



저작자표시-비영리-변경금지 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:



저작자표시. 귀하는 원저작자를 표시하여야 합니다.



비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.



변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 [이용허락규약\(Legal Code\)](#)을 이해하기 쉽게 요약한 것입니다.

[Disclaimer](#)

M.S. THESIS

Spatial Scheduling and Load Balancing Techniques for the Argobots Runtime System

Argobots 런타임 시스템을 위한 공간 분할 스케줄링 및
부하 분산 기법

AUGUST 2018

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

AN NA

M.S. THESIS

Spatial Scheduling and Load Balancing Techniques for the Argobots Runtime System

Argobots 런타임 시스템을 위한 공간 분할 스케줄링 및
부하 분산 기법

AUGUST 2018

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

AN NA

Spatial Scheduling and Load Balancing Techniques for
the Argobots Runtime System

Argobots 런타임 시스템을 위한 공간 분할 스케줄링 및
부하 분산 기법

지도교수 Bernhard Egger

이 논문을 공학석사 학위논문으로 제출함

2018 년 06 월

서울대학교 대학원

컴퓨터 공학부

안나

안나의 공학석사 학위논문을 인준함

2018 년 07 월

위 원 장	김 선	(인)
부위원장	Bernhard Egger	(인)
위 원	Srinivasa Rao Satti	(인)

Abstract

As each day passes-by, the evolution of multi-core processors is progressing. While this shift carries a massive parallel processing potential, it equally sets a noteworthy challenge. In particular, keeping all cores busy and minimizing the synchronization costs is a difficult issue.

Argobots is a framework based on user-level threading for OpenMP applications on multi-core processors designed and implemented by Argonne National Laboratory. In this thesis, we present the fundamental theory and preeminence of load balancing methods on the Argobots runtime system. We alter the structure of the Argobots runtime framework and then subsequently bring in a new mechanism to balance the workload using user-level thread spatial arrangement. Both the static load and dynamic load balancing methods are used together by the optimized Argobots.

Using the NAS Parallel Benchmark suite, we assess and analyze the performance of the optimized Argobots runtime system. We achieve a performance improvement between 10% to 20%. Improvements are more pronounced for multiple co-runnnig applications. To summarize, this thesis reveals the benefits of load balancing and spatial scheduling in the Argobots runtime system.

Keywords: spatial scheduling, Argobots framework, workload balancing, user-level thread

Student Number: 2016-26674

Contents

Abstract	i
Contents	iii
List of Figures	vii
List of Tables	ix
Chapter 1 Introduction	1
1.1 Multi-core Multi-threaded Processors	1
1.2 Task Scheduling	2
1.3 Parallel Runtimes	3
1.4 Contributions and Outline	4
Chapter 2 Background and Motivation	6
2.1 Background	6
2.1.1 Multi-threaded Parallelism	6
2.1.2 OpenMP	7
2.1.3 BOLT	8
2.1.4 ARGO	9

2.1.5	User-Level Threads	10
2.1.6	Load Balancing in Multi-core Programming	10
2.2	Related Work	12
2.3	Motivation	13
Chapter 3	Argobots Runtime Architecture	15
3.1	Argo Architecture Layout	15
3.2	Argo Runtime: Argobots	16
3.2.1	User-Level Threads and Tasklets	17
3.2.2	Execution Streams	19
3.2.3	Scheduler and Pool	20
3.2.4	Work Unit Migration	23
Chapter 4	Implementation	24
4.1	Data Structures	26
4.2	Work Load Balancing	28
4.3	Choosing the Threshold	30
4.4	Core Allocation	32
Chapter 5	Evaluation Results and Analysis	36
5.1	Benchmarks	36
5.2	Evaluation of Work Load Balancing	38
5.3	Evaluation of Different Thresholds	39
5.4	Evaluation of Core Allocation	39
5.4.1	Executing Two Parallel Applications	40
5.4.2	Executing Three or More Parallel Applications	42
Chapter 6	Conclusions	47

Bibliography	48
--------------	----

요약	53
----	----

List of Figures

Figure 2.1	OpenMP execution model.	8
Figure 2.2	The structure of threads, runtimes, and compilers.	9
Figure 3.1	Argo architecture layout.	16
Figure 3.2	Relationship of execution streams and work units.	17
Figure 3.3	User-level thread states.	19
Figure 3.4	Execution stream states.	20
Figure 3.5	Relationship between execution streams, pools, sched- ulers, and cores.	22
Figure 3.6	Push and pull operation.	22
Figure 4.1	Relationship of thresholds and work load.	25
Figure 4.2	The composition of a work set.	27
Figure 4.3	Flow chart of work load balancing.	29
Figure 4.4	The relationship between controller and monitors.	29
Figure 4.5	Flow chart of core allocating.	34
Figure 5.1	Evaluation results of single application.	38
Figure 5.2	Evaluation results of different thresholds.	40

Figure 5.3	Evaluation results of two co-located parallel applications.	41
Figure 5.4	Evaluation results for three co-located applications. . . .	43
Figure 5.5	Evaluation results for four co-located applications. . . .	44
Figure 5.6	The relationship of applications and nodes.	44
Figure 5.7	Evaluation results under half nodes.	45
Figure 5.8	Runtime results of the longest applications.	46

List of Tables

Table 5.1	Benchmarks used for the evaluation.	37
Table 5.2	Input data size per class for the different benchmarks. . .	37

Chapter 1

Introduction

1.1 Multi-core Multi-threaded Processors

With the increasing demand for computer processing performance, the method of improving computer performance by raising the clock frequency of the processor could not fit the new requirements. It means that the single-core processors do not satisfy the application requirements, so multi-core processors have been developed [16].

Multi-core processors integrate two or more processing cores on one integrated circuit chip. Stanford University in 1996 proposed the idea, and after over twenty years of development, multi-core processors have become the mainstream microprocessor in the market. Meanwhile, with the development of science and technique, multi-core processors have widespread use as system functions and performance requirements are increasing. Compared with traditional single-core processors, multi-core processors have the following advantages [19]:

- Multi-core processors execute more tasks simultaneously because of the

multiple computing cores, which more effectively improves the parallel processing capabilities of the processor.

- The multi-core processor system is secure to expand. Multi-core processor encapsulates multiple independent cores inside one chip to achieve higher processor performance at a lower frequency. Therefore, the cycle of the multi-core processor is much shorter than that of the single-core processor.
- Multi-core processors share resources among multiple processing cores to improve the resource utilization.
- The multi-core processor can monitor the load distribution of each processing core, balance the load on each core in real time, optimize task scheduling, and efficiently reduce its power consumption [21].

In short, multi-core processors have advantages that single-core processors do not have, which makes multi-core processors to replace the single-core. With the rapid development of multi-core technology, multi-core processors have begun to penetrate into various areas, including parallel systems including the embedded system domain. Multi-core processors provide a broader platform and also make systems more complex.

1.2 Task Scheduling

The development of multi-core multi-threaded processors challenges traditional compilation and scheduling techniques because most applications were serial until then. Single-core schedulers or serial programs are run unmodified on multi-core processors are unable to exploit the high parallel processing capabilities of multi-core processors. To adapt to multi-core multi-threaded processors, parallel compilation techniques are introduced [21]. There are many challenges to take full use of multiple cores: how to divide the program, that is, how to

divide the original complete program into multiple tasks to execute in parallel; how to schedule tasks, in other words, how to allocate the task to the appropriate core.

The quality of task scheduling directly affects the performance of the system [7]. It is very likely that the advantages of parallelism will be completely wiped out, or even worse than the serial one, if not correctly scheduled. Task scheduling methods are static or dynamic. Static task scheduling decides the program schedule at compile time [13]. In dynamic task scheduling, the scheduler dynamically allocates different tasks to each processor according to the runtime conditions to minimize the total runtime and reduce the overhead caused by schedulers [17, 4].

In addition to the necessary processor computing resources, task execution also consumes resource in the memory and on I/O devices. Because multiple tasks share resources, to reduce the negative impact of resource competition, we must adopt the necessary mutex mechanisms such as semaphores and monitors. For simplicity, we assume that only processor computing resources compete between tasks and ignore other shared resources.

1.3 Parallel Runtimes

There are some new problems about parallel runtimes in multi-core structure designing and application development. These include two main aspects: memory storage structure design and on-chip communication overhead.

The first one, memory storage structure design concerns the speed gap between the processor and the main memory. This has always been an existing problem to consider in the design of the processor structure. In a single-processor, this problem can be substantially solved by using a cache to achieve

a perfect performance. However, for multiple processors, as the number of cores increase, the need for main memory access increases. The problem caused by the speed gap between the cores and the main memory has become severe.

The second aspect is on-chip communication overhead. In multi-core applications, each core executes its code, but data sharing and synchronization among different cores is also needed. Hence, the performance of the on-chip communication structure will directly affect the performance. Moreover, when several applications work together, if different applications are dependent, that is, applications will share data with each other, the communication overhead can have a significant impact on performance. Otherwise, if there are only independent applications, that is, there is no communication or shared data, there is also a considerable overhead caused by context switching among different applications.

1.4 Contributions and Outline

This thesis describes the characteristics of multi-core processors architectures and analyzes the state of task scheduling on multi-core processors. Starting with static task scheduling, this thesis focuses on the task scheduling problem on multi-core and multi-threaded processors. Based on the Argobots runtime [3], a load balancing method for multi-core multi-threaded application is proposed. After implementation and an evaluation, the results are analyzed.

According to the order of the research, the content of this thesis is divided into four chapters. The thesis is organized as follows:

Section 1. Introduction. The research backgrounds and significances are explained. The main work and organization structure of this thesis are also summarized.

Section 2. Background and motivation. The motivations and related technologies are explained, the backgrounds are also introduced.

Section 3. Argobots Runtime Architecture. Introduce the research platform and analyze the workflow in task scheduling.

Section 4. Implementation. A load balancing method to optimize the task scheduling and the associated implementation are proposed under the Argobots runtime architecture. Modify the data structure, and then optimize the load balancing both in the single application and multiple application.

Section 5. Evaluation. Introduce the benchmark and the environment used in this thesis briefly. Expand the test process in detail, demonstrate and analyze the evaluation result.

Section 6. Conclusions. Some conclusions are given.

Chapter 2

Background and Motivation

2.1 Background

2.1.1 Multi-threaded Parallelism

As each day passes-by, the evolution of processors is progressing. As the realm of information develops, multi-core processors have revealed unique benefits. Market and future technological developments also place more hope on multi-core processors. These all indicate that the multi-core processor is the next development trend. However, while the multi-core processor brings vast processing potential, it also challenges program developers and development tools.

It is necessary to write programs that can be executed concurrently based on the multi-core processor model to make full use of multi-core processor computing power [9]. At present, there are mainly four parallel programming models based on multi-threading [23]. The first type is API functions of the operating system. The second one is adopting a programming language that supports multi-threaded programming. The third way is linking and integrating

the program with a library that is optimized for a multi-threaded execution environment. The last method is extending existing programming languages; here, OpenMP is one of the most famous examples. Due to its support for multiple programming languages and simple API, it is popular with the programmer. This thesis will do some research on the spatial scheduling of threads among multi-core processors.

2.1.2 OpenMP

Open Multi-Processing is an application programming interface that supports C, C++, and the Fortran language [15]. OpenMP explicitly performs parallelization on a thread basis, providing full control of parallelization. It uses the fork/join model when executing programs in parallel. The basic idea is that the serial area is executed by the main thread, parallel programs are executed by parallel threads, and the next serial execution starts after all the parallel programs have completed. The following are the steps of the OpenMP execution model, and Figure 2.1 shows the structure of it.

Step 1. Only the main thread runs, when the application starts execution.

Step 2. When parallel computing is needed, the main thread derives parallel threads by creating new threads or waking existing threads.

Step 3. The main thread and derived threads work together during parallel execution.

Step 4. Once parallel work completes, all the derived threads exit or stop. Control flow goes back to the main thread.

OpenMP has been widely used in many applications because of its simplicity of incremental parallelism [15]. But in recent high-performance computing trends, it has encountered some challenges [1]:

1. Because the number of CPU cores are increasing, OpenMP instances are

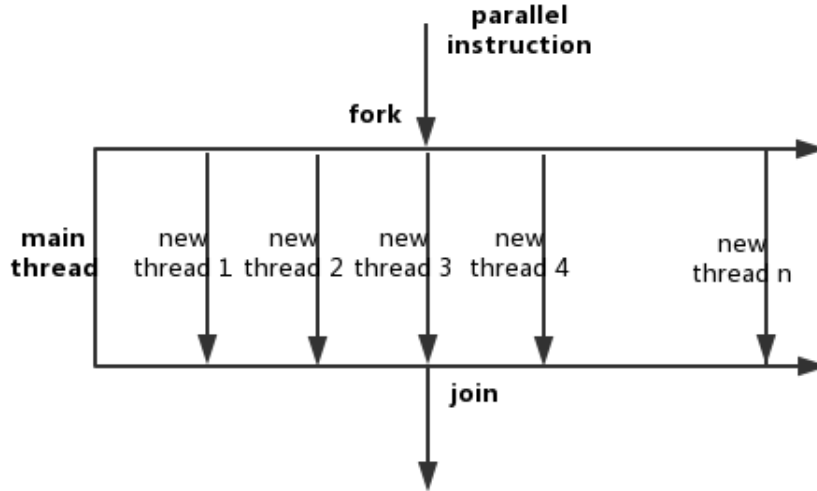


Figure 2.1: OpenMP execution model.

required to express more parallelism.

2. Hybrid programming requires better co-ordination, so threading model is usually used to communicate.

Because of its heavyweight threading model, it may be difficult to handle these challenges or be inefficient in current OpenMP implementations.

2.1.3 BOLT

Bolt is an abbreviation for “BOLT is OpenMP over Lightweight Threads” [22]. BOLT implements OpenMP by exploiting Argobots and can solve the above challenges while achieving better performance. The goal of BOLT is a high-performance OpenMP implementation, especially for fine-grained parallelism [5]. BOLT uses a lightweight threading model as its underlying mechanism, and the current implementation is based on the OpenMP runtime in LLVM. Thus BOLT

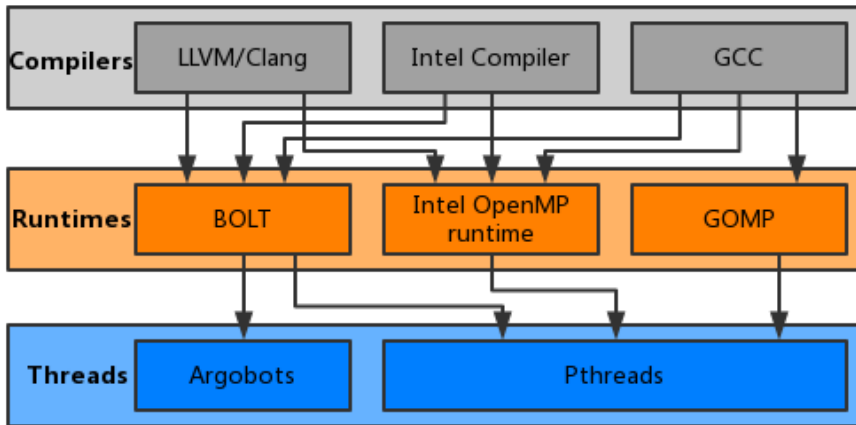


Figure 2.2: The structure of threads, runtimes, and compilers.

can be used with LLVM/Clang, the Intel OpenMP compiler, and GCC. Figure 2.2 shows the structure of threads, the runtime, and the compiler [22].

2.1.4 ARGO

Argo is an exascale runtime research project to support large-scale calculations. It is designed from the ground up to run future high-performance computing applications at extreme scales [19].

Argobots is the runtime component of Argo, and it is a lightweight low-level threading framework [22]. Argobots implements low-level threading entirely in userspace, which makes users have full control of resource and provides data movement infrastructure and task libraries for large-scale concurrent systems. The Argobots runtime also offers an Argobots interface for higher level applications and libraries.

As concurrency in HPC system explodes, there will be billions of threads

in future applications, so scientists are adjusting the calculation methods and programming models for emerging applications. The new programming model divides the computational problem into lightweight tasks to solve the massive parallelism problem. Meanwhile, it can hide latency, increase concurrency, and increase resiliency. Argobots is a lightweight runtime system to supports integrated computing and data movement as well as large-scale concurrency.

2.1.5 User-Level Threads

There are two main threads: kernel-level threads and user-level threads. Kernel-level threads can be managed and known by the kernel, and they are created and destroyed by the system. They have better performance especially for applications that frequently block, but they significant overhead and are hundreds of times slower than that of user-level threads [2].

User-level threads exist only in user-space. The kernel knows nothing about creation, revocation, synchronization, and communication [20]. They are cheap and fast, also do not require modification to operating systems. However, user-level threads require non-blocking systems calls; otherwise, the entire process will be blocked.

2.1.6 Load Balancing in Multi-core Programming

To give full play to the potential processing power of each processor core in a multi-core processor, it must be ensured that tasks assigned to each CPU core have better load balancing performance. Otherwise, some CPU cores are running, while other CPU cores are idle and they cannot fully exploit the advantages of multi-core CPUs. There are usually two schemes for achieving the right load balance: static load balancing and dynamic load balancing [7]. In static load balancing, it is necessary to manually divide the program into mul-

multiple parts that can be executed in parallel. It is essential to ensure that the divided regions can be distributed evenly to each CPU, that is, the workload is to be even among multiple tasks. Dynamic load balancing is in the process of the operation of the task to accomplish the purpose of load balancing. It is to dynamically migrate the tasks from heavy load processors to light processors based on the task load of the current processor.

There are many challenges in reality that remain unsolved with static load balancing [13]. For example, in a large loop, the number of loops is input from the outside, and this number is uncertain in advance. Implementing a static load balancing strategy at this time is complicated. The system implements the task scheduling in dynamic load balancing. Programmers usually can only choose dynamic and balanced scheduling strategies, but they cannot modify the scheduling strategy, because there are many uncertainties in the actual task, the scheduling algorithm is complicated to do the best [7].

This phenomenon is more apparent when multiple applications run together. Therefore, this thesis proposes a method for thread scheduling among various forms to improve overall performance. We show the benefits of user-level thread spatial scheduling in Argobots. The primary goal of task scheduling is assigning multiple tasks to multiple processors. In multiprocessors, to maximize processor utilization, each processor must be fully utilized to achieve load balancing. We must avoid learning one or more processors idle, while other processors are in a high load state.

Argobots uses local queue scheduling, where each processor creates and maintains a local task queue. The advantage is that each processor usually only accesses the private local task queue, and the shared task of the local task queue is not high so that the parallelism can be improved. The disadvantage is that it causes the load of each processor to be unbalanced.

2.2 Related Work

Pan H [18] provides a system which enables efficient composition of parallel libraries. The authors believe the parallel software industry plays an essential role in multi-core system implementation. They define a resource API standard to increase the parallelization.

Fluet M and Rainey M [10] introduce a new heterogeneous language multi-core project to implement parallel programming. The author combines standard meta language-style concurrency, with implicit threads and parallel constructs. This language supports shared state.

Kale LV [14] proposes that most parallel language designing communities reject the existing package although there are many free thread packages on the internet. He describes the design and fundamental principles of the converse threads system, which supports interoperating for multiple parallel languages.

Fluet M [11] describes a new scheduling manycore framework for general-purpose parallel languages. He takes the micro-kernel approach to realize the scheduling platform. The platform supports several different scheduling strategies, especially for nested scheduling program.

It is essential that parallel applications run together on the same machine. The multiple applications do not achieve the excellent performance in the current framework because the applications cannot adapt well to the changed number of kernels. Focus on these shortcomings, Nickolls J [12] introduces Callisto, which is a co-scheduling parallel runtime system. It shows the performance when two applications work together, one is for OpenMP computing and the other is for a task-parallel program. The parallel runtime system dynamically partitions the contexts between jobs. Each part has its hardware, while the idle contexts are available.

In this thesis, we focus on multiple co-located multi-threaded applications working together. We decrease the conflict between different applications in the runtime to reduce the overhead caused by context switching and achieve a performance improvement between 10% to 20%, specifically for multiple applications. To summarize, our research reveal the benefits of a spatial arrangement for Argobots applications.

2.3 Motivation

The core of software development in multi-core environments is multi-threaded development. Multi-threaded programming techniques can improve system and program performance, such as throughput, calculation speed, response time and so on. Therefore, high performance and high efficiency are one of the goals of multi-threaded program parallelization. However, in many cases, the parallel program cannot achieve the expected performance. Many reasons affect performance such as the OpenMP parallelization overhead, dynamic migration of threads among CPU cores, load balancing, and thread synchronization overhead.

We assume that all the applications are independent, that is, they do not communicate with each other and there is not shared data because this is the common model in Open Computing Language, Cuda, or Intel Threading Building Blocks. When multiple applications work together, they share the same core resources, because the Argobots applications are bound to the same hardware. This can cause a significant conflict among different co-located applications, and this overhead increases as the number of co-located applications raises. The main idea to decrease overhead among various applications is to separate the hardware resources of the different applications. When several applications

share the same core resources, we logically separate the cores into several partitions, and each application uses a distinctive set of core resources. This method decreases the conflict caused by context switching among applications.

In this thesis, we focus on improving computational performance through dynamic migration of threads between CPU cores and work sets between different applications and demonstrate the benefits of spatial scheduling in the Argobots applications.

Chapter 3

Argobots Runtime Architecture

3.1 Argo Architecture Layout

Argo is an exascale operating system and runtime research project to support large-scale calculations [22]. It can fundamentally solve the problems faced by high-performance computing (HPC) operating at vast scale. Argo design supports hierarchical views of the entire exascale system. Figure 3.1 shows the organization and central component of the Argobots runtime system. The Argo architecture is divided into four parts [22]: the Node Operating System(NodeOS), the Argo runtime, the Global Operating System(Global OS) and the Global Information Bus. Part 1. The NodeOS is the operating system running on each node of an Argo machine. It is based on the Linux kernel, tuned and extended for HPC use on future architectures [8].

Part 2. Argobots is the runtime component of Argo, a lightweight low-level threading framework. It implements low-level threading entirely in user-space. Meanwhile, the Argobots runtime provides an interface for higher-level

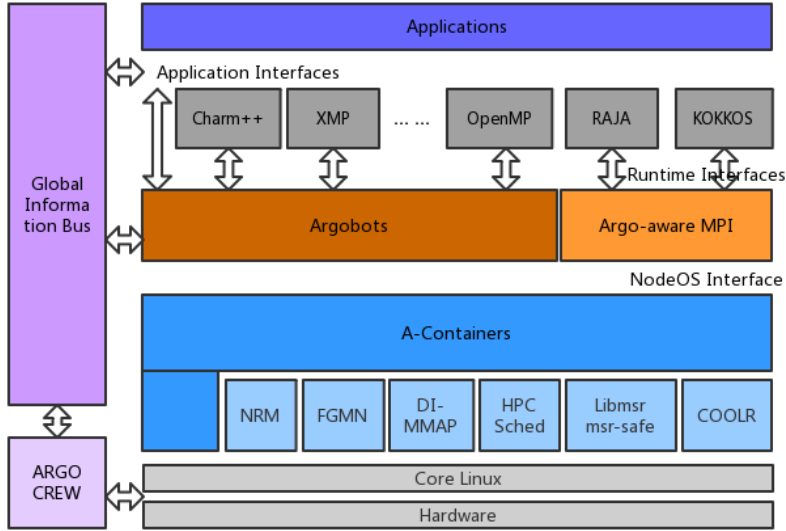


Figure 3.1: Argo architecture layout.

applications and libraries.

Part 3. The Argo Crew is a hierarchical infrastructure which is a component of Global OS.

Part 4. The Global Information Bus is a scalable infrastructure focusing on providing common interfaces for communication and management of system data across all layers of the Argo stack.

3.2 Argo Runtime: Argobots

Argobots is the underlying infrastructure that supports lightweight threads for large-scale concurrency. It utilizes hardware and the lowest level structure in the system through a data movement engine, a light-weight notification mechanism, a data placement strategy, and memory mappings [22].

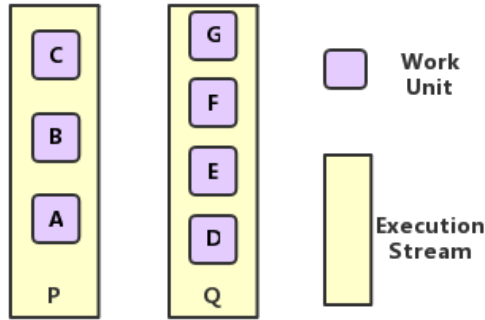


Figure 3.2: Relationship of execution streams and work units.

Argobots supports two levels of parallelism: execution stream parallelism and work unit parallelism. The execution stream is bound to the specified hardware resource, and it is a sequential instruction stream composed of work units. A work unit is a lightweight execution unit such as a user-level thread or tasklet. They are associated with a function call and complete the specific task of a work unit. Work units are bound to an execution stream. This execution stream runs all work units according to the relevant scheduling strategy. In other words, in an execution stream, work units are executed serially. In different execution streams, the execution of work units implements as parallel execution. For example, as shown in Figure 3.2, work units A, B, and C are bound to execution stream P, and work units D, E, F, and G are bound to execution stream Q. Work units A, B, and C run one by one, so they are sequential. However, work units A and D can run at the same time, so they are parallel.

3.2.1 User-Level Threads and Tasklets

Work units support lightweight concurrent execution and are associated with function calls and can dynamically deal with the needs of applications and hardware resources. User-level threads and tasklets are the smallest independent

execution units of a program schedule. The output of tasklets depends only on their input data and is usually produced when the tasklets complete. The difference between the two is that a user-level thread has a private stack and a tasklet does not. Tasklet borrows the scheduler stack of the host execution stream. This thesis only covers user-level threads with independent stack areas, hence we focus on user-level threads. Information about tasklets can be found in [22].

User-level thread scheduling. User-level threads can be independently scheduled according to different scheduling policies. Each user-level thread has its associate execution stream and scheduler, that is, once the user-level thread is ready to execute, it will be scheduled by the scheduler. Moreover, at a specific point in time, only one user-level thread can run in the execution stream. In different execution streams, the user-level threads execute in parallel. User-level threads are yieldable and can be explicitly yielded using their handles. User-level threads can make blocking calls, but users must be careful. If a user-level thread is blocked without yielding, it will block the entire execution stream.

User-level thread types. There are two types of user-level threads in Argobots, primary and secondary. The primary user-level thread is created by the Argobots library when initialized. It cannot be migrated to other execution stream; when the primary user-level thread terminates, all other user-level threads terminate. The primary user-level thread creates secondary user-level threads, which can be relocated to other execution stream or pools.

User-level thread states. There are four classes of a user-level thread states. `READY` is the initialized state of user-level thread when created. When execution stream's scheduler executes it, it changes to `RUNNING` state, and later to `TERMINATED` after finishing the associate work. The `RUNNING` state is changed to `TERMINATED` when there is an `exit` or `cancel` call in the

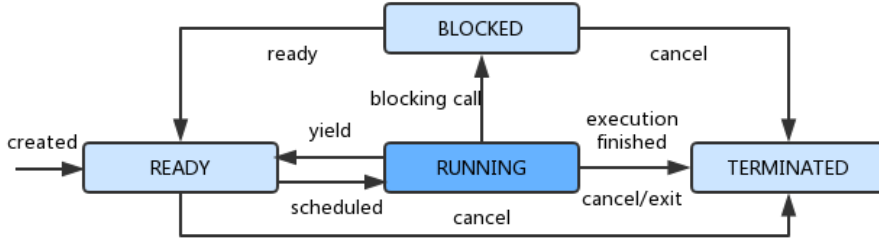


Figure 3.3: User-level thread states.

system, BLOCKED when there is a blocking call, or READY when it is yielded. Figure 3.3 shows the states of user-level threads and the associated conversion relationships.

3.2.2 Execution Streams

An execution stream is a sequential instruction stream of execution. It is a container of work units, especially user-level threads, in this thesis. Execution streams support interleaved execution of work units, but because the execution stream is a sequential instruction stream, work units cannot execute in parallel. However, work units can run concurrently in different execution streams, because they are bound to the separate hardware resources.

Execution stream types. Argobots includes two types of execution streams: primary and secondary. The primary execution stream is created by the Argobots architecture when Argobots library initialized; users cannot handle it. The secondary execution streams are created by the primary execution stream. Only one primary execution stream exists in one application, and when it starts, it contains one implicit user-level thread that executes the main function. Secondary execution streams are created at runtime by any work unit. The primary

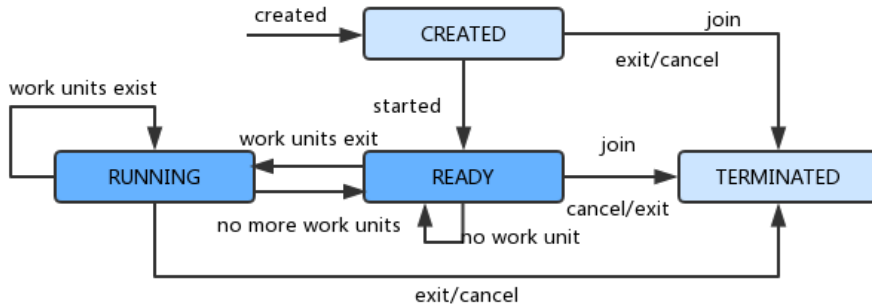


Figure 3.4: Execution stream states.

execution stream is the parent of all the secondary execution streams; when it terminates, all secondary execution streams terminate. However, terminating a secondary execution stream does not affect the other execution streams. Figure 3.4 shows the state of the execution streams and the associated conversion relationships.

Execution stream states. There are four states of execution streams: CREATED, READY, RUNNING, and TERMINATED. When an execution stream starts, it turns from CREATED to READY. If work units exist, it switches to the RUNNING state and executes the task. Once the work unit has completed, it goes back to the READY state. Only exiting or canceling an operation can turn it to the TERMINATED state, and this is an irreversible operation.

3.2.3 Scheduler and Pool

The scheduler is the controller of an execution stream. Each scheduler is associated with a particular execution stream; it controls the execution order of

scheduling units in this execution stream. Scheduling units include work units and other schedulers. Also, a scheduler has different scheduling strategies. In Argobots, the user can define the user scheduler and change the scheduler or the scheduling strategy at runtime. The scheduling units can migrate from the associated scheduler.

Scheduler Types. There are two types of schedulers: basic schedulers and user schedulers. The basic scheduler is created by Argobots when it is initialized. It supports three types of scheduling strategies: first In first out (FIFO), last in first out (LIFO) and priority. FIFO is the default scheduling strategy. Users can also implement their personal scheduler and associated tactics. In this thesis, we focus on the FIFO scheduling strategy.

A pool is a container of work units. The data structure of a pool can be implemented as a FIFO queue, a LIFO queue, or a tree. In this thesis, we focus on FIFO queues. In our Argobots research, each execution stream is associated to two pools, a private pool and a shared pool. The private pools store the primary user-level thread and basic schedulers. The shared pools store the secondary user-level threads and user schedulers. Only data structures in shared pools can be relocated.

Relationship between execution streams, pools, schedulers, and cores. The execution stream is bound to the target CPU core; it only runs on the associated hardware resource. Each execution stream has its scheduler and the related pools. When the scheduler is running, according to the scheduling strategy, migrate the work unit from the pools to the execution stream. The state of the work unit turns from READY to RUNNING. The picture 3.5 shows the relationship between them.

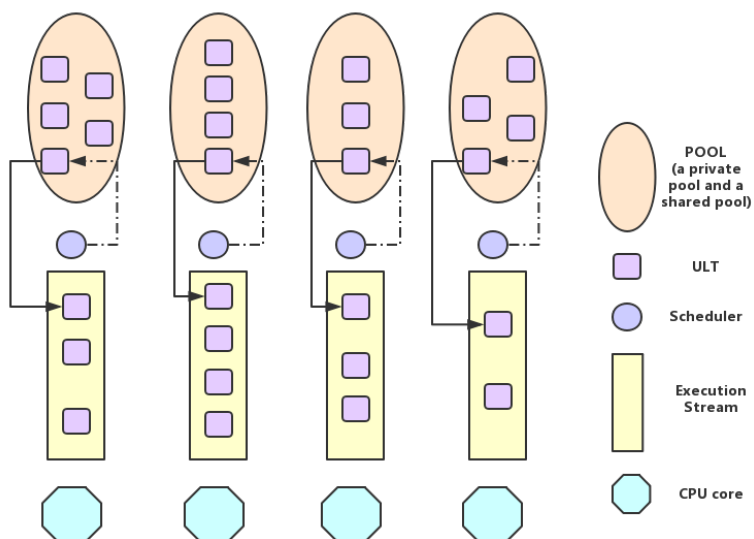


Figure 3.5: Relationship between execution streams, pools, schedulers, and cores.

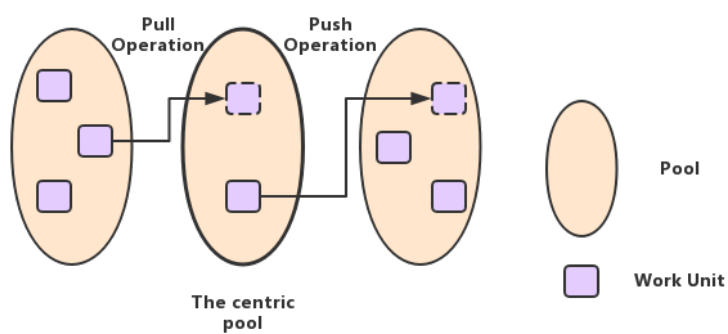


Figure 3.6: Push and pull operation.

3.2.4 Work Unit Migration

Work units can migrate between execution streams or pools. All user-level threads can migrate except the primary thread or threads in the TERMINATED state. But, only READY user-level threads can relocate to other locations. The migration operation includes two types: work unit-centric migration and location-centric migration.

Work unit-centric migration refers to migrating one work unit from a pool to another pool or execution stream. It includes two operations: the pull operation and the push operation. Figure 3.6 shows the process of pull and push operation. If an execution stream or pool wants to migrate a work unit from other execution stream or pool to its location, this is a pull operation. Otherwise, if an execution stream or pool intends to migrate the work unit associate with itself to another position, it is called push operation.

Location-centric migration is relocated the local work units. We focus on the migrated location rather than migrated work units. That is, if we want to empty a target pool, we use location-centric migration. In this way, all the work units who belongs to this pool are migrated to other execution streams or pools.

Chapter 4

Implementation

The cores in a multi-processor execute their code when the application is running, but different cores need to share and synchronize data. We assume that all the applications are independent, that is, they do not communicate with each other and not share data. Co-located applications share the same resource, because the Argobots applications are bound to the same hardware. There can be a massive conflict between different co-located applications, and this overhead can increase if the number of application raises. The main idea to decrease overhead between various applications is separating the hardware resource of different applications. When several applications share the same core resources, we separate the hardware into several partitions, and each application uses the distinct core resources. This method decreases the conflict caused by context switching among applications.

To achieve multi-processor load balancing of task scheduling, the most critical point is to determine the load standard. This thesis introduces “number of READY work units in the queue” as a load indicator. If the number of running

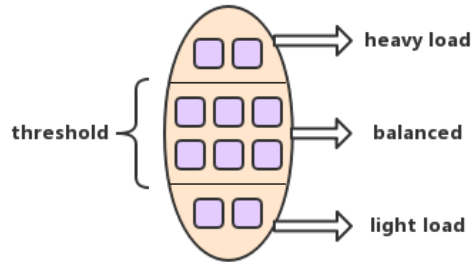


Figure 4.1: Relationship of thresholds and work load.

tasks in queues is higher than an upper threshold, we say the load is heavy. If that number is less than a lower threshold, the load is light. So we need a threshold which has an upper limit and a lower limit. Figure 4.1 shows the relationship between thresholds and work load. We use this threshold to balance the work units. If the scope of the load balancing is too narrow, the effect of load balancing is not achieved; if the range of the load balancing is too broad, the balance overhead is too high and can affect the system performance.

In general, load balancing includes static task scheduling and dynamic task scheduling. Static task scheduling allocates to each processor according to the characteristics of the task and the load of the processor to achieve load balancing before the application runs.

In dynamic task scheduling, migrated the tasks from heavily loaded processors are dynamically to lightly loaded processors based on the current work load. Dynamic task scheduling is divided into four steps:

1. Load estimation. Calculate the work load of each processor and check for heavily loaded processors and lightly loaded processors.
2. Migration decision. For each processor, decide whether to perform task

migration according to the load status.

3. Task selection. Select the task to migrate by not only considering the load state before immigration, but also the state after balancing, that is, the difference of their load. If the heavy load processor changes to the light load or light load become heavy, canceling the operation is necessary.

4. Task migration. Implement task scheduling. Static task scheduling focuses on the first assignment of tasks. After task dispatching completes, it no longer demands intervention. The implementation is manageable, but the flexibility is not as excellent as dynamic task scheduling. Dynamic task scheduling has strong effect on runtime performance. Due to the need to collect tasks and kernel status in real time, the overhead is needs to be kept to a minimum.

In this thesis, we employ combined load balancing of static task scheduling and dynamic task scheduling. To achieve co-locating of various applications in multi-core processors task scheduling, we modify basic data structures of BOLT and the Argobots environment. Also, we add another monitor program, change the contents of the task queues, add a balancing mechanism, redefine the execution streams and pools states, and set thresholds to achieve better dynamic task scheduling. Moreover, for co-located applications, we introduce a controller which is responsible for work unit scheduling between applications, reduces the conflict and overlap between instances, and increases performance.

4.1 Data Structures

We introduce and modify some data structures and variables of the original Argobots runtime:

Es_state records the execution stream state in the application under the Argobots structure. It can be accessed and updated by the monitor, which is

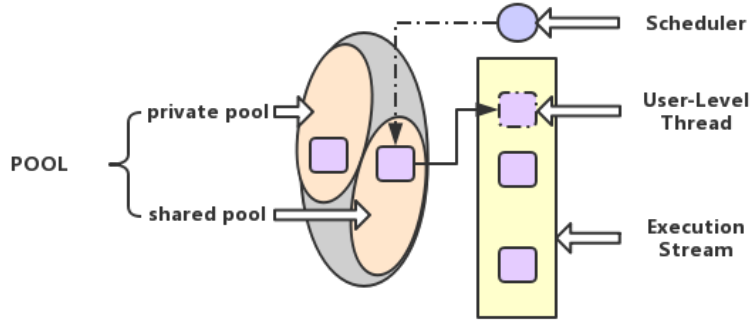


Figure 4.2: The composition of a work set.

another mechanism we introduce.

Pool_state. Record the status of pools in one application. In the scope of this thesis, each execution stream corresponds to two pools, one private and one shared. When recording the pool state, we consider the two pools as a whole part.

Work set (WS). The new unit of operation for redefining load balancing, consisting of an execution stream and the associated pools. Figure 4.2 shows the structure of a work set.

WorkSet_state. Record the global state of the work set. Because in Argobots environment, such a work set is bound to a particular hardware resource.

Core_distribution. Generated by the controller and assigned to the corresponding application, records the contents of the core distribution information.

Decrease_list. Used in a single application to store the work set which needs to reduce tasks when load balancing.

Increase_list. Used in a single application to store the work set which needs to increase tasks when load balancing.

Empty_list. Used in multiple applications to store the work set which needs to be decrease or empty tasks when load balancing.

4.2 Work Load Balancing

From Chapter 3, we know that Argobots supports work unit migration. The primary scheduler dispatches the work units to pools, and the associated execution stream run them. But, at runtime, the work units that are distributed to the pools are not balanced. Once the pool is empty, there are not work unit in the execution stream, the execution stream state turns to READY; meanwhile, the associated core is idle, wasting hardware resources. So, we introduce a mechanism that monitors the pools and execution streams to balance work units at runtime.

In the Argobots framework, the introduced global variables `es_state` and `pool_state` are used by the monitor program. The monitor program checks the status of all work sets in the application. For the execution stream, the Argobots runtime provides APIs for querying the status of the execution stream. The monitor can track the execution stream through the scheduler to get the state information. For pools, the monitor judges whether it is empty or not. Figure 4.3 shows the flow chart of work load balancing.

The purpose of the balancing mechanism, in short, is to bring each execution stream state to a relatively balanced state while the application is running. In other words, the number of work units in each execution stream and pool needs to be as balanced as possible. As we know, it is impossible to make the execution time of each work set uniform, and it is also impossible to control the execution time of a work unit because it is input dependent. However, we can balance the load by adjusting the number of work units. Assuming a terrible situation, the

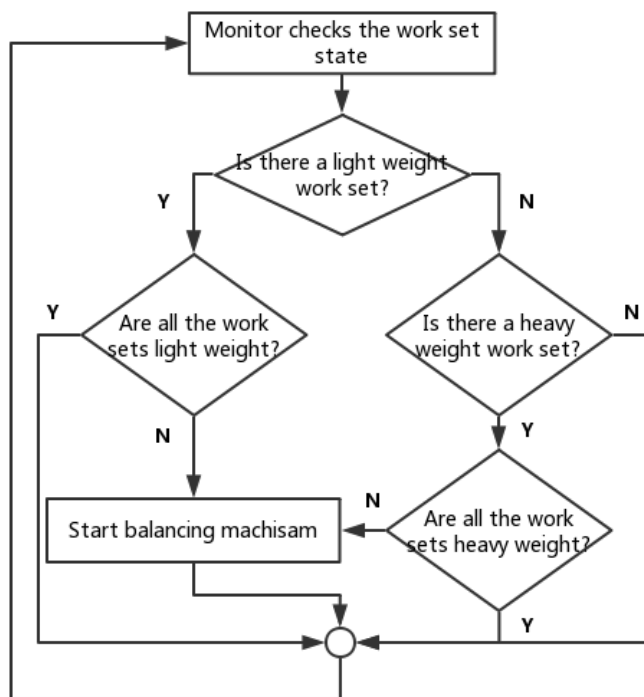


Figure 4.3: Flow chart of work load balancing.

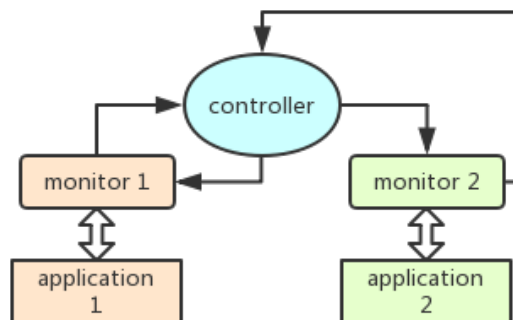


Figure 4.4: The relationship between controller and monitors.

number of work units in each work set is balanced now, but one of the work unit in a work set needs a very long execution time, that is, there is no absolute balance. When this work unit is in the RUNNING state, many other work sets have completed several work units; and the monitor executes the load balancing operation to re-balance. Therefore, it can be considered that the balance of work set can reach the load balancing requirement.

4.3 Choosing the Threshold

The rule of the original execution stream state is simple: After Argobots is initialized and there is a work unit, the execution stream is in the RUNNING state; otherwise, it is in the READY state. Also, for the pool, if there is a work unit, it is not-empty; otherwise, it is empty.

Firstly, introduce a threshold. Since each work set is bound to the hardware, redefine that, if the number of work units in the pool is higher than the upper limit of threshold, the work set is overloaded. Conversely, if this number is less than the lower limit of threshold, the work set is in light load. Secondly, redefine the work set state. In a work set, if execution stream is in RUNNING state and pool is overload, we define the work set as an overloaded work set or heavy load work set. Otherwise, it is a light load work set, or it is in equilibrium.

In the case that a work set is in a lightly loaded state and not empty, it is expected to complete soon. Moreover, once finished, the monitor recognizes it and then starts the work unit balancing. It is not necessary to wait until the pool empties to let the controller detect the state transition, which saves the communication time and initiation time of load balancing. In short, the monitor can notify the state of work sets proactively.

How does the monitor accomplish load balancing? After the monitor re-

ceives the message of each work set, it perceives the application is under the unbalanced state. In this situation, initiating the load balancing mechanism is necessary. On one hand, if the work set is overloaded, the monitor adds this work set to the Decrease_list. Then it starts to migrate several work units from this work set to other positions by the push operation until it reaches a balanced state. At this point, the state of the target work set may vary. One difficulty that needs attention is that after the push operation, if the target work set state turns to an overloaded state, a cancel operation is essential. On the other hand, if the work is light, add this work set to the Increase_list. The monitor starts to migrate several work units from other work set into this location using the pull operation until it is in a balanced state. The target work set's state is another necessary point to pay attention. If the target work set changes to light load, also cancel the operation.

Several details need to be considered:

- The frequency of the monitor to check the work set. A too fast frequency will increase the runtime overhead, on the contrary, a too low frequency will not reflect the real-time status of the application and the load balancing mechanism will not achieve great results.
- When all the work sets of an applications are overweight, the monitor cannot start the load balancing mechanism. Because once it starts, the movement of work units between work sets will be useless. If it forces one or several work sets to be balanced, the other work set will be in a more massive loaded state. Therefore, when all work sets are in the overload state, it does not start the load balancing mechanism.
- When all the work sets of an application are in a light load state, the monitor does not start. Because at this time, each work set will complete the task quickly. No matter how they migrate, they cannot reach an

equilibrium.

4.4 Core Allocation

Chapters 4.2 and 4.3 consider balancing the load in one application to improve performance. In this part, we focus on the co-located applications. Consider a situation in which two or more applications use the same core resources at the same time, and the application seizes hardware resources through context switches. User-level threads used by Argobots are lightweight threads which cost little when context switching. However, when context switches are applied to threads of different applications, user-level thread lose their advantage as lightweight threads. When a context switch occurs, the current operating state should be stored firstly, which includes the user-space stack and register information, so that the thread can continue after switching back. Switching and communicating frequently will reduce performance, especially for multiple applications. The purpose of this part is to minimize this overhead. To easily understand, we use two parallel applications as an example. The situation of three or more instances is similar, and in the next chapter, the related evaluation result and analysis will be listed.

Firstly, set a control program to communicate with two applications. This control program can reallocate the hardware resource, especial the CPU cores, to applications. Figure 4.4 shows the relationship between the controller, the monitors, and the applications. The monitor resides in the Argobots application, and is aware of all the work set states in the application. The controller recognizes all the application states. When the Argobots applications are running, the monitor sends all the work set information to the controller. The controller receives them and reallocates the cores to the Argobots applications.

Once the application receives the new distribution, it adjusts the execution location of its work sets and the associated CPU cores. Meanwhile, it also sends the work set state to the controller periodically. Figure 4.5 shows the flow of core allocation.

The principal idea to reduce the overhead among various applications is separating the hardware resources of different applications. In this situation, there are two co-located applications, we set each application to use half cores. As mentioned before, when the two applications start initializing, the monitor acquires the status of each work set in real-time and then transfers them to the controller. After the controller obtains the information, it redistributes the core set of each application, assigns one of the applications to use the distinct CPU core, and the other application uses the remaining. The above is the implementation part of the static load balancing mechanism of this thesis.

In related research, the most simple and easy way to realize is the half-distribution principle, that is, two instances share the resources, each instance uses half of the hardware. After each instance receives a different core distribution, it updates the `empty_list`. If a work set is in the `empty_list`, the monitor pops the whole work units in it and then pushes them to other locations which are not in the `empty_list`. After reallocating the cores for multiple applications, each application has separated cores to continue the job. The monitor always balances the load inside the application until it terminates. The monitor and controller exchange real-time information in each instance.

In reality, applications do not usually start and finish at the same time. Therefore, when one of the instances completes, its resources are freed, and only the other half of the hardware works. This is a massive waste of resources. What needs to improve is reallocating the hardware resources after one or more instances complete while others are still running. The monitor transmits

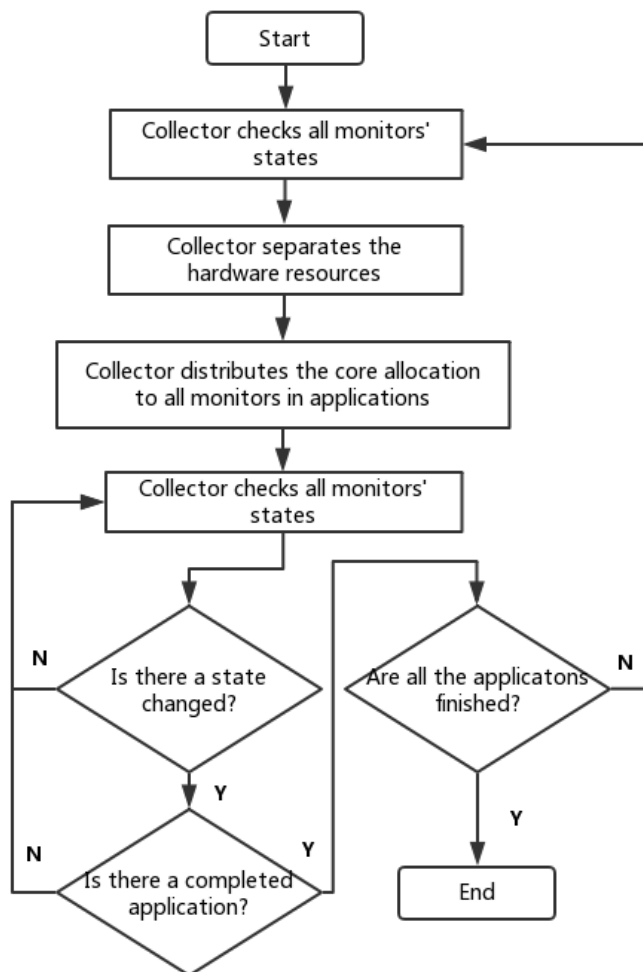


Figure 4.5: Flow chart of core allocating.

the application state to the controller periodically. Once the controller receives a completion message, it summarizes all the current data and reassigns the hardware resources to the working instances.

How to decide which group of CPU core is assigned to which application? In the implementation of this thesis, we use a Linux system with 40 cores, and these 40 cores are allocated to two Non-Uniform Memory Access architecture (NUMA) nodes. Therefore, when assigning CPU core to an application, the work sets of the same application should be allocated to the same NUMA node [6]. Since in a multi-application structure, when one or more applications finish, the controller should disrupt the original allocation structure and redistribute the cores according to the current state. Of course, it requires more interaction and allocation time than directly assigning. However, it is more advantageous to disrupt and redistribute the location if the running program needs a long time to complete. Based on this, we determine the disrupt and reallocation mechanism.

Pipe communication between controller and applications. So far, the fundamental operating mechanism of multi-application has been introduced then we concentrate on how to implement the exchanging arrangement. We use a pipe to communicate between the controller and the monitors. A pipe is a particular file that facilitates communication between processes. A notable feature is when we create a pipe, it will obtain two file descriptors for reading and writing. The monitors and the controller operate one port respectively.

In this chapter, we already introduced the basic implementation of our research. The optimized Argobots architecture uses a combination of dynamic and static load balancing to reduce the overhead. The single-application load balancing is implemented through work units migration, meanwhile the multi-application load balancing is achieved through work sets movement.

Chapter 5

Evaluation Results and Analysis

5.1 Benchmarks

We examine the original Argobots and the improved runtime and then compare and analyze the results using the NAS Parallel Benchmarks (NPB) suite, a set of benchmarks targeting performance evaluation of highly parallel supercomputers [6].

For the evaluation of this thesis, the following five benchmarks are used: BT, CG, EP, SP, UA. Details of these benchmarks are given in Table 5.1. There are different input data classes for each benchmarks: class A, B and C. The size of class A and B are too small to be suitable for comparing the improvement, so we use input class C for all benchmarks. Table 5.2 describes the different sizes of the five benchmarks we use in the evaluation.

The evaluated metric is the total turnaround time, that is, the runtime of the application under Argobots structure from start to completion. The shorter the turnaround time, the better the performance and the higher the performance

Table 5.1: Benchmarks used for the evaluation.

Benchmark	Name	Description
CG	Conjugate Gradient	Estimate the smallest eigenvalue of a large sparse symmetric positive-definite matrix using the inverse iteration with the conjugate gradient method as a subroutine for solving systems of linear equations.
EP	Embarrassingly Parallel	Generate independent Gaussian random variates using the Marsaglia polar method.
SP	Scalar Pentadiagon	Solve a synthetic system of nonlinear PDEs using three different algorithms involving block tridiagonal, scalar pentadiagonal and symmetric successive over-relaxation solver kernels, respectively.
BT	Block Tridiagonal	
UA	Unstructured Adaptive	Solve heat equation with convection and diffusion from moving ball. Mesh is adaptive and recomputed at every 5th step.

Table 5.2: Input data size per class for the different benchmarks.

Benchmark	Class A	Class B	Class C
CG	14000	75000	150000
EP	2^{28}	2^{30}	2^{32}
SP	64^3	102^3	162^3
BT	64^3	102^3	162^3
FT	128×256^2	512×256^2	512^3

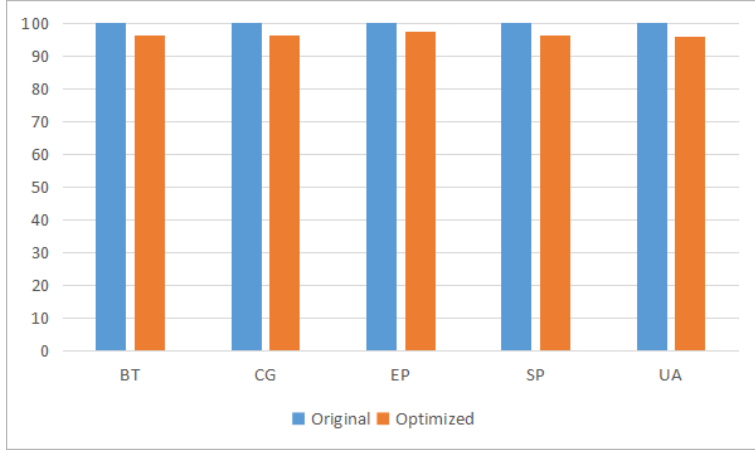


Figure 5.1: Evaluation results of single application.

improvement. Corresponding to the last chapter about the implementation, the evaluation includes three parts: work unit balancing, the threshold, and core allocation.

5.2 Evaluation of Work Load Balancing

In this part, we evaluate the Argobots runtime with a single application, and compare the performance of the individual program under the original Argobots and the optimized Argobots runtime.

Figure 5.1 shows the normalized execution time of a single application. In this figure, the x-axis shows the different classes of benchmarks; the y-axis is the normalized runtime of a single Argobots application. The blue bar represents the original Argobots application runtime, and the red bar represents the optimized Argobots application with the work set balancing mechanism. We can accomplish a 2% to 3% improvement compared with the original Argobots runtime.

5.3 Evaluation of Different Thresholds

Selecting a reasonable load, dividing the heavy load and the light load, and determining the threshold are essential for load balancing. If the scope of the threshold is too narrow, the system is inefficient. On the other hand, if the range of the threshold is too broad, the balancing overhead is higher, affecting the system efficiency. Therefore, a proper threshold is needed. When selecting the lower limit, we assume there is no upper limit, which means that the load balancing mechanism only starts after detecting a specific workload imbalanced. Similar to the upper limit, thinking there is only the upper limit of the threshold, that is, we start balancing once receiving information about extensive load information. We set the upper and lower threshold values, respectively, to find the best combination for performance.

Figure 5.2 shows the evaluation result of different thresholds. We select a set of the upper limit and lower limit of the threshold, which has the best performance. As we can see from the figure, the bar in baby blue shows the best performance, which costs the lowest turnaround time. Hence, in the following benchmarks, we use this number as the threshold, the lower limit is 17, and the upper limit is 40.

5.4 Evaluation of Core Allocation

In this part, we examine the performance of a different number of instances working together. Then we compare them with each other, and also with the performance under the original Argobots runtime.

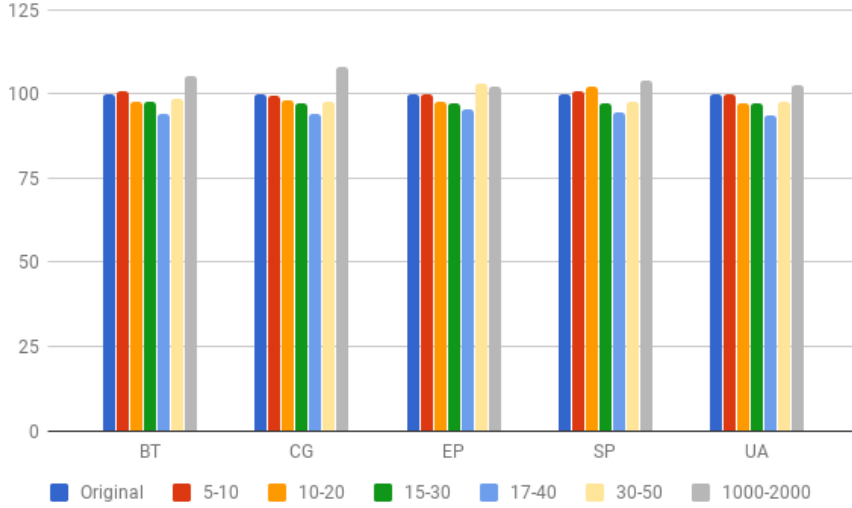


Figure 5.2: Evaluation results of different thresholds.

5.4.1 Executing Two Parallel Applications

Figure 5.3 shows the evaluation result of two co-located parallel applications. The graph provides some interesting data regarding the performance of various applications. Two applications consist of one set, and one benchmark runs in each application. For example, the third set in this figure is CG and EP. There are two applications that run together; one executes the CG benchmark, the other runs EP. Meanwhile, each benchmark includes five bars; these represent turnaround time for the original Argobots, the Argobots with work set balancing, Argobots with internal application load balancing mechanism, Argobots with core allocation, and Argobots with internal application load balancing and external application load balancing separately.

- The first bar, **Original** Argobots bar exhibits the application runtime under original Argobots architecture.

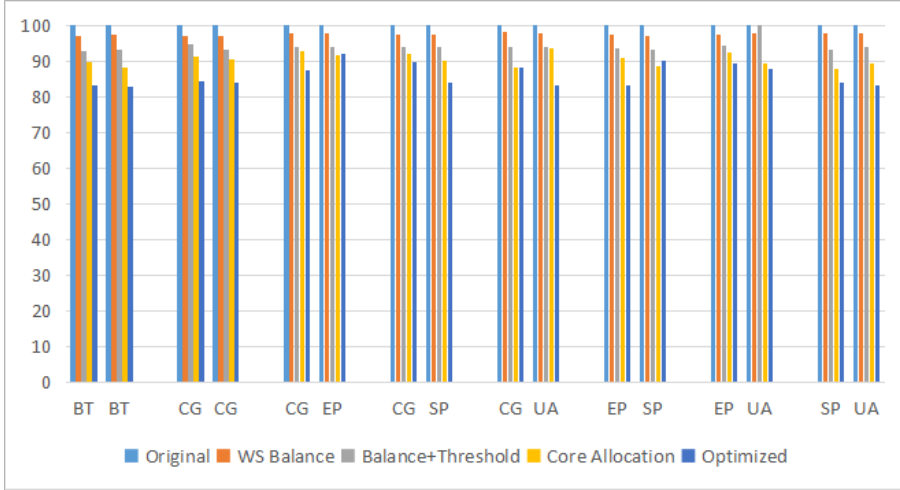


Figure 5.3: Evaluation results of two co-located parallel applications.

- The second bar, **WS balance** bar shows the performance of Argobots with internal application load balancing, but without a threshold. Because there isn't a threshold, when the application is running, monitor balance the work set to the same size. However, it is unbelievable to realize it, the size of work set relies on the actual system environment. In the implementation process, the monitor makes the size of each work set relatively balanced. Due to the lack of equilibrium range, the load balancing process is precarious.
- The third bar, **internal load balancing** (balance+threshold) bar exhibits the result of a single instance under Argobots structure with internal application load balancing that has a suitable threshold. The monitor keeps the size of each work set within the equilibrium of the threshold in a program. The result represents the optimization of the single application. Compared with the original Argobots runtime, the performance of Argobots with optimized internal balancing has improved.

- The fourth bar, **core allocation** bar describes the performance of external application load balancing, which has the controller but does not have a monitor to balance the internal instance.
- The last one is **optimized** Argobots with internal application load balancing and external application load balancing for multi-application.

As is shown in the diagram, the benchmark divides into two parts, the same application benchmarks, and the different application benchmarks. It is clear that when two identical instances are running at the same time, we achieve the maximum optimization, about 12%-17%. In contrast, the optimization of two different cases is lower, reaching 8%-15%. Why does this happen? The same cases have the similar execution time, and the different applications have the different runtime. For the latter, once one of them complete, it sends the information to the controller. Afterward, the controller re-allocates the core distribution to the running application. The target instance receives the instruction, changes the internal work set. The series of operations need more extra cost than previous. Furthermore, the incomplete application has twice as many CPU cores as before, and the required runtime is naturally reduced.

5.4.2 Executing Three or More Parallel Applications

Similarly, this part of benchmark also includes two components, multiple identical applications, and different application. Figures 5.4 and 5.5 show the benchmark result of three and four co-located applications, respectively. Take four instances as an example to analyze its operation process. At startup, each application uses a quarter of the hardware resources through the communication between applications and controller. Once an instance completes, the controller reclaims this particle hardware resource, equally dispatches them to the remaining three programs. These executing instances continue to run until one of them

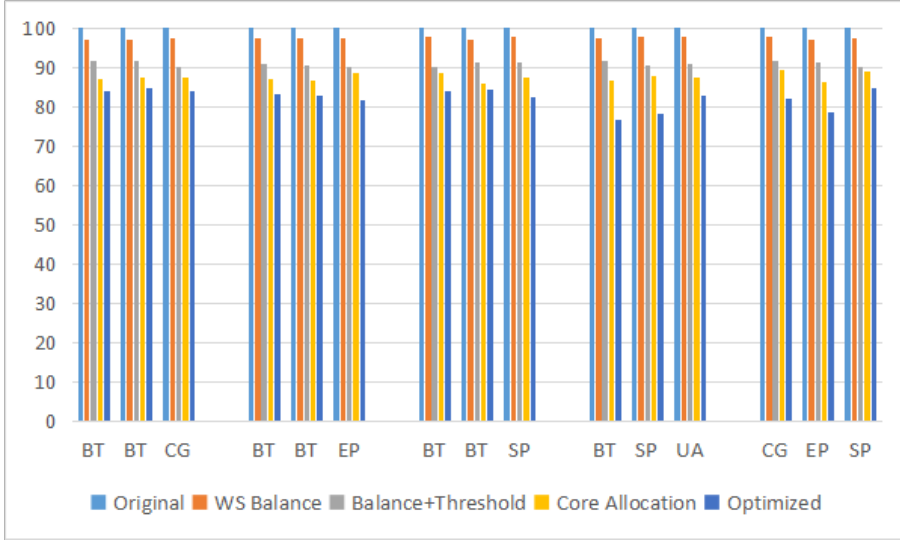


Figure 5.4: Evaluation results for three co-located applications.

finish and interchange message with the controller. The controller refreshes and sends the newest instruction to remained cases until all the applications complete.

In the next section, the benchmark uses the half hardware resource. Since the Linux server used for benchmark consist of 40 CPU cores in two nodes, only one node is used for testing while the other is idle.

Figure 5.6 shows the relationship between instances and NUMA node. Both applications run on the same node at the same time, that is, half the CPU cores. When the application runs under the original Argobots runtime, it also scrambles for resources. Compared to the previous arrangement, cases work under the improved Argobots structure; the controller divides the hardware which on one node into two parts, and each application possesses one piece.

Figure 5.7 demonstrates the result of the improved performance for one node. The increasing rates are relatively smaller than the abundant resources,

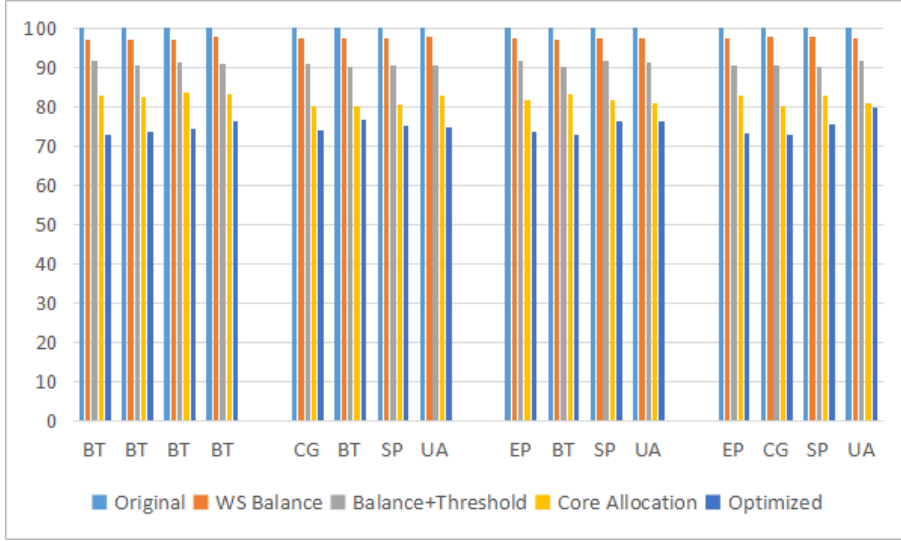


Figure 5.5: Evaluation results for four co-located applications.

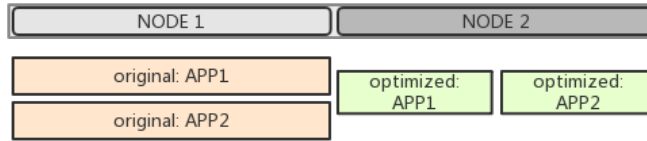


Figure 5.6: The relationship of applications and nodes.

from 2% to 8%.

It can be reasonably inferred that in the case of hardware reduction, the execution time of the program becomes longer, the conflicts time between applications raise, at the same time, the conflict location reduced. Therefore, the impact of overall performance under balancing is not as significant as the abundant resource.

In most cases, users do not care how much time each program is optimized while multiple applications are executing. They only care about how much

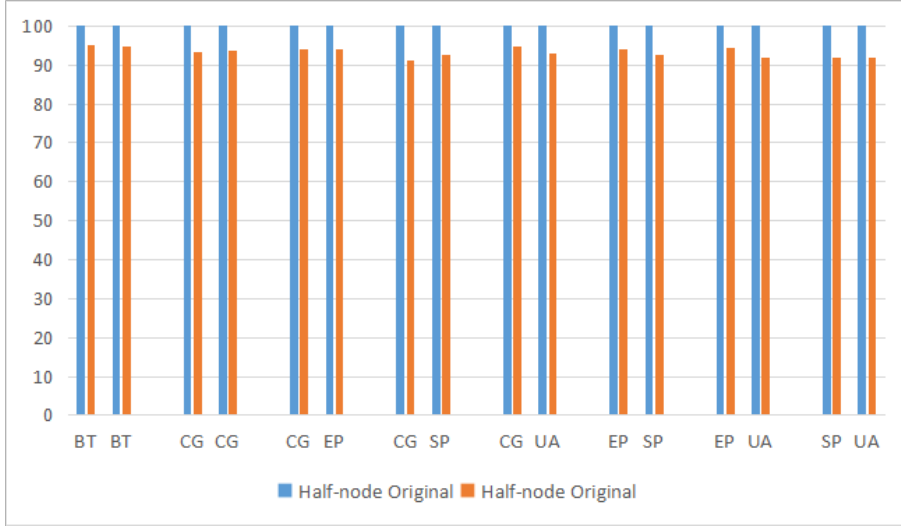


Figure 5.7: Evaluation results under half nodes.

improvement they got about the most extended instance. For example, three applications open at the same time, the runtime of A is longest. After the largest C complete, the whole task finished. Therefore, no matter how much performance is improved by the shorter A and B, users would not consider the improvement to be useful. Based on this, the benchmark in this section focuses on the performance improvement of the longest running program.

As shown in Figure 5.8, when several examples work together they scramble for hardware resources. The controller performs load balancing between various cases. Multiple-application tests two identical programs and two different programs, triple same and different instances, and four cases. The figure leads us to the conclusion that the maximum running time of many cases is shorter than the original architecture. The general trend is similar to the previous section. The optimization degree of four applications is slightly larger than three instances, more significant than two cases.

In summary, this chapter describes the detailed evaluation for improving the

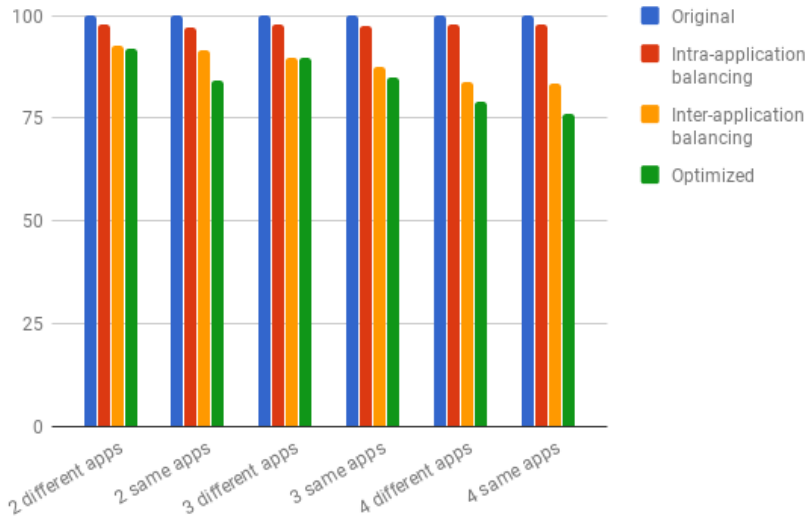


Figure 5.8: Runtime results of the longest applications.

Argobots runtime, demonstrating the results of improvement benchmark, also comparing and analyzing them. The outcome depicts the optimized structure enhance the performance of runtime, and it is evident for co-located applications. The rise in the size of the instance will increase the conflict among various application, which overcomes in optimized structure.

Chapter 6

Conclusions

In this thesis, we present the fundamental theory and the superiority of load balancing. There are two main methods to balance a work load: static load balancing and dynamic load balancing. We introduce and compare the advantages and disadvantages of them. Meanwhile, we present a modified framework based on the Argobots runtime, a lightweight framework that implements a low-level threading and task framework entirely in user space. Then we introduce a new mechanism to balance the workload of the application under Argobots architecture.

The optimized Argobots runtime uses a combination of dynamic work balancing and static work balancing to reduce the overhead. For single-application, load balancing is implemented through work units migration mechanism, and multi-application load balancing is achieved through work sets movement.

Finally, the NAS Parallel Benchmark suite is used to evaluate the performance of the improved structure. For multiple applications under improved Argobots runtime, we achieve a performance improvement of 10% to 20%, es-

pecially for multiple application. In summary, we show the benefits of spatial scheduling in the Argobots applications.

Bibliography

- [1] BOLT motivation. <http://www.bolt-omp.org/>, 2014. [online; accessed April 2018].
- [2] Kernel-level threads and user-level threads. <http://www.cs.iit.edu/~cs561/cs450/ChilkuriDineshThreads/dinesh's%20files/User%20and%20Kernel%20Level%20Threads.html>, April 2007. [online; accessed April 2018].
- [3] ARGOBOTS. <http://www.argobots.org/>, January 2014. [online; accessed April 2018].
- [4] I. Ahmad and A. Ghafoor. Semi-distributed load balancing for massively parallel multicomputer systems. *IEEE Transactions on Software Engineering*, 17(10):987–1004, Oct 1991.
- [5] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Trans. Comput. Syst.*, 10(1):53–79, Feb. 1992.
- [6] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S.

- Schreiber, et al. The nas parallel benchmarks. *The International Journal of Supercomputing Applications*, 5(3):63–73, 1991.
- [7] J. Corbalan, A. Duran, and J. Labarta. Dynamic load balancing of mpi+ openmp applications. In *Parallel Processing, 2004. ICPP 2004. International Conference on*, pages 195–202. IEEE, 2004.
- [8] A. Drebes, A. Pop, K. Heydemann, A. Cohen, and N. Drach. Scalable task parallelism for numa: A uniform abstraction for coordinated scheduling and memory management. In *2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 125–137, Sept 2016.
- [9] D. E. Dudgeon and R. M. Mersereau. *Multidimensional Digital Signal Processing Prentice-Hall Signal Processing Series*. Prentice-Hall, Englewood Cliffs, NJ, 1984.
- [10] M. Fluet, N. Ford, M. Rainey, J. Reppy, A. Shaw, and Y. Xiao. Status report: The manticore project. In *Proceedings of the 2007 Workshop on Workshop on ML, ML '07*, pages 15–24, New York, NY, USA, 2007. ACM.
- [11] M. Fluet, M. Rainey, and J. Reppy. A scheduling framework for general-purpose parallel languages. *SIGPLAN Not.*, 43(9):241–252, Sept. 2008.
- [12] T. Harris, M. Maas, and V. J. Marathe. Callisto: Co-scheduling parallel runtime systems. In *Proceedings of the Ninth European Conference on Computer Systems, EuroSys '14*, pages 24:1–24:14, New York, NY, USA, 2014. ACM.

- [13] S. Ichikawa and S. Yamashita. Static load balancing of parallel pde solver for distributed computing environment. In *Proc. 13th Int'l Conf. Parallel and Distributed Computing Systems*, pages 399–405, 2000.
- [14] L. V. Kale, J. Yelon, and T. Knauff. Threads for interoperable parallel programming. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 534–552. Springer, 1996.
- [15] S. V. Kendre and D. Kulkarni. Optimized convex hull with mixed (mpi and openmp) programming on hpc. *International Journal of Computer Applications*, 1(5):80–84, 2010.
- [16] J. Manoharan, S. Chandrakumar, and A. Ramachandran. Real-time scheduling methods for high performance signal processing applications on multicore platform, 2012.
- [17] M. A. Palis, J.-C. Liou, S. Rajasekaran, S. SHENDE, and D. S. Wei. Online scheduling of dynamic trees. *Parallel Processing Letters*, 5(04):635–646, 1995.
- [18] H. Pan, B. Hindman, and K. Asanovic. Lithe: Enabling efficient composition of parallel libraries. *Proc. of HotPar*, 9, 2009.
- [19] S. Perarnau, R. Thakur, K. Iskra, K. Raffenetti, F. Cappello, R. Gupta, P. Beckman, M. Snir, H. Hoffmann, M. Schulz, et al. Distributed monitoring and management of exascale systems in the argo project. In *IFIP International Conference on Distributed Applications and Interoperable Systems*, pages 173–178. Springer, 2015.
- [20] S. Perarnau, J. A. Zounmevo, M. Dreher, B. C. V. Essen, R. Gioiosa, K. Iskra, M. B. Gokhale, K. Yoshii, and P. Beckman. Argo nodeos: Toward

- unified resource management for exascale. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 153–162, May 2017.
- [21] L. Rudolph, M. Slivkin-Allalouf, and E. Upfal. A simple load balancing scheme for task allocation in parallel machines. In *Proceedings of the Third Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA '91*, pages 237–245, New York, NY, USA, 1991. ACM.
- [22] S. Seo, A. Amer, P. Balaji, C. Bordage, G. Bosilca, A. Brooks, P. Carns, A. Castelló, D. Genet, T. Herault, S. Iwasaki, P. Jindal, L. V. Kalé, S. Krishnamoorthy, J. Lifflander, H. Lu, E. Meneses, M. Snir, Y. Sun, K. Taura, and P. Beckman. Argobots: A lightweight low-level threading and tasking framework. *IEEE Transactions on Parallel and Distributed Systems*, 29(3):512–526, March 2018.
- [23] H. Zhong, S. A. Lieberman, and S. A. Mahlke. Extending multicore architectures to exploit hybrid parallelism in single-thread applications. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 25–36, Feb 2007.

요약

멀티코어 프로세서는 나날이 진화하고 있다. 멀티 코어 프로세서들은 대규모의 병렬 연산을 수행할 수 있는 잠재력을 지니고 있지만, 어려운 과제들 또한 가지고 있다. 특히, 모든 코어들을 바쁘게 활용하면서 동기화 비용들을 최소화하는 것은 어려운 일이다.

Argobots 는 멀티 코어 OpenMP 어플리케이션들을 위한 런타임 프레임워크이다. 이 논문에서 우리는 부하를 분산하는 방법들의 근본적인 원리와 그 효과를 Argobots 런타임 시스템을 통해서 보여준다. 우리는 Argobots 런타임 프레임워크의 소프트웨어 구조를 수정하고, 워크로드의 부하를 user-level 스레드의 공간 분할 스케줄링을 통해 적절하게 분산시키는 새로운 방법을 적용한다. 개선된 Argobots 런타임 시스템에서 정적 및 동적 부하 분산 기법들이 함께 사용된다.

우리는 NAS Parallel Benchmark 를 사용하여 개선된 Argobots 런타임 시스템의 성능을 측정하고 분석한다. 그 결과 평균적으로 10% 에서 20% 사이의 성능 향상을 확인할 수 있었다. 공간 분할 방법에 기반한 스케줄링은 여러 개의 Argobots 어플리케이션들이 동시에 동작할 때 더 눈에 띄는 성능 향상을 얻을 수 있었다. 즉, 이 논문은 Argobots 런타임 시스템에서 공간 분할 스케줄링과 부하 분산 방법의 효과를 보여준다.

주요어: 공간 배치, Argobots 아키텍처, 부하 평형, 유저 박스

학번: 2016-26674