PAPER

Improving Per-Node Computing Efficiency by an Adaptive Lock-Free Scheduling Model

Zhishuo ZHENG^{†*a)}, Deyu QI[†], *Nonmembers*, Naqin ZHOU^{††}, *Member*, Xinyang WANG[†], and Mincong YU[†], *Nonmembers*

SUMMARY Job scheduling on many-core computers with tens or even hundreds of processing cores is one of the key technologies in High Performance Computing (HPC) systems. Despite many scheduling algorithms have been proposed, scheduling remains a challenge for executing highly effective jobs that are assigned in a single computing node with diverse scheduling objectives. On the other hand, the increasing scale and the need for rapid response to changing requirements are hard to meet with existing scheduling models in an HPC node. To address these issues, we propose a novel adaptive scheduling model that is applied to a single node with a many-core processor; this model solves the problems of scheduling efficiency and scalability through an adaptive optimistic control mechanism. This mechanism exposes information such that all the cores are provided with jobs and the tools necessary to take advantage of that information and thus compete for resources in an uncoordinated manner. At the same time, the mechanism is equipped with adaptive control, allowing it to adjust the number of running tools dynamically when frequent conflict happens. We justify this scheduling model and present the simulation results for synthetic and real-world HPC workloads, in which we compare our proposed model with two widely used scheduling models, i.e. multi-path monolithic and two-level scheduling. The proposed approach outperforms the other models in scheduling efficiency and scalability. Our results demonstrate that the adaptive optimistic control affords significant improvements for HPC workloads in the parallelism of the node-level scheduling model and performance.

key words: job scheduling, adaptive lock-free scheduling, optimistic concurrency control, high performance computing, many-core

1. Introduction

High performance computing (HPC) systems employ distributed paradigms designed to support the concurrent execution of performance-critical and large-scale applications while minimizing cost in terms of initial investment, operating costs, and energy consumption. The overall performance of an HPC system depends heavily upon the quality of the scheduling system. One problem with the HPC scheduling system is that jobs are allocated integral numbers of nodes, meaning that if a job uses a fraction of a node's resources (for example, one-third of the processor cores or half of the memory), the remaining resources will be wasted. Previous studies have noted this problem regard-

Fig. 1 The data results from [15], demonstrating the I/O throughput for DISC and HPC systems from the year of 2009 to 2017.

ing the low use of a node's memory and underutilization of a node's CPU resource [1]–[3]. Consequently, various cluster scheduling algorithms (such as POPPA [4], DFRS [5]) and backfilling algorithms [6]–[8] have been developed to allow multiple jobs to coexist on the same node to improve overall turn-around times and fairness, which leads to improving the utilization of HPC systems.

Trends for HPC workload point toward diversity. HPC workloads are traditionally dominated by parallel applications that are large, parallel and tightly-coupled. Recently, however, other application types have become more prevalent, including dramatically more data-intensive and high-throughput applications and real-time jobs [9], [10]. Therefore, jobs that share the same node can have distinct scheduling objectives and different resource requirements. The performance model for data-intensive applications often differs from that of traditional parallel applications (e.g., message passing interface: MPI) [11]. As explored in previous studies [12], [13], the performance of MPI applications depends upon execution uniformity and communication latency, whereas data-intensive applications' performance typically depends upon memory and I/O performance. More noteworthy is that the performance of an HPC scheduler is greatly affected by the workload to which the HPC system is applied [14]. This behavior is illustrated in Fig. 1, which demonstrates that the capacity for HPC systems to handle an I/O-intensive workload is far below that of the Data-Intensive Scalable Computing (DISC) systems. Clearly, future per-node schedulers will have to efficiently support different scheduling logics for the various applications in a scalable way, placing a heavy burden on node-level schedulers.

Manuscript received January 31, 2018.

Manuscript revised May 17, 2018.

Manuscript published July 6, 2018.

[†]The authors are with the School of Computer Science and Engineering, South China University of Technology, P.R. China.

^{††}The author is with the Guangzhou University, P.R. China.

*Presently, also with the Department of Physics and Information Engineering, Guangdong University of Education, P.R. China.

a) E-mail: z.zhishuo@mail.scut.edu.cn

DOI: 10.1587/transinf.2018EDP7038

To address the above issue, we propose a novel approach that applies a new parallel scheduling mechanism built on shared state of many-core underlying hardware resources. The approach supports flexible scheduling logics and uses an adaptive transaction memory mechanism to coordinate competition for resources between co-running jobs, which achieves both utilization and performance scalability. We then investigate the efficiency of different scheduling models in many-core systems and propose an adaptive lock-free scheduling model guided by full knowledge about the underlying hardware resources with no resource management.

The contributions of this study are as follows:

- We propose a lock-free scheduling model for a HPC node and make it adaptive to adjust the number of running tools for different jobs when conflicts occur, which happens frequently.
- We evaluate our scheduling model via the simulation of synthetic and real-world HPC workloads and compare it with multiple-path monolithic and two-level scheduling models.
- We show that our approach comprehensively outperforms these other common scheduling models and the simulation results show that the adaptive lock-free scheduling model performing well in most experimental scenarios.

We organize the remainder of the paper as follows. Section 2 provides more detail on related research. Section 3 formalizes the problem of per-node scheduling model. Section 4 describes the design of an adaptive lock-free scheduling model and how its design addresses scalability problems. Section 5 explains the experimental procedure for evaluating the scheduling models presented in Sect. 2 and Sect. 4, and Sect. 6 discusses the results using comparative diagrams. Section 7 concludes the paper with a summary of our results and provides an outlook for future works.

2. Related Work

The resource-allocation requirements of applications vary, which is not new to HPC centers [16]. Moreover, the rapid development of data-intensive workloads and the increasing use of real-time workloads have complicated resource-allocation requirements more complicated. With the advent of powerful many-core systems, applications such as MPI applications, data-intensive applications, real time jobs, and regular jobs, are likely to coexist in the same node, making it difficult for the current node-level scheduling model to flexibly support the increasing number of scheduling logics.

The most general choice for a node-level operating system is Linux, whose scheduling model is monolithic. However, Linux support multiple scheduling logics by enabling different jobs running on multiple code paths. This multi-path monolithic scheduling model is presented in Fig. 2 (a). Such a scheduling model is difficult to be scalable if more scheduling logics must be added. The addition of different scheduling logics leads to a rapid growth in code and a lack of flexibility [17], [18]. The Lithe project has re-

Fig. 2 Schematic overview of multi-path monolithic and two-level scheduling mode.

cently revisited application level scheduling as an alternative to general-purpose threads and process schedulers, and concluded that a single centralized OS scheduler is neither scalable, nor sufficiently flexible for modern multi-core application. Consequently, some studies have proposed two-level scheduling models [19], [20]. This scheduling model (Fig. 2 (b)) separates resource allocation from resource scheduling. It also advocates supporting a variety of parallel programming models in a uniform and composable way via Lithe. Two-level scheduling models assume that the tasks run on them are largely short-lived and relinquish resource frequently, a valid assumption for data-intensive workloads [21], [22]. However, because HPC workloads are dominated by compute-bound and long-running applications [1], [23], two-level scheduling models are not applicable for HPC nodes.

Other scheduling models such as fully distributed and statically partitioned types have also been proposed to address the scaling issues in many-core computers. The distributed scheduling model [24] consists of distributed schedulers, each of which has a replication of global OS state and is designed to be completely decentralized. Each individual scheduler can make a placement decision without coordination and communicate with each other through message passing. The statically partitioned scheduling models argue that the key factors limiting the performance scalability are mainly the use of hardware locks and implicitly resource sharing. Therefore, one of statically partitioned scheduling models [25] factors OS into function specific services, distributes them into spatially distributed cores and communicates via message passing, while the other one [26] is an exokernel that solves the scalability problem by reducing sharing to the bare minimum desired by an application. The statically partition solutions lead to fragmentation and suboptimal utilization, which will be further investigated in Sect. 6.3. The distributed scheduling models have difficulty in affording application-specific scheduling policies in that they are intended for rapid decisions based on minimal knowledge. Subsequently, distributed scheduling is not feasible for us, and we will not explore it any further.

Considering the pros and cons of the aforementioned solutions, we address scheduling efficiency and scalability by using adaptive lock-free scheduling, which is one example of an optimistic concurrency control mechanism. This mechanism has been studied in the database community for

some time and was recently investigated in the transaction memory community as a new parallel programming mechanism [27]. We draw on the principles of Exokernel [28] in terms of exposing sufficient information about underlying hardware resource to jobs but suggest that the coordination for the competition for underlying hardware resources should be left to user-level schedulers rather than to application writers. Therefore, the adaptive lock-free model will expose the entire many-core state to all user-level schedulers, which will implement scheduling based on shared state. This will be further discussed in the following sections.

3. Model Assumption

Various scheduling algorithms have been proposed to permit node sharing between jobs. Many-core systems are used extensively, making the underlying operation of a node much more similar to a distributed system than it was previously [29]. In other words, past studies have reported that HPC workloads, previously dominated by tightly coupled MPI jobs, are increasingly including high-throughput, data-intensive, and stream-processing applications [30]. Overall, workloads are becoming more diverse at both the application and job levels, leading to new challenges for classical HPC schedulers. This fact drives the realization of flexibly supporting multiple scheduling logics at the node level, as the number of processor cores increasing. The most common technique used in current node-level scheduling is multi-path monolithic. It lacks of parallelism and supports a large number of scheduling logics at the price of adding a large amount of extra code. Because of these problems, some studies have advocated using a two-level scheduling model, which lacks control of the underlying resource and loses the global knowledge of a node for determining priority. In this work, we opt for supporting multiple scheduling logics in a composable and uncoordinated manner, which is enabled by Exokernel. Our anticipation is to handle the problems of monolithic scheduling model without suffering from the drawback of multi-path monolithic and two-level scheduling models.

3.1 System Overview and Job Models

Because modern many-core machines that can easily apply 48 or more CPU cores to process a 100GB dataset entirely in memory are already used in many practical applications [31], we assume that each node is a many-core system with multi-lane high-speed networking cards, hundreds of gigabytes of memory and terabytes of disk space. As discussed in previous studies [32]–[35], jobs with different resource demands and distinct scheduling logics tend to co-exist on the same node. Accordingly, we assume the workloads used in our simulator are diverse at both application and job level. Each job consists of one or more tasks to be executed in parallel. Our goal is to design a reasonable mechanism to implement quick and sound resource allocation

at the node level.

In this work, we target a mixture of two different types of workloads: HPC workloads and data-intensive workloads, which will be discussed detailedly in Sect. 5.2. HPC workloads mostly comprise regular parallel applications, which are primarily compute-bound. Data-intensive workloads are dominated by I/O bound applications. As previously shown [36], most large distributed scientific applications fall under the Single Program Multiple Data (SPMD) programming model, which consists of multiple identical or nearly identical tasks that make very similar demands on the system. We assume that all the tasks in a job have the same memory and CPU requirements and that they must progress at the same rate. The memory requirement and CPU need for a task is expressed as a fraction of total node memory and a fraction of the available CPU cores for a node that allows the task to run at maximum speed. For instance, a task may require 30% of a node memory and would use 50% of the node's CPU resource in dedicated mode. We also assume that such proportions are publicly known and will not be changed throughout the entire job execution process.

3.2 Use of Exokernel

Some studies advocate revealing very little about the state of the overall system to applications, and avoiding a knowledge of the existence of other applications in the system by virtual machine (VM) technology. Parallel applications are sensitive to the underlying state of the system and require information exposure that includes details of the underlying system and the actual resource usage. VMs have been able to meet severe performance penalties [31]–[34]. Consequently, Exokernel suggests exposing information about the underlying system to the application writers to allow them to make the best decisions, i.e. permitting the application writers to specialize a library operating system (lib-os for short) [28], [34] and writes its own scheduling logic to suit the application's need. The main problem in Exokernel is that it leaves the coordination of resource competition to the application writers, which stresses on application writers and results in weakening of its advantages.

We target a many-core node and suppose that each application is implemented on the basis of lib-os and has its own virtual machine [33]. A similar system has been advanced in literatures [35], [36]. We use physical names of underlying resources so that each lib-os is capable of organizing resource lists for many-core node. To overcome the main issued of exokernel, we suggest that each lib-os will have a shared state list recording the current state of all many-core underlying resources rather than the resources it owns.

4. Adaptive Lock-Free Scheduling Model

In this section, we start from a lock-free scheduling model and later improve it with an automated control. Our goal is two-fold: first, to develop a scheduling model that ad-

Fig. 3 Schematic overview of lock-free scheduler.

dresses the scalability problem and second, to deliver good scheduling performance. The proposed scheduling models support multiple user-level schedulers in a composable way and allow them competing underlying hardware resources in an uncoordinated way. Our hope is that the resulting performance degradation caused by uncoordinated user-level schedulers will be compensated for the benefits introduced by of adaptive control.

4.1 Lock-Free Scheduling Model

Our proposed scheduling model is lock-free because it is based on the shared state approach: we grant each user-level scheduler in a lib-os full access to the entire many-core node and allow them to compete in a free-for-all manner. This approach utilizes optimistic concurrency control to mediate conflicts when user-level schedulers update the many-core state. As shown in Fig. 3, each user-level scheduler (S_i) is given a resilient master copy of the resources allocation, which is called the *resource state* and maintained by user-level schedulers. At this stage, we assume that resources represent CPU cores and physical memory pages; later they can be extended to I/O bandwidth and network. Each user-level scheduler makes its own scheduling decision based on its *resource state*. Therefore, user-level schedulers have complete autonomy to lay claim to any available resources, or even ones that have been acquired by another scheduler provided it has the appropriate permissions and priority. This immediately eliminates two issues in the two-level scheduler approach, i.e. limited parallelism due to pessimistic concurrency control and restricted visibility of resources in a user-level scheduler, at the potential cost of redoing work when the optimistic concurrency assumptions are incorrect.

As mentioned before, once a user-level scheduler makes a placement decision, the shared copy of the *resource state* is updated in an atomic commit similar to the transactional memory mechanism. In reality, the time between *resource state* synchronization and commit attempt is a transaction defined in transactional memory mechanism. This transaction may fail in that another user-level scheduler may have made a conflicting change in the meantime. The transactional memory mechanism guarantees that at most one of

such commits will succeed if a conflict takes place. Whether the transaction succeeds or not, user-level schedulers will resynchronize its local copy of the resource state afterwards and, try to run its scheduling algorithm again.

Lock-free scheduling model operates completely in parallel and do not have to wait for jobs in other user-level schedulers, and there is no inter-scheduler head of line blocking. However, this free-for-all manner may lead to high contention due to lack of underlying resources and produce frequent conflicts that result in starvation. To address starvation, this paper uses incremental transaction, accepting all but the conflicting changes so that transactions provide atomicity but not independence. In addition, individual schedulers have configuration settings to limit the total amount of resource they may claim, and to limit the number of jobs they admit.

4.2 Adaptive Lock-Free Scheduling Model

It is worth noting that the *resource state* is being frequently updated by each user-level scheduler, which will make a local placement decision and attempt to commit changes back to the shared copy with an atomic commit. When there are conflicting updates, the scheduler's transaction will be aborted and the scheduler must retry with a new *resource state*.

If conflicts and synchronizing the *resource state* among all the user-level schedulers forms a bottleneck, the performance of the lock-free scheduler will dramatically degrade. This situation is liable to take place when the numbers of running user-level schedulers are increasing rapidly. As the types and numbers of jobs will gradually increase, the number of user-level schedulers will also increase. This leads to the number of cores that can be allocated on average for a user-level scheduler become smaller, i.e. the ability of exploitable parallelism that a node can provide for each user-level scheduler becomes less. Similar issue has already been reported in transactional memory community [37]. We argue that dynamically adjusting the number of user-level schedulers that are allowed to execute concurrently in response to the fluctuating exploitable parallelism can be used to solve the above issue. Our proposed scheduler should improve resource usage when exploitable parallelism is low, and improve execution time when exploitable parallelism is high. We investigate the *Transaction Conflict Fraction* (TCF) as suitable measure of exploitable parallelism; the TCF is the percentage of aborted scheduler's transactions out of the total number of all scheduler's transactions in a sample period. The TCF increases during phases with high contention thus suggesting that the number of active user-level schedulers can be reduced, and vice versa.

Based on the above idea, the proportional control will be applied to improve our lock-free scheduler, which has a long history of application in a diverse range of fields to maintain some variable within a bounded range. According to the control theory terminology, the control objective of our proportional control is to maintain *process variable*

Fig. 4 Process of adaptive controlling active user-level schedulers (ULS).

TCF at a set point desirable variable, in spite of unmeasured disturbance from fluctuating exploitable parallelism that is restricted by available hardware resources. The TCF is defined as follows:

$$TCF = \frac{\sum A_i}{\sum T_i} \quad (1)$$

The symbol A_i denotes the number of aborted transactions for user-level scheduler S_i and T_i is the total number of transactions executed over a sample period. This data ranges between 0.0 (no conflicting transactions) and 1.0 (all conflicting transactions) and will be used to determine if the number of active user-level schedulers should be increased or decreased.

The output of our adaptive control is to modify the number of user-level schedulers that can be run concurrently in responsive to changes in TCF. The control algorithm used in our adaptive control is defined as follows:

$$f(t) = \begin{cases} numActiveULS + 1, & \Delta TCF \leq 0. \\ \frac{numActiveULS \times \Delta TCF}{order of magnitude for numActiveULS}, & \Delta TCF \geq 0. \end{cases} \quad (2)$$

where $\Delta TCF = TCF - SP$, and SP denotes a user-defined set point that is a target TCF that our lock-free scheduler tries to maintain. If TCF falls below SP due to an increase in

the number of transactions committing, our model will activate user-level schedulers that have been deactivated and will improve the parallelism of lock-free scheduling model by attempting to execute more user-schedulers concurrently. If TCF increases above SP due to an increase in the number of transactions aborted, our model will deactivate numbers of active user-level schedulers, which in turn reduces the number of concurrent transactions competing for the underlying resources.

The process of adaptive control for the number of actively running user-level schedulers (called *ULS* for short), developed by Petri net, is illustrated by Fig. 4. It shows that the adaptive control starts from sampling the number of actively running *ULS* over a period, which will be assigned to the variable *numOfRunningULS* later. A user-defined *miniULS* stands for the minimum number of *ULS* need to run in the scheduling model. This *miniULS* will be compared with *numOfRunningULS* in the next step. If *numOfRunningULS* < *miniULS*, the automata will be back to the sampling state; otherwise, it will step into the phase of adapting the number of active user-level schedulers by calling the function *increaseULS* (), when TCF is lower than SP , or *decreaseULS* () when TCF is higher than SP . Additionally, our adaptive lock-free scheduling model calculates ΔULS , the number of *ULS* that need to be activated, as

a product of currently active user-level schedulers (i.e. *numActiveULS*) and the difference between TCF and *SP*. User-level schedulers are activated/deactivated individually by array index.

Based on TCF, adaptive concurrency control simply adjusts the number of active user-level schedulers dynamically to what is best suited for current situation in a computing node. It also can be noted that our adaptive control has two parameters: user-defined set point (i.e. the target TCF) and the sample interval over which the TCF is sampled in order to make optimistic lock control decision. The set value of these two parameters will be described in Sect. 6.2.

5. Simulation Methodology

We will compare our proposed scheduling models with two prevalent node-level scheduling models: multi-path monolithic and two-level. To understand the tradeoff among multi-path monolithic, two-level, lock-free and adaptive lock-free schedulers, we build a simulator that is based on a manycore system. This simulator is driven by synthetic workloads and its parameters are drawn from empirical workload distributions [1], [22], [30]. We use this to compare the behavior of multi-path monolithic, two-level, lock-free and adaptive lock-free scheduling models under the same conditions with identical workloads. The rest of this section describes the simulators and how our experiment is set up.

5.1 Discrete-Event Simulator

We have developed a discrete event simulator that implements four scheduling models, including two prevalent scheduling models: multipath monolithic scheduling and two-level scheduling, lock-free and adaptive lock-free scheduling model. As discussed before, different types of jobs are capable of coexisting on the same node if appropriate cluster scheduling algorithms have been applied. Therefore, our simulator mainly takes a list of jobs as input that are likely to be scheduled on the same compute node. Each job is described by a submit time, a required number of tasks, one CPU need and one memory requirement specification and an execution time.

Our simulator makes some simplification for the four targeted scheduling models that is summarized in Table 1, allowing us to sweep across a broad range of operating points within a reasonable runtime. As a result, we are able to compare the behavior of all these scheduling models under the same conditions and with identical workloads. The rest of this section will continue to describe our simulators.

The scheduler decision times ($t_{decision}$) for assigning cores and memory pages in a user-level scheduler is modeled as a linear function as follows:

$$t_{decision} = o_{job} + o_{task} + tasks \text{ per job} \quad (3)$$

o_{job} represents per-job overhead due to coordination needs of cores and memory pages [38], [39], and o_{task} is the

Table 1 Design issues in multi-path monolithic, two-level, lock-free and adaptive lock-free scheduling models.

	Synthetic workload	Real-life workload
CPU architecture	manycore	manycore
Cores and memory request size	sampled	actual data
Initial cores and memory state	sampled	actual data
Tasks per job	sampled	actual data
Job arrival rate λ_{job}	sampled	actual data

incremental cost to place each task, which conforms with our assumption discussed in Sect. 3. The values for o_{job} and o_{task} are based on estimates from real-world HPC workloads: $o_{job} = 0.1m$ s and $o_{task} = 50ns$.

Since the dominated application in HPC workload is compute-bound, we interest in how the added I/O bound applications will influence the performance of HPC node. Our experiments in Sect. 6 will explore the effects of varying o_{job} (I/O bound) for I/O bound user-level scheduler, which aims to investigate how the proposed lock-free and adaptive lock-free scheduling models will be affected by the longer decision times needed for more complicated placement algorithm for I/O applications. Poisson distribution has been adopted to simulate the jobs arrival on node level. We also vary the job arrival rate λ_{job} to a 32-cores node to see how it impacts the scheduling performance.

Metrics in Simulation. Typically, users evaluate the perceived quality of manycore's scheduling by considering the time until their jobs start running, as well as their runtime to completion. We refer to the former metric as *per-job wait time*, which we define as the difference between the job submission time and the beginning of the job's first scheduling attempt. Our user-level schedulers process one request at a time, so a busy scheduler will cause delay of the jobs that want to enqueue. Consequently, *per-job wait time* measures the depth of scheduler queues, and will increase as the scheduler gets busy, either because it receives more jobs, or because these jobs take longer to schedule.

As mentioned above, *per-job wait time* depends on time fraction in which the scheduler is busy making scheduling decisions, called the *busyness of scheduler*. It increases with the *per-job decision time* (i.e. o_{job}), and it also increases if scheduling work must be redone in the condition of conflicts in lock-free approaches. To evaluate the occurrence probability of the latter, we measure the average number of conflicts per overall committed transactions, called the *fraction of conflict*. The value 0 for the fraction of conflict means no conflicts took place, while the value 3 indicates that the average workloads experiences three conflicts, and thus requires four scheduling attempts.

5.2 Workloads

Workload heterogeneity [30] is commonplace in current HPC workloads. Our simulators aim to address two main applications: data-intensive applications and traditional MPI applications. There are many approaches to partition workloads among clusters [1], [23]. To simplify and obtain the critical behavior characteristics of different scheduling

models, we select a simple two-way split on top of our simulators between I/O-bound jobs that provide end-user operations (e.g., web services) and internal infrastructure services (e.g., BigTable), and compute-bound jobs that perform a computation.

We adopt synthetic workloads for part of the study, using parameters drawn from empirical workload distributions from previous studies [30], [40]. There are a number of reasons to use synthetic workloads. Real workloads are often of poor quality, and may not contain all of the information that we require. In addition, real workloads are for specific systems, whereas synthetic workloads are generated using a model instantiated from multiple systems, and thus can be more representative. The specific detail will be discussed in Sect. 5.2.1.

Furthermore, to test the actuality of our simulator, we conduct experiments with a real-world workload from a well-established on-line repository [41]. Most logs provide standard information such as job arrival time, start time, completion time, requested duration and size in terms of the number of nodes. This will be discussed in the following Sect. 5.2.2.

5.2.1 Synthetic Workload

Through the use of Exokernel, page coloring [42]–[44] and other techniques, the CPU cores and memory pages of a many-core node can be precisely shared, partitioned and distributed to different applications precisely. Thus, the total amount of allocated CPU resources in a node is constrained not to exceed 100%. CPU resource utilization of 100% can only be reached by a single job if that job is CPU-intensive and implemented using multiple threads. A CPU-intensive sequential job can only use $\frac{100}{n}\%$ of the node's CPU resources, where n is the number of processor cores for that node. In our synthetic trace experiments, we arbitrarily assume a 64-core node, which means that a sequential task would use at most 2% of the node's CPU resources.

As discussed [1], [24], we assume that all the tasks of jobs from HPC workloads are mainly focused on CPU-intensive, and the task in one-task job are sequential. For data-intensive jobs [40], we assume that a job is made up of one or more tasks (occasionally thousands of tasks), which are mostly CPU-intensive ($> 80\%$) but with fast turnaround, whereas the rest are I/O-intensive ($< 20\%$) and consume the majority of resources. The latter typically runs for much longer and has fewer tasks than CPU-intensive jobs.

One thousand distinct traces of 2000 jobs were generated according to the parameters drawn from an empirical distribution of workloads [30], [40] and annotated with CPU need and memory requirement as described. The generated traces assume a 64-core node and thus contain jobs with between 1 and 64 tasks.

5.2.2 Real-Life Workload

To test the actuality of our simulators, we choose the Gaia

workload to test the simulators. This workload is a 3-month trace from the Gaia cluster at the University of Luxemburg operated by the oar scheduler. It includes 271 nodes and 3312 cores. A primary reason for choosing this workload is that it contains almost complete information concerning memory requirements and has I/O activity in a separate file; other workloads often contain no or incomplete information. Furthermore, the Gaia cluster has primarily used by biologists working with large data problems and engineers who are working with physical simulations, which is consistent with our assumptions for synthetic workloads.

The Gaia workload required some processing for use in our experiments. To be consistent with our previous assumptions, we focus on those jobs that only request up to the number of processors less than 64. Of the 51,987 jobs in the trace, 49385 jobs (more than 90% of the jobs) do satisfy with our requirements. The swf file format used by [41] contains information about the required number of “processors” but not the required number of tasks; thus, this value had to be inferred. Job per-processor memory requirements were set as the maximum of either the requested or used memory as a fraction of the system memory of 3GB, with a minimum observed value of 9%. Of the 49385 jobs selected from the trace, only 1,464 (3%) lacked values for either used or requested memory; we assigned these a value of 9%.

In summary, our experiments require a scheduling model that can accommodate both types of jobs, flexibly support job-specific policies, and scale to an ever-growing amount of scheduling work.

6. Evaluation

In this section, we construct simulators and compare the behavior of all the scheduling models: multi-path monolithic, two-level scheduling, lock-free and adaptive lock-free scheduling, under the same conditions and with identical workloads described in Sect. 5.

Our baseline for comparison is a multi-path monolithic scheduler with a fast code path for CPU-intensive jobs and we refer to it as a multi-path monolithic scheduler in that it still schedules only one job at a time.

The second scheduling model: two-level scheduling is modeled on the offer-based design. We simulate a first-level scheduler that can distribute CPU cores and memory pages to user-level schedulers, one dealing with CPU intensive jobs and the other is for I/O intensive jobs. To keep things simple, we assume that the user-level schedulers only have the right to manage the set of resources available to it when it begins a scheduling attempt for a job. Subsequently, any offers that arrive during the attempt are ignored. Resources unused at the end of a job scheduling are returned to the first level scheduler; they may be re-offered again if the user-level scheduler is the furthest one below its fair share. The gang scheduling used by first level scheduling is quite efficient, and owing to that, we suppose that it takes 100 ns to make a resource offer.

We use the simulator to explore our adaptive lock-free

approach. Before that, we will begin with its precedent - lock-free approach. We again simulate two user-level schedulers: one dealing with the CPU intensive job and the other handling the I/O intensive job. Each user-level scheduler is given a private, local frequently updated copy, i.e. *resource state*. Once a user-level scheduler makes a scheduling decision for a job, it will refresh their local copy of the *resource state* and synchronize it with those belong to other user-level schedulers. We define the operation of updating a *resource state* as a transaction. If there are no conflicts, then the entire transaction is accepted; otherwise only those changes that do not result in an overcommitted core or memory page are accepted. All of these materialize *resource state* in a single core for a computing node: the *resource state* in our lock-free scheduling can be read or write and the scheduling transactions are submitted directly to the different cores. The cores themselves check for conflicts and accept or reject the changes. This enhances our lock-free scheduler to make progress even if the shared-state is temporarily unavailable. The problem is that we use resource-fit in our lock-free scheduling. Obviously, if there is no feedback control, the conflict fraction is likely to rapidly increase with more and more user-level schedulers, even worse because starvation may take place. To overcome the problem in lock-free scheduling, we use Transaction Conflict Fraction (discussed in Sect. 4.2) as an index to control the number of running user-level schedulers. This will be further explored in the Sect. 5.2.

As all our scheduling models support two scheduling logics, CPU-intensive and I/O intensive, we will split the jobs into CPU-intensive and I/O intensive throughout the experiments.

6.1 Comparison by Varying Parameters

In this experiment, we keep the decision time for the CPU intensive user-level scheduler as constant, and vary the decision time for the I/O intensive user-level scheduler by adjusting o_{job} (I/O intensive). The performance of multi-path monolithic, two-level and lock-free scheduling model is shown in Fig. 5.

Multi-path monolithic model: With a fast path for CPU-intensive job in the multi-path monolithic model, both average per-job wait time and the busyness of scheduler increase significantly even at long decision times for I/O intensive jobs. Although the majority of jobs are CPU-intensive ones, CPU-intensive jobs can still get stuck in a queue behind the slow-to-schedule I/O intensive jobs or the occurrence of head-of-line blocking. Scalability is still limited by the processing capacity of a single logic scheduler (Fig. 5 (a) and Fig. 5 (b)). To avoid this, we need parallel processing.

Two-level model: The CPU intensive user-level scheduler busyness (Fig. 5 (b)) turns out to be much higher than in the monolithic multi-path case. This is a consequence of an interaction between the offer-based two-level model and the I/O user-level scheduler's long scheduling decision time. Offer-based two-level model achieves fairness by al-

Fig. 5 Performance of scheduling models.

Fig. 6 Two-level approach: unscheduled job while varying o_{job} (I/O intensive).

ternately offering all available computing resources to different user-level schedulers, predicated on assumptions that resources become available frequently and scheduler decisions are quick. As a result, a long scheduler decision time conveys that nearly all manycore resources are locked down for a long time, inaccessible to other user-level schedulers. These are often insufficient to schedule an above-average size CPU-intensive job, implying that the CPU-intensive user-level scheduler cannot make progress while the I/O intensive user-level scheduler holds an offer. It will keep trying, and as a consequence, we find that a number of tasks are abandoned because they did not finish scheduling their tasks by the 1,000-attempt retry limit in the offer-based two-level mode case (Fig. 6).

This pathology occurs because of offer-model's assumption of quick scheduling decisions, small tasks and high resource churn, which do not hold for our I/O intensive jobs. Offer-based two-level mode could be extended to make only fair-share offers, although this would complicate the resource allocator logic, and the quality of the placement decisions for big or picky jobs would possibly de-

Fig. 7 Effects of varying σ_{job} (I/O intensive: right axis) and σ_{task} (I/O intensive: left axis) on scheduler busyness (z-axis) in different scheduling schemes. Red shading of a 3D graph indicates that some of the jobs remained unscheduled.

crease, since each user-level scheduler could only perceive a smaller fraction of the available resources.

Lock-free: Fig. 5 (a) shows that the average *per-job wait time* for the lock-free approach is comparable to those for multi-path monolithic. It suggests that conflicts and interference are relatively rare, and this is confirmed by the graph of scheduler busyness (Fig. 5 (b)). Unlike offer-based two-level mode (Fig. 6), the lock-free scheduler manages to schedule all tasks in the job. Unlike the monolithic multi-path implementation, it does not suffer from head-of-line blocking: the lines for CPU-intensive and I/O intensive jobs are independent.

Furthermore, we also consider the impact of scaling σ_{task} as an additional dimension and the results are shown in Fig. 7. The results demonstrate that the monolithic scheduler is not scalable. Although adding the multi-path feature to support multiple scheduling logics, head-of-line blocking is still a problem for CPU intensive jobs. It means that multi-path monolithic model may not be able to scale to the workloads that we project for the large many-core node. The offer-based two-level model can support independent scheduler implementations, but it is hampered by pessimistic locking, and does not handle long decision time well. Therefore, it could not schedule much of the heterogeneous load we offered it.

The lock-free approach offers competitive, scalable performance, supports independent scheduler implementations, and exposes the entire allocation state to the schedulers. Our results indicate that the lock-free approach can scale to many schedulers, as well as to challenging workloads.

Fig. 8 Performance of lock-free scheduling based on varying the arrival rate λ_{job} for the CPU-intensive job.

6.2 Scalability Exploration for Adaptive Lock-Free Scheduling

In this section, we will study at how the lock-free model scales as the workload changes. For this purpose, we increase the job arrival rate of the CPU-intensive schedulers, λ_{job} . As demonstrated in Fig. 8, both *per-job wait time* and the busyness of scheduler increases. In the CPU intensive case, this is due to the higher job arrival rate, while in the I/O intensive case, and it is due to additional conflicts.

As mentioned in Sect. 4.2, there are two parameters of our adaptive schemes needed to be set: sample interval and target TCF range. Through experimentation these were set to a sample interval of 10 seconds, while lower TCF threshold of 40% and upper threshold of 70%.

Since the CPU intensive user-level scheduler is the main scalability bottleneck, we repeat the same scaling experiment with multiple CPU intensive user-level schedulers in order to test the ability of the lock-free model to scale to larger loads. The CPU intensive scheduling work is load-balanced across the schedulers using a simple hashing function. As expected, the *conflict fraction* rises with the increasing number of user-level schedulers, because greater possibility of conflict exists (Fig. 9 (a)). However, the *per-scheduler busyness* decreases with more user-level schedulers to be added (Fig. 8 (b)). This is not an encouraging result. With exploration on the waste time in scheduler busyness from Fig. 9, we infer that the decreasing of the busyness of scheduler, observed from Fig. 8 (b), may due to the high conflict fraction, which makes part of the schedulers idle, awaiting enough resource for next schedule again.

Fig. 9 Additional metrics for evaluating lock-free scheduling: varying the arrival rate for the CPU-intensive job, λ_{job} (CPU-intensive: 1.0 is the default rate).

Therefore, we try to improve the lock-free model with automated control, described in Sect. 4. The adaptive lock-free scheduling model can scale not only to a high CPU intensive job, but can also provide good behavior for I/O intensive job. We repeat the same input with our adaptive lock-free scheduling model and the results are shown in Fig. 10. To our delight, the result for conflict fraction has been greatly improved, and keeps with moderate scheduler busyness, which consists with our assumption.

6.3 Comparison to Statically Partitioned Scheduling Model

To compare our proposed adaptive lock-free scheduling model and statically partitioned scheduling model, we implement cilk [45] and its standard benchmark on a 64-core virtual machine. We make use of the cilk's default scheduling algorithm WS, and apply a statically partitioned scheduling model that statically partition 64 cores into four parts, each of which owns four cores.

Using the workloads in the cilk standard benchmark

Fig. 10 Adaptive Lock-free scheduling: varying the arrival rate for the CPU-intensive job, λ_{job} (CPU-intensive: 1.0 is the default rate).

Fig. 11 Speedup of Adaptive Lock-free with increasing number of cores.

will produce a total of 12 jobs and submit them to the system in batch mode. Taking the running time for the job and the average utilization rate of the processor cores as indicators, we test the performance scalability of our proposed adaptive lock-free scheduling model and statically partitioned scheduling model as the number of processor cores increases. The results are illustrated in Fig. 11 and Fig. 12.

As shown in Fig. 11, compared with the statically par-

Fig. 12 Utilization of adaptive lock-free scheme and statically partitioned scheme with increasing number of cores.

tioned scheme, as the number of cores increases, our adaptive lock-free scheduling model makes the execution efficiency of the job increase nearly linearly, indicating that its performance can be better expanded. The main reasons for this difference are that dynamic logical grouping of many-core resources reduces the vicious competition of tasks for resources and that there is enhanced flexibility in the allocation of resources and scheduling efficiency.

Due to the stealing nature of the WS algorithm, the processor core is always in the “busy” state, i.e. stealing immediately when it is idle. Therefore, the utilization statistics according to the working state of the worker thread in the experiment more accurately reflect the actual utilization of the processor core. The results are shown in Fig. 12. The experimental results show that compared with the adaptive scheduling model, the actual utilization of resources in the statically partitioned scheme is low, and the waste of resources is large. When the number of cores is large, the situation is even more serious and the actual resource utilization rate is less than 50%. The experimental results shown in Fig. 12 indicate that the adaptive lock-free scheduling model has a high utilization rate for resources, which is usually maintained at 90% or more.

Overall, although the statically partitioned scheme guarantees certain levels of QoS for performance and privacy, it is less scalable than the adaptive lock-free scheduling scheme as the number of cores is increases.

6.4 Analysis of Real Workload Output

Having compared the different scheduling models using the simulator with a synthetic workload, we use a real-life workload, Gaia workload that has been discussed in Sect. 5.2.2 to test our adaptive lock-free scheduling approach.

This Gaia workload has been used in a previous study [1] to test the actuality of simulators. For the same reason, it is scientific to take the Gaia workload as input to test our simulator.

We repeat a process similar to the process described in Sect. 6.2 with the real-life workload. We are encouraged that results similar to those shown in Fig. 11 are produced in Fig. 13, and this finding gives us confidence in our adaptive lock-free scheduling method.

Fig. 13 Results from using the real-life workload to evaluate adaptive lock-free scheduling: varying the arrival rate λ_{job} for the CPU-intensive job and the performance effect of varying ρ_{job} (I/O intensive).

7. Conclusions and Future Work

We have proposed an adaptive lock-free scheduling model, a novel approach applied to a single node in HPC system. This proposed scheduling model solves the scalability problem for current node-level schedules through the adaptive optimistic control. We have compared the adaptive lock-free model and its precedent with two prevalent node-level scheduling models: multi-path monolithic and two-level scheduling, using both synthetic and real-world workloads. In our simulation, all the targeted scheduling models have been simplified to sweep across a broad range of operating points within a reasonable runtime, so that we can compare the behavior of all the targeted scheduling models under the same conditions and with identical workloads.

The results show that our approach outperforms the compared scheduling models in the respect of scheduling efficiency and scalability. Our results also demonstrate that great improvement on the parallelism and the scalability of node-level scheduling model is achieved, because it allows

independent scheduler implementations that are given the knowledge of the underlying resource information to compete resource in an uncoordinated manner, and adjusts active scheduler implementations dynamically when frequent conflicts take place affords greatly improvement.

We will carry out our future work in the following directions. First, our work will focus on ways to provide global guarantees (fairness, starvation avoidance, etc.) in the adaptive lock-free model. Next, we have simply used global TCF as an indicator to adjust the lock-free scheduling model, deactivating any user-level scheduler randomly. Therefore, for the next step, our works will investigate more complex indicators to see how they influence our adaptive lock-free approach. And we will also investigate different mechanisms to guarantee QoS and privacy. Finally, to further verify the reliability of our simulator, we will develop an adaptive lock-free scheduling model-based operating system.

Acknowledgments

Thanks to many people contributed to the work described in this paper. This research has been supported by Foundation of China (Grant No. 61070015), Guangdong Provincial Science and Technology Projects (Grant No. 2014B010110004), Foundation of Guangdong (Grant No. 10351806001000000) and Guangzhou major research collaborative innovation projects (Grant No. 201604016074).

References

- [1] J. Emeras, S. Varrette, M. Guzek, and P. Bouvry, "Evalix: Classification and prediction of job resource consumption on hpc platforms," *Job Scheduling Strategies for Parallel Processing*, Lecture Notes in Computer Science, vol.10353, pp.102–122, 2017.
- [2] H. Li, D. Groep, and L. Wolters, "Workload characteristics of a multi-cluster supercomputer," *International Conference on Job Scheduling Strategies for Parallel Processing*, vol.3277, pp.176–193, 2004.
- [3] A. Batat and D.G. Feitelson, "Gang scheduling with memory considerations," *Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000*, pp.109–114, 2000.
- [4] A.D. Breslow, A. Tiwari, M. Schulz, L. Carrington, L. Tang, and J. Mars, "Enabling fair pricing on HPC systems with node sharing," *SC '13 - International Conference for High Performance Computing, Networking, Storage and Analysis*, pp.1–12, 2013.
- [5] M. Stillwell, F. Vivien, and H. Casanova, "Dynamic fractional resource scheduling for HPC workloads," *IEEE International Symposium on Parallel and Distributed Processing*, pp.1–12, 2010.
- [6] Y. Zhang, H. Franke, J. Moreira, and A. Sivasubramaniam, "An integrated approach to parallel scheduling using gang-scheduling, backfilling, and migration," *IEEE Transactions on Parallel and Distributed Systems*, vol.14, no.3, pp.236–247, 2003.
- [7] Y. Zhang, M.S. Squillante, A. Sivasubramaniam, and R.K. Sahoo, "Performance implications of failures in large-scale cluster scheduling," *Lecture Notes in Computer Science*, vol.3277, pp.233–252, 2004.
- [8] N. Fallenbeck, H.-J. Picht, M. Smith, and B. Freisleben, "Xen and the art of cluster scheduling," *International Workshop on Virtualization Technology in Distributed Computing*, p.4, 2006.
- [9] G. Fox, J. Qiu, S. Jha, S. Ekanayake, and S. Kamburugamuve, "Big data, simulations and HPC convergence," *Big Data Benchmarking, Lecture Notes in Computer Science*, vol.10044, pp.3–17, 2015.
- [10] D.A. Reed and J. Dongarra, "Exascale computing and big data," *Communications of the Acm*, vol.58, no.7, pp.56–68, 2015.
- [11] G.P. Rodriguez, "HPC scheduling in a brave new world", Ph.D. dissertation, Umeå universitet, Umeå, 2017.
- [12] L. Adhianto and B. Chapman, "Performance modeling of communication and computation in hybrid mpi and openmp applications," *Simulation Modelling Practice and Theory*, vol.15, no.4, pp.481–491, 2007.
- [13] J. Ekanayake, S. Pallickara, and G. Fox, "Mapreduce for data intensive scientific analyses," *IEEE Fourth International Conference on Escience*, 2008. *Escience*, pp.277–284, 2009.
- [14] D.G. Feitelson, "Workload modeling for performance evaluation," *Performance Evaluation of Complex Systems: Techniques and Tools*, vol.2459, pp.114–141, 2002.
- [15] J. Gray, "Sort benchmark home page," <http://sortbenchmark.org/>
- [16] D.G. Feitelson and L. Rudolph, "Toward convergence in job schedulers for parallel supercomputers," *The Workshop on Job Scheduling Strategies for Parallel Processing*, vol.1162, pp.1–26, 1996.
- [17] J.-P. Lozi, B. Lepers, J. Funston, F. Gaud, V. Quéma, and A. Fedorova, "The linux scheduler: a decade of wasted cores," *Eleventh European Conference on Computer Systems*, p.1, 2016.
- [18] S. Boyd-Wickizer, A.T. Clements, Y. Mao, A. Pesterev, M.F. Kaashoek, R. Morris, and N. Zeldovich, "An analysis of linux scalability to many cores," *Proc. Usenix Symposium on Operating Systems Design and Implementation, OSDI 2010*, October 4-6, 2010, Vancouver, Canada, pp.1–16, 2010.
- [19] J.A. Colmenares, S. Bird, H. Cook, P. Pearce, D. Zhu, J. Shalf, S. Hofmeyr, K. Asanovi, and J. Kubiawicz, "Resource management in the tessellation manycore os," *Usenix Workshop on Hot Topics in Parallelism*, 2010.
- [20] B. Rhoden, K. Klues, D. Zhu, and E. Brewer, "Improving per-node efficiency in the datacenter with new os abstractions," *ACM Symposium on Cloud Computing, SOCC*, pp.1–8, 2011.
- [21] C. Reiss, A. Tumanov, G.R. Ganger, R.H. Katz, and M.A. Kozuch, "Heterogeneity and dynamics of clouds at scale: Google trace analysis," *ACM Symposium on Cloud Computing*, pp.1–13, 2012.
- [22] S. Di, D. Kondo, and W. Cirne, "Characterization and comparison of google cloud load versus grids," *2012 IEEE International Conference on Cluster Computing*, pp.230–238, 2012.
- [23] H. You and H. Zhang, "Comprehensive workload analysis and modeling of a petascale supercomputer," *Job Scheduling Strategies for Parallel Processing*, Lecture Notes in Computer Science, vol.7698, pp.253–271, 2012.
- [24] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhanian, "The multikernel: a new os architecture for scalable multicore systems," *ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, Usa*, pp.29–44, Oct. 2009.
- [25] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, and Y. Dai, "Corey: An operating system for many cores," *Usenix Conference on Operating Systems Design and Implementation*, pp.43–57, 2008.
- [26] D. Wentzlaff and A. Agarwal, "Factored operating systems (fos): the case for a scalable operating system for multicores," *Acm Sigops Operating Systems Review*, vol.43, no.2, pp.76–85, 2009.
- [27] N. Shavit and D. Touitou, "Software transactional memory," *Distributed Computing*, vol.10, no.2, pp.99–116, 1997.
- [28] D.R. Engler, M.F. Kaashoek, and J. O'Toole, "Exokernel: an operating system architecture for application-level resource management," *Proc. ACM Symposium on Operating System Principles*, vol.29, no.5, pp.251–266, 1995.
- [29] A. Baumann, S. Peter, A. Singhanian, T. Roscoe, P. Barham, and R. Isaacs, "Your computer is already a distributed system. Why isn't your os?," *Conference on Hot Topics in Operating Systems*, pp.12–12, 2009.
- [30] G.P. Rodrigo, P.O. Östberg, E. Elmroth, K. Antypas, R. Gerber, and

- L. Ramakrishnan, "Towards understanding HPC users and systems: A nersc case study," *Journal of Parallel and Distributed Computing*, vol.111, 2017.
- [31] S. Peter, J. Li, I. Zhang, R.K.P. Dan, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe, "Arrakis: the operating system is the control plane," *ACM Transactions on Computer Systems*, vol.33, no.4, pp.1–30, 2015.
- [32] R. Sakellariou and R. Sakellariou, "Mapping Virtual Machines onto Physical Machines in Cloud Computing: A Survey," *ACM Computing Surveys*, vol.49, no.3, pp.1–30, 2016.
- [33] R.J. Creasy, "The origin of the vm/370 time-sharing system," *IBM J. Research and Development*, vol.25, no.5, pp.483–490, 1981.
- [34] H. Alam, T. Zhang, M. Erez, and Y. Etsion, "Do-it-yourself virtual memory translation," *ACM/IEEE International Symposium on Computer Architecture*, pp.457–468, 2017.
- [35] Chen and Benjie, "Multiprocessing with the exokernel operating system," *Massachusetts Institute of Technology*, vol.13, no.1, pp.5, 7, 18–24, 2000.
- [36] V. Uhlig, "The mechanics of in kernel synchronization for a scalable microkernel," *ACM Sigops Operating Systems Review*, vol.41, no.4, pp.49–58, 2007.
- [37] I. Watson, C. Kirkham, and M. Lujan, "A study of a transactional parallel routing algorithm," *International Conference on Parallel Architecture and Compilation Techniques*, pp.388–398, 2007.
- [38] J.T. Havill, W. Mao, and V. Dimitrov, "Improved parallel job scheduling with overhead," *Joint Conference on Information Sciences*, 2008.
- [39] D. Akhmetova, G. Kestor, R. Gioiosa, S. Markidis, and E. Laure, "On the application task granularity and the interplay with the scheduling overhead in many-core shared memory systems," *IEEE International Conference on CLUSTER Computing*, pp.428–437, 2015.
- [40] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: flexible, scalable schedulers for large compute clusters," *ACM European Conference on Computer Systems*, pp.351–364, 2013.
- [41] Parallel workloads archive, <http://www.cs.huji.ac.il/labs/parallel/workload/>
- [42] S. Blagodurov, S. Zhuravlev, and A. Fedorova, "Contention-aware scheduling on multicore systems," *ACM Transactions on Computer Systems*, vol.28, no.4, pp.1–45, 2010.
- [43] J. Feliu, S. Petit, J. Sahuquillo, and J. Duato, "Cache-hierarchy contention-aware scheduling in cmps," *IEEE Transactions on Parallel and Distributed Systems*, vol.25, no.3, pp.581–590, 2014.
- [44] X. Zhang, S. Dwarkadas, and K. Shen, "Towards practical page coloring-based multicore cache management," *Proc. 4th ACM european conference on Computer systems - EuroSys '09, Nuremberg, Germany*, pp.89–102, April 2009.
- [45] M. Frigo, C.E. Leiserson, and K.H. Randall, "The implementation of the cilk-5 multithreaded language," *ACM Sigplan 1998 Conference on Programming Language Design and Implementation*, pp.212–223, 1998.

Zhishuo Zheng is the Ph.D. candidate in South China University of Technology, received her B.S. in Department of Mathematics from Jinan University in 2005 and M.S. in Computer Science from School of Informatics, University of Edinburgh in 2007. She also works as tutor in Department of Physics and Information Engineering, Guangdong University of Education, Guangzhou, Guangdong from 2008 to now. Her research interests include job scheduling, operating system, parallel computer architecture, parallel computing, distributed computing system, operating system.

computing system, the new generation computer architecture and computer system security. He has published more than 200 journal papers, one monograph and two educational materials. He also obtained many patent for invention and software copyright. He has also held the 863 project, NSFC project, and so on.

and grid computing systems.

Deyu Qi received the MSc degree in NUDT and Ph.D. degree in South University of Technology. He is now a full professor and doctoral supervisor in the School of Computer Science and Engineering at SCUT, the academic team leader of advanced computing architecture and the director of Research Institute of Computer Systems at South China University of Technology. Dr. Qi's research interest include software developing method and architecture, software developing environment and tools, distributed

Naqin Zhou is a lecturer in the School of Computer Science and Engineer at Guangzhou University, Guangzhou, China. She received her the BSc degree in Computer Science from Hunan University of Science and Technology, HuNan, China in 2004, the MSc in Computer Science from HuNan Normal University, HuNan, China in 2008, and the Ph.D. in Computer Science from South China University of Technology in 2017. Her research interests include parallel computing, distributed computing

is now taking his research for Ph.D. degree in the Research Institute of Computer Systems at South China University of Technology.

Xinyang Wang is the Ph.D. candidate of South China University of Technology, received the MSc degree from the Guangxi University and BEng degree from Wuhan University of Technology. He has authored and coauthored some papers in parallel and distributed system architecture, parallel computing and network topology properties. His research interests is mainly on computer network topology, parallel computing, cloud computing, heterogeneous data integration and network fault-tolerance. He

Mincong Yu received his B.S degree in Software Engineering from South China University of Technology in 2008. During 2008, he worked for HSBC Software Development Center, during 2008-2009, he worked for CCB Software Development Center, from 2010, he has been a Ph.D. candidate in South China University of Technology, and is interested in automation system development method.