

Chapter 2

Conventional Task Scheduling Policies

Abstract In this chapter, a survey on emerging task scheduling policies will be presented respectively. There are generally two categories of task scheduling policies: manual task scheduling policies and automatic task scheduling policies. As automatic task scheduling policies are more adaptive, we further introduce several widely-used parallel programming environments that adopt automatic task scheduling policies, such as Hadoop, Spark, Cilk, X10 etc. In the last part of this chapter, we analyze the drawbacks of existing task scheduling policies.

In contrast to the quickly development of the hardware for parallel architectures, many softwares are still not effectively parallelized and thus cannot fully utilize the powerful computational ability of parallel architectures. The requirement to utilize hardware efficiently motivates the development of parallel programming environments and parallel task scheduling policies.

There are generally two categories of task scheduling policies are used in emerging popular parallel programming environments: *manual task scheduling policies* and *automatic task scheduling policies*. In manual task scheduling policies, programmers need to explicitly schedule tasks to processing elements (e.g., nodes, cores). The most popular programming environments that use manual task scheduling policies include MPI [1] and Pthreads [2]. The manual assignment of tasks is often burdensome for developing parallel applications.

In automatic task scheduling policies, parallel programs can dynamically generate tasks at runtime, and these tasks can be scheduled between the processing elements automatically. Nowadays, most well-known programming environments, such as MIT Cilk [3], Cilk++ [4], TBB [5], Java's fork-join framework [6], X10 [7], and OpenMP [8], use automatic task scheduling policies.

When a parallel program is scheduled to run on a parallel architecture, the tasks of the program are executed concurrently when the dependency between tasks are satisfied. The task scheduling policies are used to assign the parallel tasks to run on the limited numbers of processing elements. If the workload of different processing elements is not balanced, the processing element carrying the heaviest workload would

degrade the performance of the whole parallel program. Therefore, task scheduling policy has a tremendous impact on the performance and energy efficiency of parallel system. Through optimizing task scheduling policy, we can greatly improve the performance of parallel programs without rewriting the parallel programs. In addition, because the automatic task scheduling policies can relieve the burden of parallelization and task assignment, they have become one edge-cutting research direction for both academia and industrial world.

2.1 Manual Task Scheduling Policies

Conventional parallel programming environments normally adopt manual task scheduling policies. In these programming environments, such as MPI and Pthreads, programmers need to manually balance the workload between threads/processes for the good performance.

2.1.1 Message Passing

Message Passing Interface (*MPI*) is a message-passing application programmer interface, together with protocol and semantic specifications for how its features must behave in any implementation. There are many efficient implementations of MPI, and one of the most well-known implementation is MPICH2 [1]. By using MPI together with C, C++ or Fortran, programmers can create parallel programs.

The purpose of MPI is to provide a compatible and effective message passing protocol for widely-used message passing programs. It is the first standardized and portable message-passing system. An MPI program can run on any parallel computers (CPU-based) without any modification. When MPI is first proposed, it targets distributed memory architecture. With the quickly development of computer architecture, MPI has already supports shared memory architecture. MPI has the following main characters.

- Although the programming model of MPI is proposed for distributed memory model, the execution of MPI programs does not rely on the low level hardware architecture.
- The parallelism is explicitly defined in MPI. Programmers have to identify the potential parallelism in their programs, and explicitly define the parallelism using interfaces provided by MPI library.

2.1.2 *Multi-threading*

On shared memory architecture, we can use threads to create parallel programs. In the beginning, all the hardware vendors implemented their own thread techniques respectively. Due to the diversity of thread techniques, it is challenging to develop portable and compatible parallel programs that work on different hardware. To this end, for Unix operating system, a standardized C language threads programming interface has been specified by the IEEE POSIX 1003.1c standard. Implementations that adhere to this standard are referred to as POSIX threads (Pthreads) [2].

By introducing the multi-threading method, programmers can create the parallel programs that make full use of the multiple cores. Compared with sequential model, the multi-threading programming model has the following advantages.

- It fully explores the parallelism in program and speed up the program execution.
- It supports Asynchronous I/O. When sequential program has to waiting for slow I/O operations, multi-threading program can execute other instructions instead.

2.2 Automatic Task Scheduling Policies

Compared with manual task scheduling, automatic task scheduling policies are more user-friendly. In this book, we will discuss how to design and implement efficient automatic task scheduling policies for various emerging parallel architectures. Parallel programs generally are expressed by *data parallelism* and *task parallelism*. Data parallelism is achieved when each processor performs the same task on different pieces of distributed data. And, task parallelism is achieved when each processing element executes a different thread (or process) on the same or different data. The threads may execute the same or different instructions.

2.2.1 *Task Scheduling Policies for Data Parallelism*

MapReduce is one of the most popular programming models that is used to express programs in data parallelism. MapReduce is not only a programming model, but also a task scheduling model. In MapReduce programming model, programmers can create *Map tasks* that process key/value pairs and generate intermediate data, and *Reduce tasks* that shuffle on the intermediate data to generate final results. The scheduling model of MapReduce [9, 10] is first proposed by Google and used in Cloud computing. Besides Google, MapReduce has also been extended to improve the performance of applications that can be expressed with high data parallelism [9–13].

MapReduce is suitable for the distributed processing of large data sets across clusters of computers. In MapReduce, programmers define the data processing pro-

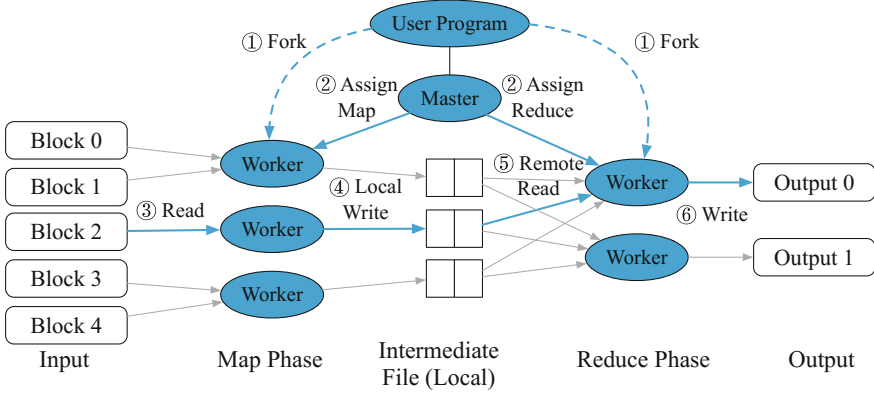


Fig. 2.1 The execution step of a MapReduce program

cedure in Map function, and define the data shuffle procedure on intermediate data in Reduce function. Programmers only need to define Map function and Reduce function to create distributed parallel programs. When a MapReduce program starts to run on a cluster/datacenter, programmers do not need to pay attention to the data splitting, allocating and scheduling. Map tasks are Reduce tasks are automatically managed by the MapReduce runtime system (e.g., Hadoop runtime system). In addition, the MapReduce runtime system is also responsible for fault-tolerance and communication managements.

Figure 2.1 shows the execution step of a MapReduce program. As shown in this figure, when a MapReduce program is invoked to run on a cluster/datacenter, it is executed in the following steps. (1) the program forks workers on distributed nodes for processing map tasks and reduce tasks. (2) the master worker of the program assign map tasks and reduce tasks to the workers housed by different nodes. (3) workers start to execute map tasks, and read corresponding data. Different workers read different data blocks (4), when a worker completes a map task, the intermediate data is written to local node to avoid data transfer through network. (5) the reduce workers then read the output of map tasks from all the nodes remotely. (6) after reduce tasks complete, the reduce workers output the final results.

It is worth noting that MapReduce uses data parallelism to speed up big data processing. In MapReduce, different Map tasks execute the same instructions but operating on different data blocks concurrently. There are low dependency between different map tasks, as well as different reduce tasks.

2.2.2 Task Scheduling Policies for Task Parallelism

Besides data parallelism, task parallelism is another widely used methodology to create parallel programs. Different from programs using data parallelism, different tasks execute different instructions in parallel programs with task parallelism. In

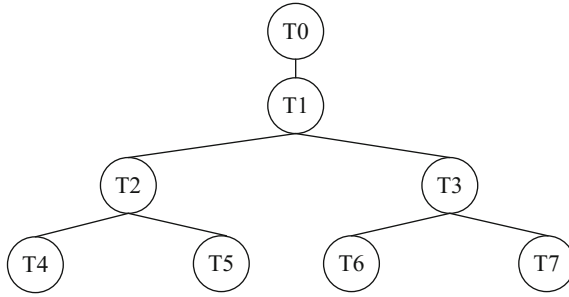


Fig. 2.2 An example of task graph (Directed Acyclic Graph, DAG)

programming environments with task parallelism, the execution of a parallel program can be represented by a task graph, which is a Directed Acyclic Graph (DAG) $G = (V, E)$, where V is a set of nodes, and E is a set of directed edges [14]. A node n_i in a DAG represents a task (i.e., a set of instructions) that must be executed sequentially without preemption. The edges in a DAG, denoted by (n_j, n_k) , correspond to the dependence relationship among the nodes.

Figure 2.2 shows an example of task graph, where T_0, \dots, T_7 represent tasks, and the directed edges between tasks show the dependency relationship between tasks. For example, task T_4 depends on task T_2 , and only after T_2 complete, task T_4 can start to run. Task scheduling policies schedule the tasks to multiple processing elements and guarantee the dependency relationship between tasks is satisfied.

For easy of description, a processing element is called a *worker*. Therefore, the execution of a parallel program can be viewed as the parallel traversal of its task graph. *Work-sharing* [8] and *work-stealing* [15] are the two most famous task scheduling strategies for programs expressed as task parallelism.

2.2.2.1 Work-Sharing

Figure 2.3 presents the task scheduling in work-sharing policy. As shown in Fig. 2.3, in work-sharing, newly generated tasks are pushed into a centralized task pool that stores all the unexecuted tasks. When a worker is free, it tries to lock the central task pool. Once the worker successfully locks the central task pool, the worker pops a task from the pool, releases the lock on the pool, and starts to execute the newly obtained task. Because the central task pool is locked when a new task is generated and pushed into the pool, and when a worker pops tasks from the pool, work-sharing often suffers from severe lock contention. Especially, when the number of workers increases, the severe lock contention would seriously degrade the system performance. The latest OpenMP [8, 16] uses work-sharing to schedule parallel tasks.

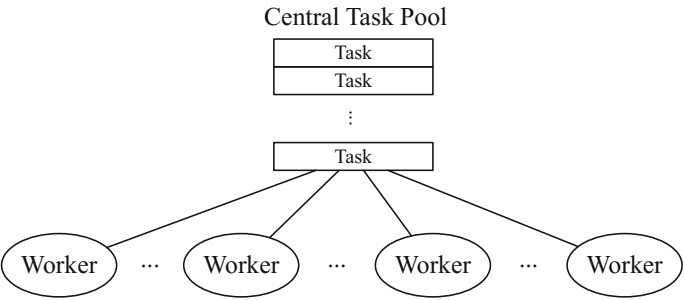


Fig. 2.3 Work-sharing task scheduling policy

2.2.2.2 Work-Stealing

In order to relieve the severe lock contention in work-sharing, researchers have proposed work-stealing policy for task scheduling. Work-stealing policy uses distributed task pool. In work-stealing policy, every worker has its own task pool. Figure 2.4 presents the task scheduling in work-stealing policy. Many programming environments and scheduling systems, such as MIT Cilk [17], TBB [5] and X10 [7] uses work-stealing policy to schedule tasks.

Most often each worker pushes tasks to and pops tasks from its own task pool without locking. Only when a worker’s task pool is empty, it tries to steal tasks from other workers with locking. Since there are multiple task pools for stealing, the lock contention is much lower than work-sharing even at task steals. Therefore, work-stealing performs better than work-sharing as the number of workers increases.

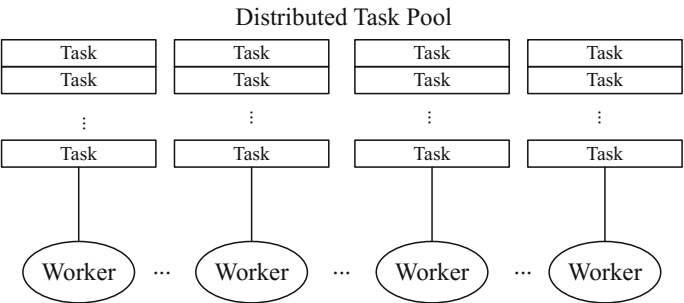


Fig. 2.4 Work-stealing task scheduling policy

2.3 Parallel Programming Environments

There are a large number of parallel programming environments have been proposed for expressing data parallelism and task parallelism. In Sects. 2.3.1 and 2.3.2, we introduce emerging widely-used parallel programming environments for data parallelism and task parallelism respectively.

2.3.1 *Programming Environments for Data Parallelism*

Apache Hadoop [18], Apache Spark [19], Apache Storm [20] (Heron [21]) are the most well-known parallel programming environments and task scheduling systems for expressing data parallelism. They have already been used in a large amount of real-world large scale clusters/datacenters.

2.3.1.1 Apache Hadoop

Apache Hadoop [18] is the most popular open-source implementation of MapReduce programming model. The Apache Hadoop project develops open-source software for reliable, scalable, distributed computing.

The Apache Hadoop software library is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models. It is designed to scale up from single servers to thousands of machines, each offering local computation and storage. Rather than rely on hardware to deliver high-availability, the library itself is designed to detect and handle failures at the application layer, so delivering a highly-available service on top of a cluster of computers, each of which may be prone to failures.

The Apache Hadoop project includes these modules:

- **Hadoop Common:** The common utilities that support the other Hadoop modules.
- **Hadoop Distributed File System (HDFS):** A distributed file system that provides high-throughput access to application data.
- **Hadoop Yarn:** A framework for job scheduling and cluster resource management.
- **Hadoop MapReduce:** A YARN-based system for parallel processing of large data sets. Hadoop MapReduce is a software framework for easily writing applications which process vast amounts of data (multi-terabyte data-sets) in-parallel on large clusters (thousands of nodes) of commodity hardware in a reliable, fault-tolerant manner.

Fig. 2.5 Components in Apache Hadoop

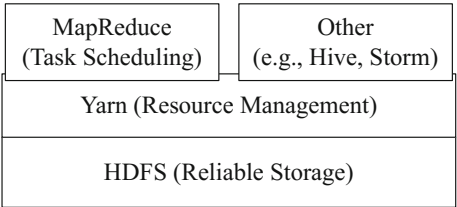


Figure 2.5 shows the components in the latest version of Apache Hadoop (version 2.x). It is worth noting that there are many other systems (e.g., Apache Hive, Apache Storm) are implemented on top of the Hadoop Yarn resource manager.

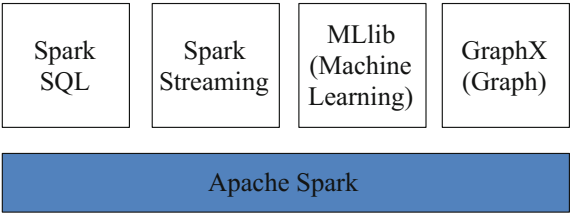
2.3.1.2 Apache Spark

Apache Spark [19] is a fast and general-purpose cluster computing system. It provides high-level APIs in Java, Scala, Python and R, and an optimized engine that supports general execution graphs. It also supports a rich set of higher-level tools including Spark SQL for SQL and structured data processing, MLlib for machine learning, GraphX for graph processing, and Spark Streaming. Figure 2.6 shows the main components in Apache Spark.

At a high level, every Spark application consists of a driver program that runs the users main function and executes various parallel operations on a cluster. The main abstraction Spark provides is a resilient distributed dataset (RDD), which is a collection of elements partitioned across the nodes of the cluster that can be operated on in parallel. RDDs are created by starting with a file in the Hadoop file system (or any other Hadoop-supported file system), or an existing Scala collection in the driver program, and transforming it. Users may also ask Spark to persist an RDD in memory, allowing it to be reused efficiently across parallel operations. Finally, RDDs automatically recover from node failures.

A second abstraction in Spark is shared variables that can be used in parallel operations. By default, when Spark runs a function in parallel as a set of tasks on different nodes, it ships a copy of each variable used in the function to each task. Sometimes, a variable needs to be shared across tasks, or between tasks and the driver program. Spark supports two types of shared variables: broadcast variables, which

Fig. 2.6 Components in Apache Spark [19]



can be used to cache a value in memory on all nodes, and accumulators, which are variables that are only added to, such as counters and sums.

2.3.1.3 Apache Storm

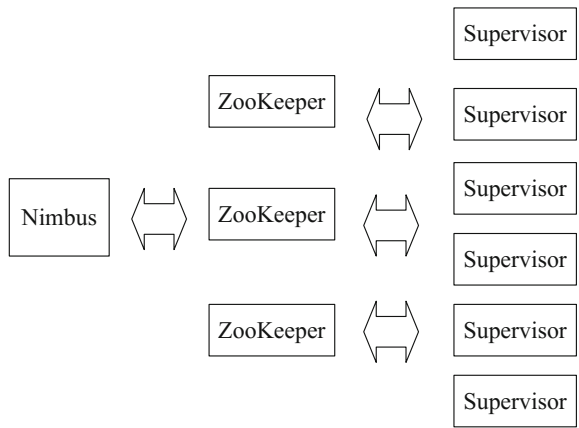
Apache Storm [20] is a free and open source distributed realtime computation system. Storm makes it easy to reliably process unbounded streams of data, doing for realtime processing what Hadoop did for batch processing. Storm is simple, can be used with any programming language.

Storm has many use cases: realtime analytics, online machine learning, continuous computation, distributed RPC, ETL, and more. Storm is fast: a benchmark clocked it at over a million tuples processed per second per node. It is scalable, fault-tolerant, guarantees your data will be processed, and is easy to set up and operate.

Storm integrates with the queueing and database technologies you already use. A Storm topology consumes streams of data and processes those streams in arbitrarily complex ways, repartitioning the streams between each stage of the computation however needed.

In Storm, nodes can be classified into three categories: nimbus node (master node), zookeeper nodes, and supervisor nodes. The nimbus node is responsible for uploading jobs for execution, distributing code across all the nodes, launching workers across all the nodes, and reallocating workers as needed. The zookeeper nodes are responsible for coordinating all the supervisor nodes. The supervisor nodes actually host the workers. They communicate with nimbus node through zookeeper nodes, start and stop workers on them according to the signals from the nimbus node. Figure 2.7 shows the organization of the nodes in a storm cluster.

Fig. 2.7 The organization of the nodes in a storm cluster [20]



2.3.2 Programming Environments for Task Parallelism

For task parallelism, due to the good performance of work-stealing policy, academia and industry world have developed task scheduling systems using work-stealing policy. For instance, Supertech Research group at MIT developed MIT Cilk [17], IBM developed X10 [7], Intel developed Cilk Plus [4] and TBB [5]. As Cilk Plus is developed based on MIT Cilk, in this section, we introduce the widely-used MIT Cilk, TBB and X10 task scheduling system.

2.3.2.1 MIT Cilk

MIT Cilk is one of the earliest parallel programming environments that implement task-stealing [17]. It extends C with three keywords: *cilk*, *spawn* and *sync* to declare parallelism in the program. *cilk* identifies a procedure as a *Cilk procedure*, *spawn* is used to generate a child task, and *sync* waits for all the child tasks that are generated by the current task to return. Only Cilk procedures can be invoked with *spawn* as a task.

Algorithm 1 shows an simple example of MIT Cilk program. It worth noting that, if we remove the keywords *cilk*, *spawn* and *sync* in Algorithm 1, the program becomes a sequential program. MIT Cilk provides user-friendly programming interface. Using MIT Cilk, programmers can simply parallelize sequential programs by creating parallel tasks using *spawn* and inserting *sync* for synchronizing the parallel tasks. The modified programs can run on parallel architecture in parallel efficiently.

Algorithm 1 An example MIT Cilk program.

```

cilk void foo (int start, int end) {
    if(end-start < threshold) {
        more instructions;
    } else {
        int mid = (start + end) / 2;
        spawn foo (start, mid);
        spawn foo (mid, end);
        sync;
        return;
    }
}

cilk void main (int start, int end) {
    spawn foo (start, end);
    sync;
    return;
}

```

MIT Cilk consists of a compiler and a scheduler. Cilk compiler, named as *cilk2c*, is a source-to-source translator that transforms a Cilk source into a C program. *cilk2c*

generates a *fast clone* and a *slow clone* for every Cilk procedure. The slow clone is executed if the task of the procedure is stolen; otherwise, the fast clone is executed instead. In addition, *cilk2c* uses a *task frame* data structure for every Cilk procedure. Once a task is generated, a task frame is created to store the information needed by the task and the scheduler. Cilk scheduler is a traditional task-stealing scheduler.

2.3.2.2 TBB

TBB (Thread Building Blocks) [5] is a set of C++ template library developed by Intel. Using the template provided by Intel TBB, programmers do not need to consider how to assign tasks to workers, and do not need to schedule the workers. There are six main modules in TBB: *Algorithm*, *Container*, *Memory Allocation*, *Synchronization*, *timing*, and *Task Scheduling*. Using the interface provided by TBB, programmers can easily define parallel tasks that can be dynamically schedule to different workers at runtime. In order to fully utilize the available resources in the parallel architecture, TBB adopts work-stealing to dynamically schedule the executable tasks. TBB provides a large amount of template functions, programmers can use these functions to create parallel tasks automatically. Algorithm 2 shows an sorting program written

Algorithm 2 An example of TBB program.

```
#include <tbb/task_scheduler_init.h>
#include <tbb/parallel_sort.h>
#include <math.h>
int main() {
    const int N = 100000;
    float a[N];
    for( int i = 0; i < N; i++ )    a[i] = sin((double)i);
    tbb::task_scheduler_init init; //Initialization
    tbb::parallel_sort(a, a + N); //Sorting
    return 0;
}
```

in TBB. Observed from the algorithm, we can find that it is easy to create parallel programs using Intel TBB. By simply replacing the sequential library call “std::sort” with “tbb::parallel_sort” provided in TBB, the sequential sorting program is updated to parallel sorting program. The task scheduling system of TBB uses work-stealing to balance the tasks generated by parallel_sort to different workers at runtime.

2.3.2.3 X10

MIT Cilk and TBB can only run on shared memory parallel architecture, and do not support distributed memory architecture. In order to solve this problem, IBM proposed the X10 [22] task scheduling system for distributed memory parallel archi-

ture. X10 is a parallel programming and scheduling system based on Java. The programming model used in X10 is “Asynchronous, Partitioned Global Address Space, APGAS”.

X10 extends the traditional programming model, and adds three keywords: *place*, *async*, and *finish*. Programmers can use *place* to define a task is created on which compute node, and use *async* to create parallel tasks, and use *finish* to create synchronization point. Functionally, the keyword *async* in X10 is similar to the keyword *spawn* in MIT Cilk; and the keyword *finish* in X10 is similar to *sync* in MIT Cilk. Meanwhile, the X10 runtime system uses work-stealing policy to schedule the tasks created with *async* to different workers running on distributed nodes.

2.4 Problems in Existing Task Scheduling Systems

However, the aforementioned task scheduling systems assume simple parallel architecture, and are not optimized against the complex parallel architectures used in real systems. Because traditional task scheduling policies lack targeted optimization, they cannot benefit from the features of the newly developed parallel architectures, which may result in poor performance.

Generally speaking, there are the following main issues that need to be resolved in order to create an effective dynamic task scheduling policy for complex parallel architectures.

- The utilization of shared cache in each socket of the MSMC architectures must be improved, to improve cache performance.
- Remote memory access on the NUMA-based shared memory system of MSMC architectures must be reduced, to reduce data access latency.
- Task distribution to the asymmetric cores in the AMC architecture must be scheduled so that the tasks are all completed at the same time.
- The workload should be balanced across heterogeneous processing elements (e.g., CPU and GPU) to achieve the best performance.
- The task scheduling policy should be optimized against big data processing, to improve data locality and relieve network congestion.
- The Quality-of-Service of user-facing application have to be guaranteed when schedule tasks to the same cluster/datacenter.

2.5 Chapter Highlights

In this chapter, we introduce widely-used task scheduling policies and the programming environments that support the corresponding scheduling policies. In the following chapters, using the random work-stealing policy proposed in MIT Cilk as the baseline, we introduce the techniques proposed to address the above problems.

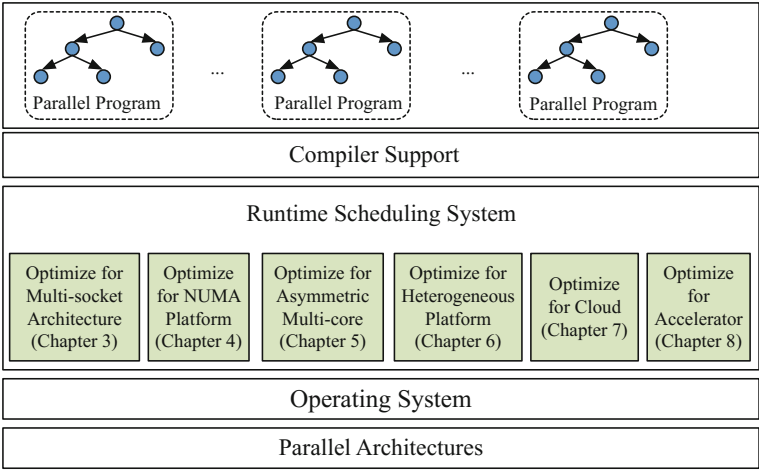


Fig. 2.8 The organization of this book. We introduce techniques proposed to improve random work-stealing for various parallel architectures

Figure 2.8 presents the organization of this book. More specifically, we first discuss task scheduling techniques for improving the performance of parallel applications on various CPU-based shared memory parallel architecture (Chaps. 3, 4 and 5). Then, we discuss task scheduling techniques for improving application performance on shared memory heterogeneous architecture consists of CPU and accelerator (Chap. 6). After that, we discuss efficient task scheduling policies for big data processing on large-scale distributed memory parallel architecture (Chap. 7). Lastly, we discuss how to schedule the co-located tasks to guarantee the QoS of high priority applications while multiple applications are executed on the same cluster/datacenter (Chap. 8).

References

1. W. Gropp. Mpich2: A new start for mpi implementations. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 2474 of *Lecture Notes in Computer Science*, pages 7–7. Springer, 2002.
2. D. Butenhof. *Programming with POSIX threads*. Addison-Wesley Longman Publishing Co., Inc, Boston, MA, USA, 1997.
3. R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, August 1996.
4. C. Leiserson. The Cilk++ concurrency platform. In *the 46th Annual Design Automation Conference*, pages 522–527. ACM, 2009.
5. J. Reinders. *Intel threading building blocks*. O’Reilly, 2007.
6. D. Lea. A Java fork/join framework. In *the ACM conference on Java Grande*, pages 36–43. ACM, 2000.

7. J. Lee and J. Palsberg. Featherweight X10: a core calculus for async-finish parallelism. In *Proceedings of the 15th ACM SIGPLAN symposium on Principles and Practice Of Parallel Processing*, pp. 25–36, Bangalore, India, 2010. ACM.
8. E. Ayguadé, N. Copt, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. The design of OpenMP tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20(3):404–418, 2009.
9. L. A. Barroso, J. Dean, and U. Holze. Web search for a planet: The google cluster architecture. *Micro*, 23(2):22–28, 2003.
10. R. Buyya, C. Yeo, S. Venugopal, J. Broberg, and I. Brandic. Cloud computing and emerging IT platforms: vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems*, 25(6):599–616, 2009.
11. J. Dean and S. Ghemawat. MapReduce: a flexible data processing tool. *Communications of the ACM*, 53(1):72–77, 2010.
12. J. Varia. Cloud architectures. *White Paper of Amazon*, <http://jineshvaria.s3.amazonaws.com/public/cloudarchitectures-varia.pdf> (2008).
13. L. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner. A break in the clouds: towards a cloud definition. *ACM SIGCOMM Computer Communication Review*, 39(1):50–55, 2008.
14. A. Gerasoulis and T. Yang. A comparison of clustering heuristics for scheduling directed acyclic graphs on multiprocessors. *Journal of Parallel and Distributed Computing*, 16(4):276–291, 1992.
15. R. D. Blumofe. *Executing Multithreaded Programs Efficiently*. Ph.D. thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, September 1995. MIT Laboratory for Computer Science Technical Report MIT/LCS/TR-677.
16. L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering*, 5(1):46–55, 1998.
17. M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 212–223. ACM, June 1998.
18. Hadoop. Hadoop home page. <http://hadoop.apache.org/> (2011).
19. Spark. Spark home page. <http://spark.apache.org> (2016).
20. Storm. Storm home page. <http://storm.apache.org> (2016).
21. Twitter. Heron home page. <https://github.com/twitter/heron> (2016).
22. X10. X10: Performance and productivity at scale. <http://x10-lang.org> (2017).

Task Scheduling for Multi-core and Parallel
Architectures

Challenges, Solutions and Perspectives

Chen, Q.; Guo, M.

2017, XVIII, 243 p. 107 illus., 73 illus. in color.,

Hardcover

ISBN: 978-981-10-6237-7