectors in each phase for both movies when four teams are used for execution.

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

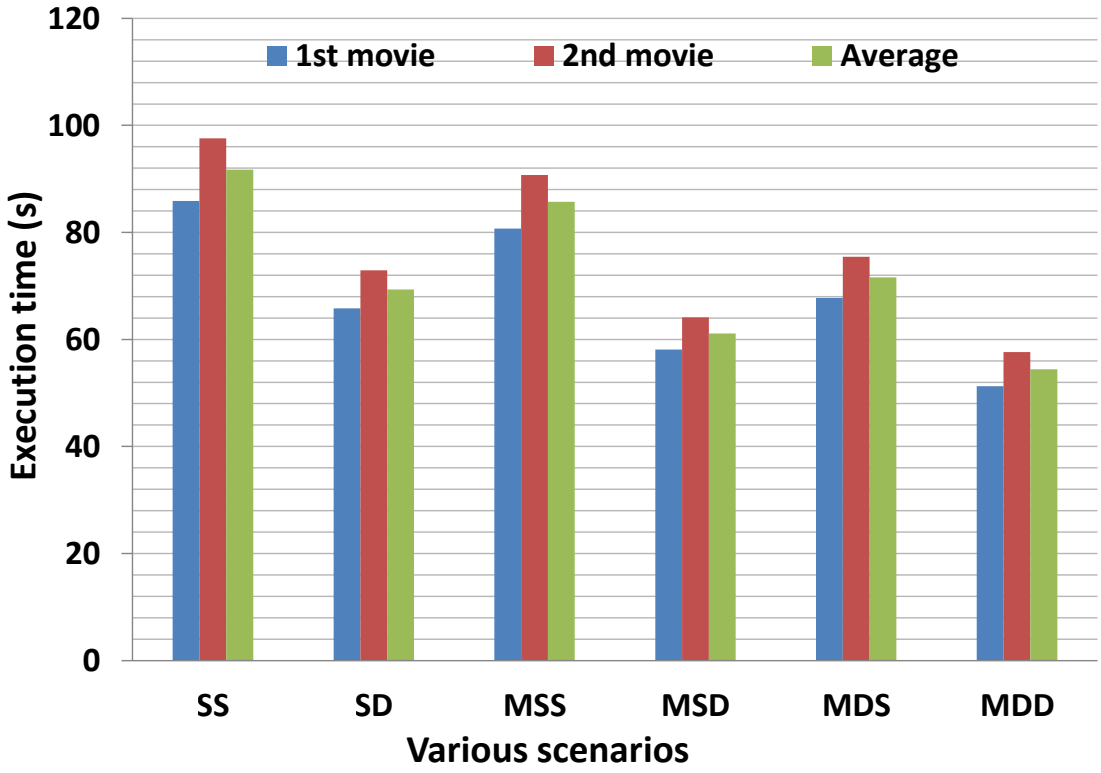


Figure 4.10: Performance comparison of six mechanisms for both movies when four teams are used for execution.

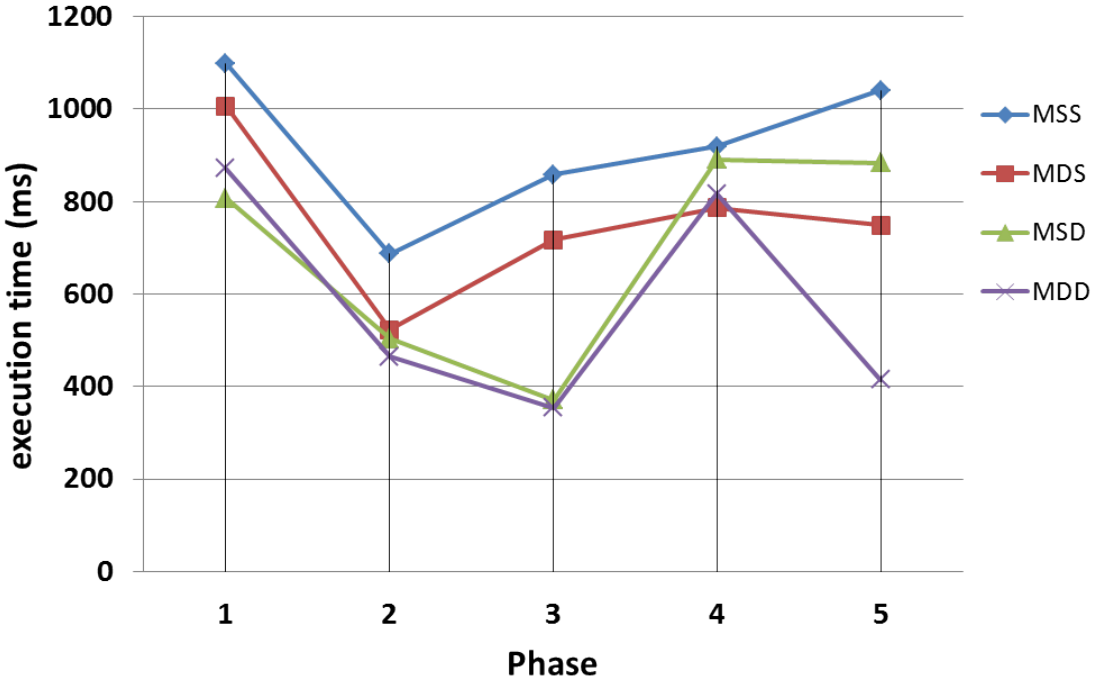


Figure 4.11: Execution time of each phase for different mechanisms. The code is SoE.

4.4.2.2 Performance results of SoE

In the evaluation, each phase is executed by four thread teams, while the iterations assigned to each thread team are changed using the dynamic workload management method described in Section 4.3.1. Figure 4.11 shows the execution time of each phase for different mechanisms. In Figure 4.11, the horizontal axis represents different phases, from phase *1* to phase *5*. The vertical axis represents the execution time in milliseconds. Different curves represent different mechanisms. Figure 4.11 shows that, for a particular mechanism, execution time varies from phase to phase. This is because SoE is an irregular application. Moreover, different mechanisms lead to different performances for a particular phase because different scheduling methods are used. Figure 4.12 shows the performances of six mechanisms. Figure 4.12 shows that MDS outperforms MSS by 17.8%, and MDD outperforms MSD by 15.3%. MDD is the best mechanism among all the mechanisms. Therefore, dynamic scheduling across teams is crucial to balance the execution times across teams.

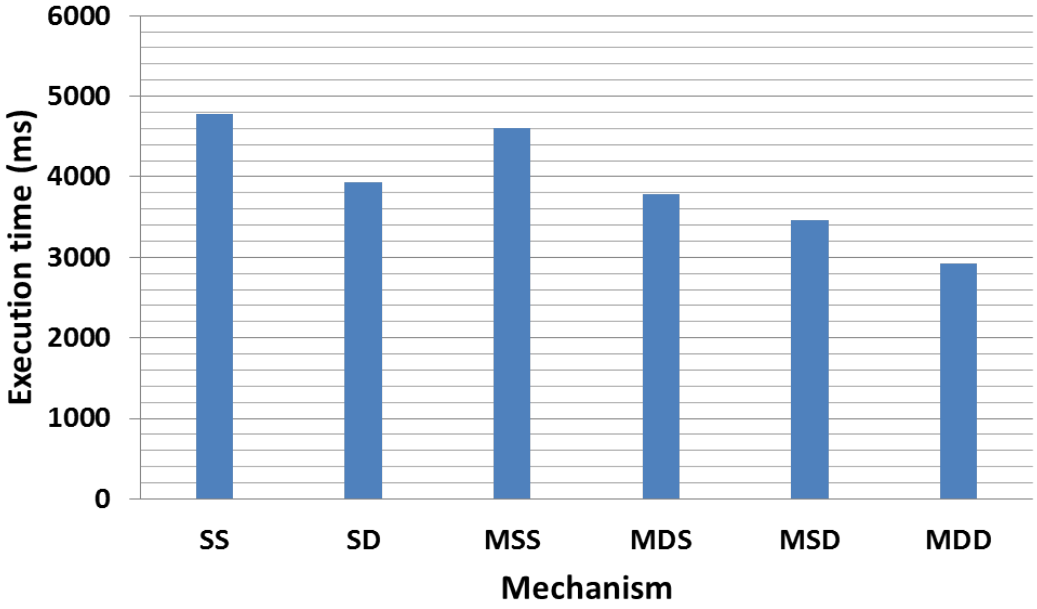


Figure 4.12: Performances of six mechanisms. The code is SoE.

4.4.2.3 Performance results of n-body

Figure 4.13 shows the execution time of each phase except phase 1 for different mechanisms. Four thread teams are used. The execution time of phase 1 is much longer than those of other phases because it includes offloading overhead, and thus the execution time of phase 1 is not shown in Figure 4.13. The horizontal axis represents the phase number, which is from 2 to 316. Different curves represent different mechanisms. In Figure 4.13, it is surprising to find that MSS performs better than MDD in phases 2 to 25 while MDD still performs better than MSD in those phases. Hence, the overhead of intra-team dynamic scheduling is so large that it kills performance gain in phases 2 to 25.

Figure 4.14 shows the standard deviation of execution times of phase 60 to 316, for various mechanisms. The execution time of phases 60 to 316 occupies more than 70% of overall execution time of a particular mechanism. Thus, phases 60 to 316 are analysed as a representative of all phases. Figure 4.14 shows that the standard deviation of MSD is smaller than that of MSS, which means intra-team dynamic scheduling reduces irregularity of n-body code. The irregularity of an application is defined as the standard deviation of execution times of all phases. Moreover, the standard deviation of MDD is smaller than

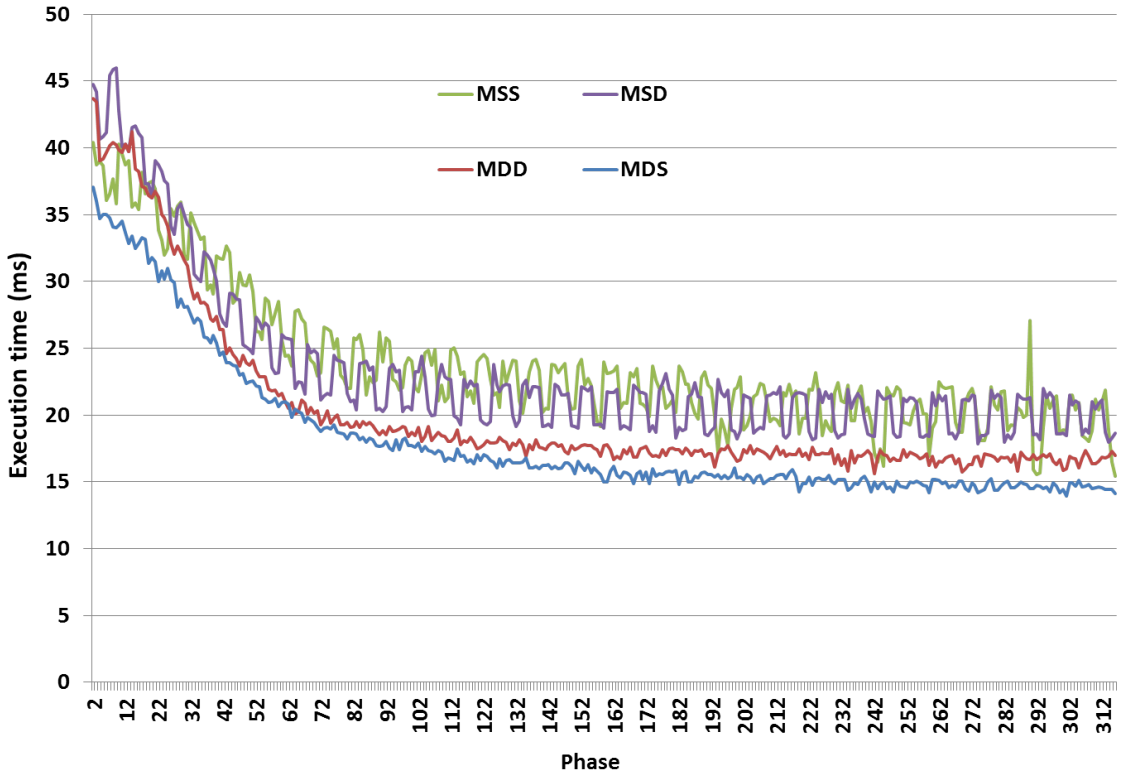


Figure 4.13: Execution time of each phase for different mechanisms in the case of 4-team execution. Phase 1 is not included in the results.

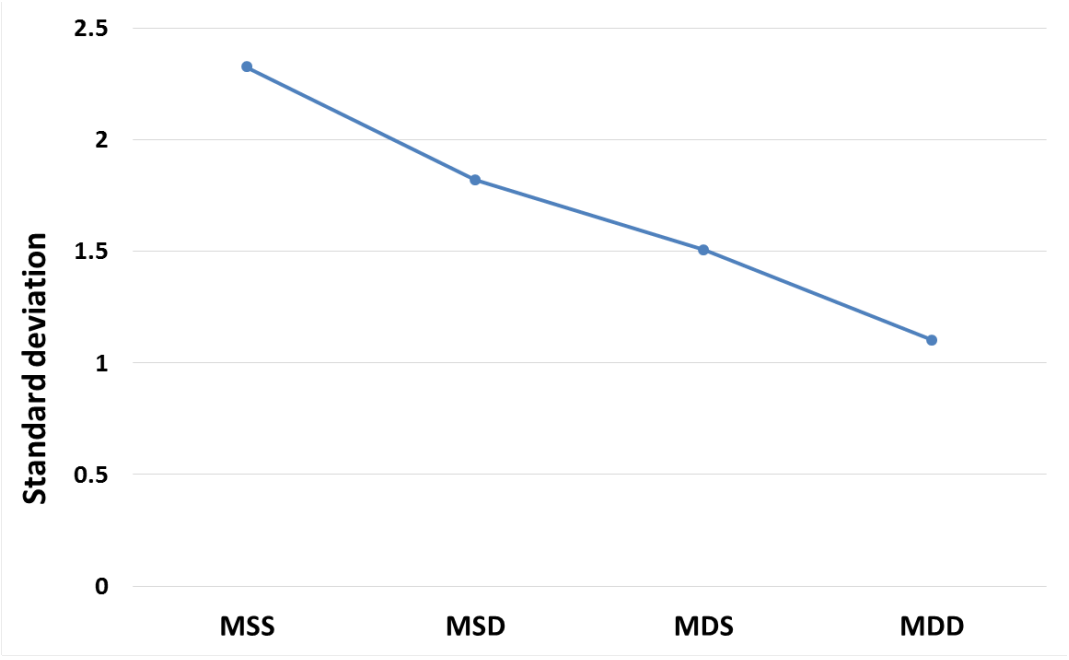


Figure 4.14: Standard deviation of execution times of phases 60 to 316 for various mechanisms.

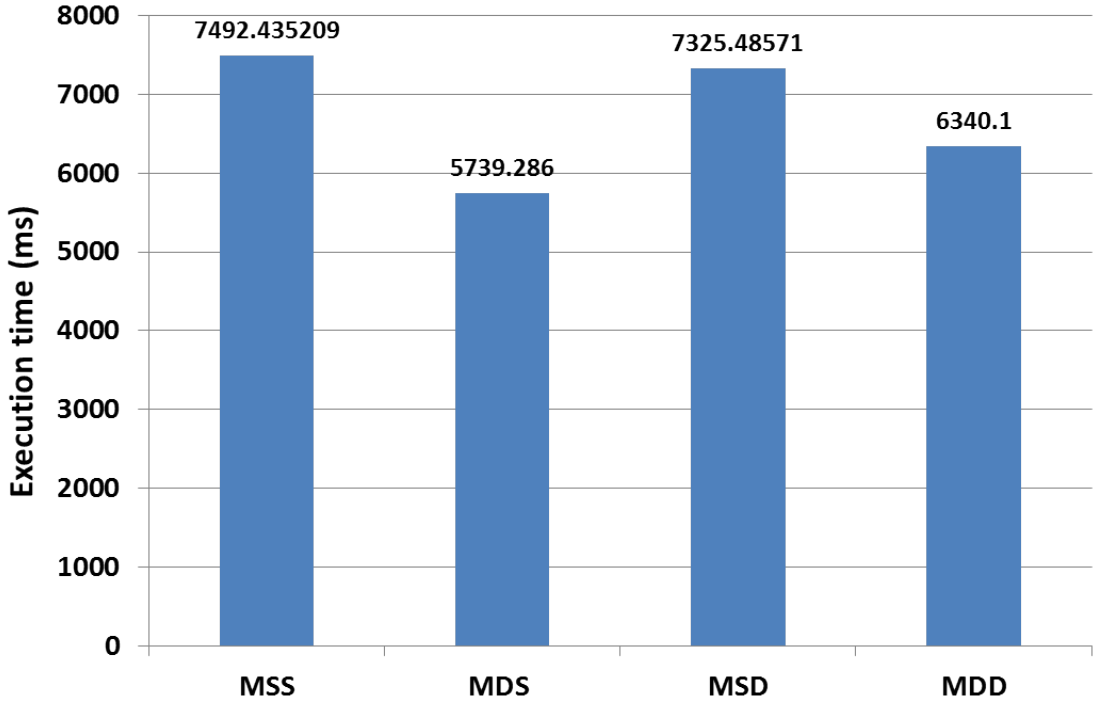


Figure 4.15: The overall execution times of various mechanisms for n-body.

that of MSD, which means inter-team dynamic scheduling further reduces irregularity of n-body code. Overall, dynamic scheduling, either intra-team or inter-team, reduces the irregularity of an application.

For a particular mechanism, by summing up the execution times of all phases except phase 1, the overall execution time of the mechanism is shown in Figure 4.15. The results in Figure 4.15 show that MDS is 23.4% better than MSS, and MDD is 14.4% better than MSD. That is, dynamic scheduling across thread teams performs better than static scheduling across teams. MSD is merely 2% better than MSS while MDS is 23.4% better than MSS. That is, inter-team dynamic scheduling has more significant impact on performance than intra-team dynamic scheduling. One interesting point is that MDS rather than MDD is the best mechanism among all the mechanisms in Figure 4.15. This is because the n-body code is becoming less irregular after inter-team dynamic scheduling is performed, and thus intra-team dynamic scheduling does not introduce significant performance gain but merely scheduling overhead that degrades performance.

One way to select MDS or MDD is to check standard deviation of execution times of

threads within a team after performing MDS. If the standard deviation is close to zero, MDS is selected. Otherwise, MDD is selected.

In summary, either intra-team or inter-team dynamic scheduling reduces the irregularity of an application. Intra-team dynamic scheduling is used for balancing the execution times within a team, and inter-team dynamic scheduling is used for balancing the execution times across different teams. Both of them can be used when load imbalance exists within a team and across different teams. However, performing both intra-team and inter-team dynamic scheduling does not necessarily achieve the best performance. When an application becomes less irregular after inter-team dynamic scheduling is performed, intra-team dynamic scheduling introduces a large overhead that might degrade the performance. Usually, it is not possible to know the optimal workload vector beforehand. With dynamic scheduling, it is possible to dynamically find the best workload vector across thread teams, though it unavoidably suffers from scheduling overhead. Section 4.4.3 discusses the inter-team scheduling overhead.

4.4.3 Scheduling overhead

This subsection shows the inter-team scheduling overhead that is estimated based on the assumptions shown in Section 4.1.

The overhead evaluation results of ray tracing are shown in Table 4.4 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 can be considered the maximum overheads. Table 4.4 shows 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,

Table 4.4: The overheads of ray tracing, 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

Table 4.5: The overheads of SoE and n-body.

Time (ms)	Overhead	MDD
SoE	152	2923.1
Time (ms)	Overhead	MDS
n-body	355.8	5739.286

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.

The scheduling overheads of SoE and n-body are shown in Table 4.5. The scheduling overhead of SoE is estimated as 152 milliseconds, which is around 5.2% of the execution time of MDD. By considering the scheduling overhead, MDS outperforms MSS by 14.5%, and MDD outperforms MSD by 11.1%, for the SoE code. The scheduling overhead of n-body is estimated as 355.8 milliseconds, which is around 6.2% of the execution time of MDS. By considering the scheduling overhead, MDS outperforms MSS by 18.6%, and MDD outperforms MSD by 8.6%, for the n-body code. Therefore, the evaluation results show that dynamic workload management is a promising way to improve performance with considering scheduling overhead.

Since the overhead of dynamic workload management across thread teams increases exponentially with the number of thread teams, it becomes riskier to use more thread teams because the overhead might kill the performance gain from dynamic workload management. Therefore, if the number of thread teams is so large that the overhead kills the performance gain, thread management introduced in Chapter 3 instead of workload management can be a solution to deal with load imbalance across thread teams.

4.4.4 Normalized performance

The performance model expressed by Eq. (4.3) 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. The performance gain under the assumption of imperfect performance prediction is discussed. As an example, suppose that a sub-optimal workload vector in each phase is predicted instead of the optimal one. In such a case, the normalized performances of MDS and MDD for the ray tracing are summarized in Figure 4.16 for both movies, and the normalized performances of MDS and MDD for the SoE are summarized in Figure 4.17, where the normalized performance is given by Eq. (4.5).

$$Normalized\ \ performance\ =\ \frac{Optimal\ execution\ time}{Suboptimal\ execution\ time} \times 100\%. \qquad (4.5)$$

The normalized performance indicates how close the suboptimal execution time is to the optimal execution time. Figures 4.16 and 4.17 show that the normalized performances are quite high (no less than 95%) for ray tracing and SoE. 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 the workload ratio is divided by a granularity of 5%, the results also indicate that the granularity is good enough so that an accurate sub-optimal performance can be predicted.

In this chapter, the inter-team dynamic workload management 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 empirical parameter tuning. By executing a given program while changing its workload vector and measuring the performance, it is possible to obtain an appropriate parameter configuration that can achieve a high performance.

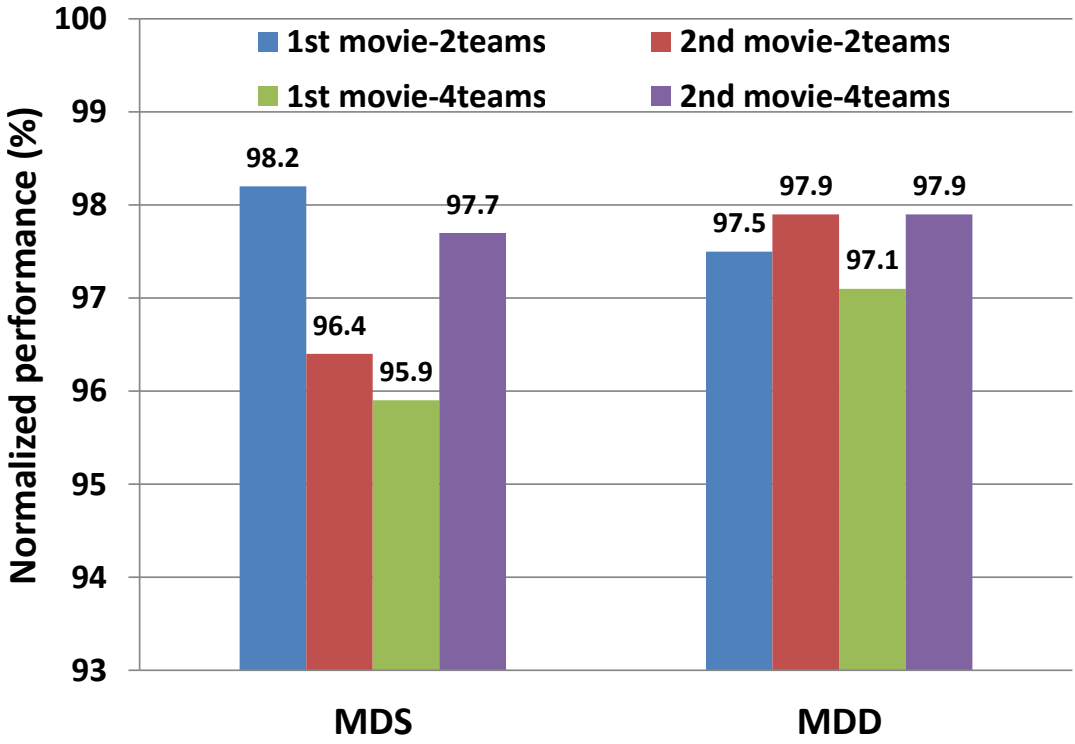


Figure 4.16: Normalized performances of MDS and MDD for ray tracing.

However, the search space of a workload vector significantly increases with the number of thread teams and the granularity of the workload ratio. Thus, how to prune the search space is a challenge. This can be a future work of this dissertation.

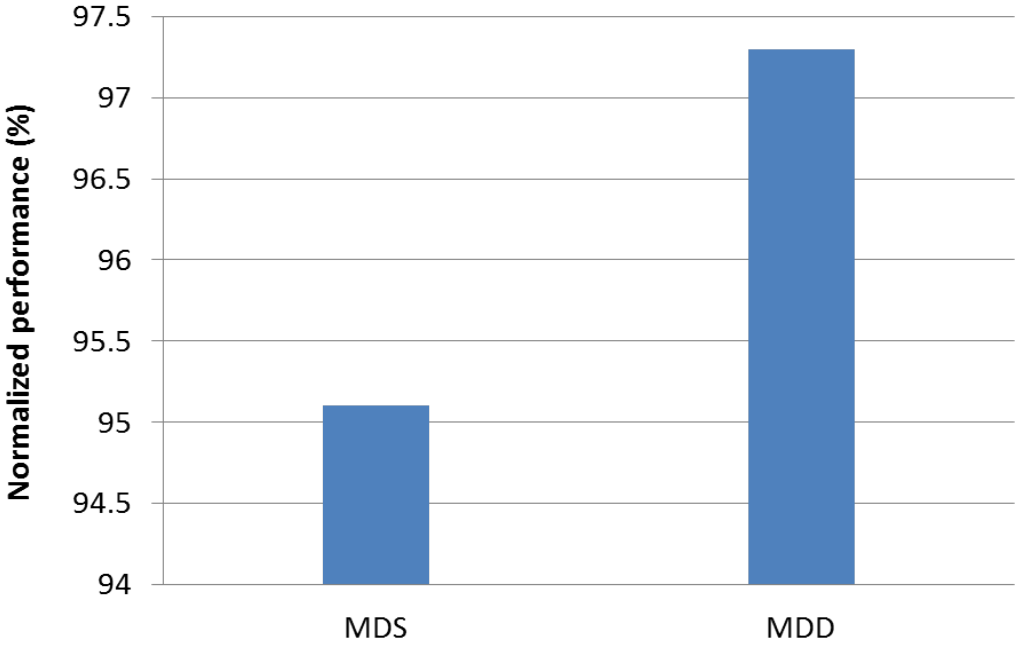


Figure 4.17: Normalized performances of MDS and MDD for SoE.

4.5 Comparison of thread management and workload management

Both thread management and workload management are introduced to solve load imbalance across thread teams. The runtime overhead of dynamic thread management is negligible compared to the total execution time of an application. On the other hand, the overhead of dynamic workload management is non-negligible, and it increases exponentially with the number of thread teams. Therefore, it is risky to use dynamic workload management when the number of thread teams is large, because the overhead would be too large to kill the performance gain from dynamic workload management. In such a case, dynamic thread management can be an alternative way to solve load imbalance across thread teams.

Thread management and workload management have their own advantages and disadvantages. Dynamic thread management has a weaker capability to solve load imbalance than dynamic workload management. However, the runtime overhead of dynamic thread management is smaller than that of dynamic workload management. Therefore, for a

given application, dynamic thread management can be employed first to deal with load imbalance. Then, if load imbalance still remains, workload management can be further employed. This is a kind of greedy strategy to solve load imbalance.

4.6 Conclusions

The objective of this chapter is to clarify the benefits of dynamic workload management for irregular applications. This chapter uses a static workload management mechanism to emulate a dynamic one. Moreover, the scheduling overhead of inter-team dynamic workload management is estimated based on two simple assumptions.

The evaluation results demonstrate that dynamic workload management, either intra-team or inter-team, can reduce the irregularity of a given application. The evaluation results also demonstrate that performing both intra-team and inter-team dynamic scheduling at the same time does not necessarily achieve the best performance. When an application is becoming less irregular after inter-team dynamic scheduling is performed, intra-team dynamic scheduling might merely introduce an overhead that degrades the performance.

This chapter shows that, at a cost of some runtime overhead, load imbalance can be solved using dynamic workload management across thread teams. However, the runtime overhead of dynamic workload management increases exponentially with the number of thread teams. Therefore, dynamic workload management may not improve performance when a large number of thread teams are used.

The future work of this dissertation includes automatically 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. This chapter assumes that only one user is using a target system. Thus, this chapter only deals with application-level load balancing. A system-level load balancing, which assumes multiple users are using a target system at the same time, can be a future work.

Chapter 5

Conclusions

Nowadays, an HPC system is becoming more heterogeneous, and many cores are integrated into the system. This dissertation discusses two types of mechanisms to deal with the heterogeneity and many-core nature of an HPC system. One type is a processor selection mechanism, and the other one is a load balancing mechanism that employs either thread management or workload management.

A research problem is that processor selection and load balancing mechanisms are often written in HPC applications, which messes up the applications and makes the applications difficult to maintain. The objective of this dissertation is to separate processor selection and load balancing mechanisms from HPC applications. The research approach is to move processor selection and load balancing mechanisms from applications to the OpenMP specification. In this way, the HPC applications become easy to maintain in long-term software development because they do not explicitly contain the two complex mechanisms.

Chapter 2 discusses runtime processor selection. The problem is that programmers need to severely modify an application for runtime processor selection. The objective of this chapter is to achieve runtime processor selection without messing up the application. To this end, this chapter proposes directive customization for runtime processor selection. The evaluation results show that, by customizing an existing OpenMP directive, three code versions are generated, and each version is suitable for a different processor. Different processors are thus selected at runtime by selecting one of the three versions based on the

problem size. This chapter demonstrates that combining code transformation and directive customization can satisfy the requirements for runtime processor selection of existing irregular OpenMP codes. In addition, directive customization can help programmers to reduce code modifications for runtime processor selection.

Chapter 3 discusses dynamic thread management across OpenMP thread teams. For execution of irregular applications, the performance changes with thread team size. Thus, the team size needs to be adjusted to find the optimal one. The objective of this chapter is to clarify the advantages of the thread management approach to solve the load imbalance across thread teams for irregular applications. This chapter uses a static thread management method to mimic a dynamic one. The evaluation results show that, even if the upper bound of the runtime overhead is considered, dynamic thread team size adjustment still can improve the performance of irregular applications in comparison with the conventional approach. Moreover, the runtime overhead of thread management is negligible in comparison with the total execution of an application, which implies that migrating OpenMP threads from one thread team to another induces only a small overhead. This chapter demonstrates that dynamic thread management cannot significantly boost performances of memory-bound irregular applications, since changing the number of threads does not significantly affect the performance.

Chapter 4 discusses dynamic workload management across OpenMP thread teams. The objective of this chapter is to clarify the benefits of dynamic workload management across thread teams. To this end, this chapter investigates the ideal performance gain and runtime overhead of dynamic workload management. This chapter uses a static workload management mechanism to emulate a dynamic one across thread teams. Moreover, the scheduling overhead of inter-team dynamic workload management is estimated based on two simple assumptions. The evaluation results show that dynamic workload management across thread teams is a promising way to improve performance. The evaluation results also show that, performing both intra-team and inter-team dynamic workload management at the same time is not always the optimal choice. This chapter demonstrates that, at a cost of some runtime overhead, load imbalance can be solved using

dynamic workload management across thread teams. However, the runtime overhead of dynamic workload management increases significantly with the number of thread teams. Therefore, dynamic workload management may not improve performance when a large number of thread teams is used.

One of the new findings of this dissertation is that multiple thread teams can outperform a single team if the application is irregular and if load balancing is properly achieved. It is because using multiple thread teams reduces the number of threads joining internal synchronizations. The second new finding is that dynamic thread management has a lower overhead in comparison with dynamic workload management, especially when the number of thread teams becomes large. The third new find is that performing both intra-team and inter-team dynamic workload management at the same time is not always the best choice. When the irregularity of an application becomes low after inter-team dynamic workload management is performed, intra-team dynamic workload management might introduce merely an overhead that degrades the performance.

One general conclusion of this dissertation is that the performances of OpenMP codes are improved without major code modifications if runtime processor selection, dynamic thread management, and dynamic workload management are integrated into future OpenMP specification.

In Chapter 3, an oracle algorithm for dynamic thread team size adjustment is assumed. Such an algorithm allows a perfect thread team size adjustment in a single step of adjustment, which is not practical. Thus, one future work of this dissertation is to develop a more practical algorithm for dynamic thread team size adjustment. Another future work is to automatically tune workload vectors to predict the best one. Since the search space of workload vectors increases drastically with the number of thread teams and the granularity of workload ratios, pruning the search space is a challenge. This dissertation only deals with application-level load balancing, which assumes that only one user is using a target system. One more future work of this dissertation is to discuss system-level load balancing, which assumes that multiple users are using a target system simultaneously.

Bibliography

[1] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. Dr. Dobb's Jounal, 30(3), March 2005.

[2] Intel Xeon Phi Processors, `https://www.intel.com/content/www/us/en/products/processors/xeon-phi/xeon-phi-processors.html`. 2018.

[3] NVIDIA Corporation. `http://www.nvidia.co.jp/page/products.html`. 2018.

[4] The Intel Corporation. Intel Xeon Phi Coprocessor Instruction Set Architecture Reference manual. Pages 1-725. Sep. 2012.

[5] The OpenMP Application Programming Interface. `http://www.openmp.org/`. Version 5.0, Nov. 2018.

[6] The OpenMP Application Programming Interface. `http://www.openmp.org/`. Version 4.0, Jul. 2013.

[7] The OpenACC Application Programming Interface. `http://www.openacc.org/`. Version 2.5, Oct. 2015.

[8] Extensible Markup Language (XML). `https://www.w3.org/XML/`. 2017.

[9] Hiroyuki Takizawa, Shoichi Hirasawa, and Hiroaki Kobayashi. Xevolver: an XML-based Programming Framework for Software Evolution. The International Conference for High Performance Computing, Networking, Storage and Analysis. 2013.

[10] Hiroyuki Takizawa, Shoichi Hirasawa, Yasuharu Hayashi, Ryusuke Egawa, Hiroaki Kobayashi, Xevolver: An XML-based Code Translation Framework for Supporting

HPC Application Migration. IEEE International Conference on High Performance Computing (HiPC), pages 1-11, Dec. 2014.

[11] Top 500 the list. `https://www.top500.org`. 2018.

[12] Wei Xue, Chao Yang, Haohuan Fu, Xinliang Wang, Yangtong Xu, Lin Gan, Yutong Lu, Xiaoqian Zhu. Enabling and Scaling a Global Shallow-Water Atmospheric Model on Tianhe-2. 2014 IEEE 28th International Symposium on Parallel and Distributed Processing. 2014.

[13] XSL Transformations (XSLT) Version 2.0. `http://www.w3.org/TR/xslt20/`. 2007.

[14] Xinwei Wang, Chunjing Cao. Mining Association Rules from Complex and Irregular XML Documents Using XSLT and Xquery. International Conference on Advanced Language Processing and Web Information Technology (ALPIT'08). 2008.

[15] Jinpil Lee and Mitsuhisa Sato. Implementation and Performance Evaluation of XcalableMP: A Parallel Programming Language for Distributed Memory Systems. The 39th international Conference on Parallel Processing Workshops (ICPPW10), pp.413-420, San Diego, CA, Sep. 2010.

[16] Masahiro Nakao, Jinpil Lee, Taisuke Boku, Mitsuhisa Sato. XcalableMP Implementation and Performance of NAS Parallel Benchmarks. Fourth Conference on Partitioned Global Address Space Programming Model (PGAS10), NewYork, USA, Oct. 2010.

[17] MPI: A Message-Passing Interface Standard. Version 3.1. Message passing Interface Forum. June 4, 2015.

[18] J. Bueno, L. Martinell, A. Duran, M. Farreras, X. Martorell, R. M. Badia, E. Ayguade, and J. Labarta. Productive Cluster Programming with OmpSs. In Euro-par parallel processing. Pages 555-566, 2011.

[19] Javier Bueno, Judit Planas, Alejandro Duran, Rosa M. Badia. Productive Programming of GPU Clusters with OmpSs. 2012 IEEE 26th International Parallel and Distributed Processing Symposium. Pages 557-568. 2012.

[20] Seyong Lee and Rudolf Eigenmann. OpenMPC: Extended OpenMP Programming and Tuning for GPUs. The International Conference for High Performance Computing, Networking, Storage and Analysis. Pages 1-11. 2010.

[21] Masahiro Nakao, Hitoshi Murai, Takenori Shimosaka, Akihiro Tabuchi, Toshihiro Hanawa, Yuetsu Kodama, Taisuke Boku, Mitsuhisa Sato. XcalableACC: Extension of XcalableMP PGAS Language using OpenACC for Accelerator Clusters. Workshop on accelerator programming using directives (WACCPD), New Orleans, LA, USA. Pages 27-36. 2014.

[22] Jinpil Lee, Mitsuhisa Sato, Taisuke Boku. OpenMPD: a directive-based data parallel language extension for distributed memory systems. International Conference on Parallel Processing Workshops. Pages 121-128. 2008.

[23] D. Quinlan and C. Liao, The rose source-to-source compiler infrastructure, in Cetus users and compiler infrastructure workshop. Pages 1-3. 2011.

[24] Daniel Quinlan, Markus Schordan, Richard Vuduc, Qing Yi, Thomas Panas, Chunhua Liao, and Jeremiah J. Willcock. ROSE Tutorial: A Tool for Building Source-to-Source Translators. Lawrence Livermore National Laboratory. Pages 1-430. 2018.

[25] Qing Yi, Keith Seymour, Haihang You, Richard Vuduc, Dan Quinlan. POET: Parameterized Optimizations for Empirical Tuning. International Parallel and Distributed Processing Symposium (IPDPS). Pages 1-8. 2007.

[26] Chun Chen, Jacqueline Chame, Mary Hall. CHiLL: A framework for composing high-level loop transformations. Technical Report 08-897, University of Southern California. Pages 136-150. 2008.

[27] Albert Hartono, Boyana Norris, P. Sadayappan. Annotation-Based Empirical Performance Tuning using Orio. Proceedings of International Parallel and Distributed Processing Symposium (IPDPS). Pages 1-11. 2009.

[28] Qing Yi. POET: a scripting language for applying parameterized source-to-source program transformations. Software: Practice and Experience. Volume 42, Issue 6, pages 675-706. 2012.

[29] Mary Hall, Jacqueline Chame, Chun Chen, Jaewook Shin, and Gabe Rudy. Transformation Recipes for Code Generation and Auto-Tuning. Proceeding of the 22nd International Workshop on Languages and Compilers for Parallel Processing. Pages 50-64. 2009.

[30] Qing Yi. Optimizing and Tuing Scientific Codes. Scalable Computing and Communications: Theory and Practice. Chapter 11. 2011.

[31] Bonifacio Martin-del-Brio, Antonio Bono-Nuez, Nicolas Medrano-Marques. Self-organizing maps for embedded processor selection. microprocessors and microsystems, Volume 29, Issue 7, pages 307-315. 2005.

[32] N. Parimala, S.R.N. Reddy. Processor selection for embedded system design. International Journal of Computers and Applications. Volume 30, Issue 4, Pages 348-353. 2015.

[33] E. Erwin, K. Obermayer, K. Schulten. Self-organizing maps: ordering, convergence properties and energy functions. Journal of Biological Cybernetics, Volume 67, Issue 1, pages 47-55. 1992.

[34] S. Haykin. Neural networks: a comprehensive foundation. Prentice Hall, New Jersey, 1999.

[35] Nicholas H. Mastronarde, and Mihaela van der Schaar. A quwuing-theoretic approach to task scheduling and processor selection for video-decoding applications. IEEE Transactions on Multimedia, Vol. 9, No. 7, pages 1493-1507. 2007.

[36] H. Takizawa, K. Sato, and H. Kobayashi. SPRAT: Runtime processor selection for energy-aware computing. Third International Workshop on Automatic Performance Tuning (iWAPT 2008). Pages 386-393. 2008.

[37] H. Takizawa, H. Shiratori, K. Sato, H. Kobayashi. SPRAT: A stream programming language with runtime auro-tuning. In Symposium on Advanced Computing Systems and Infrastructures. Pages 139-148. 2008.

[38] G. Anderson. An Ada Multitasking Solution for the Sieve of Eratosthenes. Ada Lett. VIII, 5 (Sept. 1988), pp 7174. 1988. https://doi.org/10.1145/51624.51628

[39] MELISSA E. ONEILL. The Genuine Sieve of Eratosthenes. Journal of Functional Programming 19, 1 (2009), pp 95-106. 2009. https://doi.org/10.1017/S0956796808007004

[40] Luan Orlandi. Parallel Sieve of Eratosthenes. 2016. https://github.com/luanorlandi/Parallel-Sieve-Quicksort/blob/master/sieve.c

[41] U Becciani, R Ansaloni, V Antonuccio-Delogu, G Erbacci, M Gambera, and A Pagliaro. A parallel tree code for large N-body simulation: dynamic load balance and data distribution on a CRAY T3D system. Computer Physics Communications. Volume 106, Issues 12, Pages 105-113, October 1997.

[42] Josh Barnes and Piet Hut. A hierarchical O(N log N) force-calculation algorithm. Nature 324, Pages 446-449, December 1986.

[43] Ananth Y. Grama, Vipin Kumar, and Ahmed Sameh. Scalable parallel formulations of the barnes-hut method for n-body simulations. Proceedings of the 1994 ACM/IEEE conference on Supercomputing. Pages 439-448, 1994.

[44] Mathias Winkel, Robert Speck, Helge Hbner, Lukas Arnold, Rolf Krause, Paul Gibbon. A massively parallel, multi-disciplinary BarnesHut tree code for extreme-scale N-body simulations. Computer Physics Communications. Volume 183, Issue 4, Pages 880-889, April 2012.

[45] BarnesHut 3D N-body Simulation Using the Intel Xeon Phi coprocessor. 2015. https://github.com/gpiskas/Barnes_Hut_Intel_Phi

[46] Frank Losasso, Frdric Gibou, and Ron Fedkiw. Simulating water and smoke with an octree data structure. ACM Transactions on Graphics. Volume 23 Issue 3, Pages 457-462, August 2004.

[47] Donald Meagher. Geometric modeling using octree encoding. Computer Graphics and Image Processing. Volume 19, Issue 2, Pages 129-147, June 1982.

[48] Michael Potmesil. Generating octree models of 3D objects from their silhouettes in a sequence of images. Computer Vision, Graphics, and Image Processing. Volume 40, Issue 1, Pages 1-29, October 1987.

[49] John Dubinski. the origin of the brightest cluster galaxies. The Astrophysical Journal. Volume 502, Number 1, Pages 141-149. 1998.

[50] John K. Salmon, and Mochael S. Warren. Fast Parallel Tree Codes for Gravitational and Fluid Dynamical N-Body Problems. The International Journal of High Performance Computing Applications. Volume 8, Issue 2, Pages 129-142. June 1994.

[51] Xiong Xiao, Shoichi Hirasawa, Hiroyuki Takizawa, and Hiroaki Kobayashi. Toward Dynamic Load Balancing across OpenMP Thread Teams for Irregular Workloads. International Journal of Networking and Computing. 7, 2 (2017), pages 387404. 2017.

[52] Xiong Xiao, Shoichi Hirasawa, Hiroyuki Takizawa, and Hiroaki Kobayashi. The Importance of Dynamic Load Balancing among OpenMP Thread Teams for Irregular Workloads. In The 4th International Symposium on Computing and Networking. Pages 529535. 2016.

[53] Gary E. Christensen. MIMD vs. SIMD parallel processing: A case study in 3D medical image registration. Journal of Parallel Computing. Volume 24, Issues 9-10, pages 1369-1383. 1998.

[54] Jan H. Schonherr, Jan Richling, Hans-Ulrich Heiss. Dynamic teams in OpenMP. 22nd International Symposium on Computer Architecture and High Performance Computing. 2010.

[55] Jin Wang, Norm Rubin, Albert Sidelnik, Sudhakar Yalamanchili. Dynamic Thread Block Launch: A lightweight execution mechanism to support irregular applications on GPUs. 2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture. 2015.

[56] Ahmad Lashgar, Amirali Baniasadi, Ahmad Khonsari. Dynamic warp resizing: analysis and benefits in high-performance SIMT. IEEE 30th International Conference on Computer Design. 2012.

[57] Chao-Hung Hsu, I-Wei Wu, Jean Jyh-Jiun Shann. Dynamic memory optimization and parallelism management for OpenCL Information Science, Electronics and Electrical Engineering (ISEEE), 2014 International Conference on. 2014.

[58] Yi Yang, Ping Xiang, Jingfei Kong, Huiyang Zhou. A GPGPU compiler for memory optimization and parallelism management. Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation. Pages 86-97. 2010.

[59] Nvidia Group. NVIDIA CUDA C Programming Guide. Version 4.2. 2012.

[60] Khronos OpenCL Working Group. The OpenCL Specification. Version 2.2. 2017.

[61] Lewis Bil and Berg Daniel J. Multithreaded Programming with Pthreads. Prentice-Hall, Inc. 1998.

[62] Narlikar Girija J. and Blelloch Guy E. Pthreads for Dynamic and Irregular Parallelism. Proceedings of the 1998 ACM/IEEE Conference on Supercomputing. pages 1-16. 1998.

[63] Wong Michael, Klemm Michael, Duran Alejandro, Mattson Tim, Haab Grant, B. R. de Supinski, and Churbanov Andrey. "Towards an Error Model for OpenMP". In Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and More. Springer Berlin Heidelberg. 2010, pp70-82.

[64] Ling Yibei, Mullen Tracy, and Lin Xiaola. "Analysis of Optimal Thread Pool Size". SIGOPS Oper. Syst. Rev. Vol. 34, No. 2, pp42-55. April 2000.

[65] X. Tian, M. Girkar, S. Shah, D. Armstrong, E. Su, and P. Petersen. Compiler and runtime support for running openmp programs on pentium and itanium architectures. Proceedings of Eighth International Workshop on High-Level Parallel Programming Models and Supportive Environments. Pages 47-55. April 2003.

[66] P. E. Hadjidoukas and V. V. Dimakopoulos. Nested parallelism in the ompi openmp/c compiler. In Proceedings of International European Conference on Parallel processing. Pages 662-671. 2007.

[67] Weilun Sun, Ling-Qi Yan, Yi Wu, John Kubiatowicz. GPU Memory Management for Efficient Ray Tracing Applications. Proceedings of the Eurographics Symposium on Rendering 2014. Pages 1-7. 2014.

[68] Ivan Zecena, Martin Burtscher, Tongdan Jin, Ziliang Zong. Evaluating the performance and energy efficiency of n-body codes on multi-core CPUs and GPUs. Proceedings of IEEE 32nd International Performance Computing and Communications Conference (IPCCC). Pages 1-8. 2013.

[69] Nathan R. Fredrickson, Nathan R. Fredrickson, and Ying Qian. 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.

[70] 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.

[71] 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.

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

[73] 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.

[74] 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.

[75] 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.

[76] 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.

[77] Z. M. Fadlullah, T. Taleb, N. Ansari, K. Hashimoto, Y. Miyake, Y. Nemoto, and N. Kato. Combating Against Attacks on Encrypted Protocols. In Proceedings of IEEE International Conference on Communications, pp. 1211-1216. Jun 24-28, 2007.

[78] Thomas R. W. Scogland, Barry Rountree, Wuchun 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.

[79] 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 Inter- national Workshop on OpenMP (IWOMP), pages 141-155, 2013.

[80] 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.

[81] 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.

[82] 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.

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

[84] 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.

[85] 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.

[86] 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.

[87] 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.

[88] Richard Vuduc John Shalf Katherine Yelick James Demmel Samuel Williams, Leonid Oliker. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. Journal of parallel computing, 35(3):178-194, 2009.

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

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

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

[92] 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.

[93] 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.

[94] 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.

[95] Kevin Beason. Smallpt: global illumination in 99 lines of C++. `http://www.kevinbeason.com/smallpt`.2014.

Publications

Journal Paper

[1] Xiong Xiao, Shoichi Hirasawa, Hiroyuki Takizawa and Hiroaki Kobayashi. Toward Dynamic Load Balancing across OpenMP Thread Teams for Irregular Workloads. International Jounal of Networking and Computing. Pages 387-404. Vol. 7, No. 2, July 2017. (Chapter 4)

Conference Papers

[2] Xiong Xiao, Mulya Agung, Muhammad Alfian Amrizal, Ryusuke Egawa, and Hiroyuki Takizawa. Investigating the Effects of Dynamic Thread Team Size Adjustment for Irregular Applications. Proceedings of The 6th International Symposium on Computing and Networking. Pages 76-84. Nov. 2018. (Chapter 3)

[3] Xiong Xiao, Shoichi Hirasawa, Hiroyuki Takizawa and Hiroaki Kobayashi. Importance of Dynamic Load Balancing among OpenMP Thread Teams for Irregular Workloads. Proceedings of The 4th International Symposium on Computing and Networking. Pages 529-535. Nov. 2016. (Chapter 4)

[4] Xiong Xiao, Shoichi Hirasawa, Hiroyuki Takizawa and Hiroaki Kobayashi. An Approach to Customization of Compiler Directives for Application-Specific Code Transformations. Proceedings of 2014 IEEE 8th International Symposium on Embedded Multicore/Manycore SoCs. Pages 99-106. 2014. (Chapter 2)

Domestic Publication

[5] Xiong Xiao, Shoichi Hirasawa, Hiroyuki Takizawa, and Hiroaki Kobayashi. A case study of performance tuning using the POET framework. In Student Session, Tohoku-Section Joint Convention of Institutes of Electrical and Information Engineers, Japan. Aug. 22nd, 2013.

Poster

[6] Hiroyuki Takizawa, Xiong Xiao, Shoichi Hirasawa, Hiroaki Kobayashi. An XML-based Programming Framework for User-defined Code Transformations. The 4th AICS International Symposium, Kobe, Dec. 2-3, 2013.

Acknowledgments

This dissertation would not have been carried out without the help of many people in many ways. The author would like to thank all of them gratefully.

First of all, I would like to express my sincere gratitude to my supervisor, Professor Hiroyuki Takizawa, for the continuous support of my Ph.D study and research, for his patience, enthusiasm, and invaluable advices, as well as financial aid. I benefited immensely from his support, counsel, and encouragement. His guidance helped me in all the time of research and writing of this dissertation.

I wish to express my gratitude to Professor Hiroaki Kobayashi for spending precious time in proof-reading my published papers and comments on this dissertation, kindly providing insightful advices and comments during my Ph.D study.

I would like to thank Professor Masanori Hariyama, for being one of the committee members, and for his encouragement and helpful comments on this dissertation.

I would also like to thank Associate Professor Ryusuke Egawa sincerely for providing insightful comments and advices during my Ph.D study and thoughtful review of this dissertation.

I express my sincere gratitude also to Associate Professor Kazuhiko Komatsu, Assistant Professor Shoichi Hirasawa, and Assistant Professor Masayuki Sato for their help and support during my Ph.D study.

I would also like to express my appreciation to all other members in Takizawa, Kobayashi and Egawa Laboratories, for spending precious time together, and for kind friendship among us. Special thanks go to Dr. Muhammad Alfian Amrizal and Mr. Mulya Agung for their discussions and help during my Ph.D study.

Finally, I would like to express my deep appreciation to my family. I want to thank my parents for their affectionate encouragement and vast support. I also want to thank my younger sister for her encouragement.

Xiong XIAO

January, 2019