SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

January 2024, Vol. 67, Iss. 1, 112104:1–112104:17
https://doi.org/10.1007/s11432-022-3657-3

HTDcr: a job execution framework for high-throughput computing on supercomputers

Jiazhi JIANG[1], Dan HUANG[1*], Hu CHEN[2*], Yutong LU[1] & Xiangke LIAO[1]

[1]*School of Computer Science and Engineering, Sun Yat-sen University, Guangzhou* 510006*, China;*
[2]*School of Software Engineering, South China University of Technology, Guangzhou* 510006*, China*

Received 27 July 2022/Revised 13 October 2022/Accepted 6 December 2022/Published online 22 December 2023

Abstract High-throughput computing (HTC) is a computing paradigm that aims to accomplish jobs by easily breaking them into smaller, independent components. However, it requires a large amount of computing power for a long time. Most existing HTC frameworks are job-oriented without support for coscheduling with hardware architecture and task-level execution. Also, most of the frameworks reach a limited scale, and their usability needs further improvement. Herein, we present HTDcr, a job execution framework for the HTC on supercomputers. This study aims to improve the throughput, task dispatching, and usability of the framework. In detail, the throughput optimizations include a sophisticated designed task management system, a hierarchical scheduler, and the co-optimization of the task-scheduling strategy with the application and hardware characteristics. The optimizations for usability include a programable execution workflow, mechanisms for more robust and reliable service qualities, and a fine-grained resource allocation system for the colocation of multiple jobs. According to our evaluations, HTDcr can achieve outstanding scalability and high throughput on large-scale clusters for the HTC workload. We evaluate HTDcr with several microbenchmarks and real-world applications on Tianhe-2 and Sunway TaihuLight to demonstrate its effects on existing design mechanisms. For instance, the task scheduling for two real-world applications integrated with the application and hardware characteristics achieves 1.7× and 1.9× speedups over the basic task-scheduling strategy.

Keywords high-throughput computing, supercomputer, task scheduling, middleware, password guessing

1 Introduction

High-throughput computing (HTC) is a computing paradigm that focuses on efficiently executing jobs that are easily broken up into enormous loosely-coupled tasks. While high-performance computing (HPC) sounds interchangeable with HTC, they represent distinct computing paradigms. The scientific progress of research studies on HTC applications is strongly linked to computing throughput, which requires a computing environment that delivers enormous amounts of computational power over a long period. The HPC focuses on large workflows comprising highly closely-coupled tasks and uses floating-point operations per second as the yardstick to evaluate the system. HTC applications are gradually increasing in various fields, such as gene sequencing in biomedical fields [1], offline password guessing in information security field [2], and emerging machine-learning-based steering of ensemble simulations for therapeutics against diseases such as COVID-19 [3, 4].

Existing efforts on implementing HTC implementations can be divided into two categories. The first category entails implementing domain-specific systems, such as HiSeq 2000, in gene sequencing [5]. They are often built by companies or research institutes in related fields, which can meet all the needs of a certain application type. However, these systems apply to a narrow range of applications and are difficult to be extended to other domains. The second category is oriented to multiple applications in various fields and has a wide range of applicability. The representative system is HTCondor [6]. However, these frameworks for HTC applications still suffer from several shortcomings. Three limitations of existing HTC job execution frameworks can be described as follows. (1) Most of the existing frameworks reach limited scale. For instance, the ALPS (application level placement scheduler) executes a limited number

* Corresponding author (email: huangd79@mail.sysu.edu.cn, chenhu@scut.edu.cn)

of different executables simultaneously on a cray system [7]. HTCondor encounters performance bottlenecks and challenges when a burst of short-time tasks is to be processed efficiently [8]. To improve the throughput of the HTDcr framework, HTDcr should manage numerous tasks and results more efficiently and manage as many computing resources as possible. (2) The task distribution of existing HTC framework lacks optimization for common architectural features of supercomputers such as heterogeneity and shared storage. Heterogeneity is common in supercomputers. Applications with multiple phases with distinct resource necessities can take advantage of this internode heterogeneity to improve performance and reduce resource idleness. Jobs comprising multiple tasks with different resource requirements can take advantage of the hardware heterogeneity to reduce resource idling and improve performance. The correct use of all available resources by the dispatching strategy of the framework requires certain freedom to execute the phases concurrently. Shared storage is a common architecture for supercomputers. For jobs that require frequent access to data from storage, an appropriate task distribution strategy can leverage on-node storage and reduce data access from remote shared storage. (3) The usability of the current system needs to be further improved. First, most of the existing HTC frameworks, such as HTCondor, require users to parallelize the workflow through file splitting or parameter passing and decompose the job before submission. However, many HTC workloads are actually nontrivial for manual splitting. A user-friendly programming workflow is required in the HTC framework. Second, there are some MPI-based frameworks that lack robustness and fault tolerance characteristics to provide reliable services. Various mechanisms to deal with situations such as task loss and node downtime should be provided in the framework. Thus, there is much room for improvement concerning the ease of use of these frameworks.

To run HTC applications more efficiently on supercomputers and provide robust and reliable HTC services, we propose the HTDcr framework. We propose a job execution framework HTDcr targeting enhancement of HTC applications on supercomputers. Our contributions are summarized as follows.

- Increasing the capability of HTDcr to achieve higher throughput, including an efficient task management system, hierarchical scheduling, and multidispatcher mechanisms.
- Several task scheduling strategies are designed for coscheduling tasks with the architectural features of supercomputers, including strategies for shared storage and strategy for heterogeneous architecture.
- Various mechanisms are designed to improve the usability of the framework. In detail, (a) it implements a programmable workflow to decompose and execute HTC applications more flexibly. (b) It achieves fault tolerance for both scheduling and computing modules. (c) It implements a fine-grained resource allocation system for the colocation of multiple jobs.

We conduct extensive experiments for our implementation. Microbenchmarks show that the task throughput of HTDcr can outperform SLURM by 4× and HTCondor by 1.3× on average. HTDcr can reach the scale of 20000 nodes with average throughput of 1300 tasks/s for 44 h. Our task-scheduling policy achieves 1.7× and 1.9× speedups over the basic task scheduling. It is stipulated herein that applications submitted and executed through SLURM, HTCondor and our implemented system (HTDcr) are called **jobs**, and multiple execution steps are contained in a job. Task-parallel executions of an applications involve dividing a workload into a set of self-contained units of task. These tasks can be independent, have no intertask communication, or are loosely coupled with low degrees of data dependencies. For instance, the brute-force password-guessing application tries to determine the password (H is the Hash function, such as MD5). The whole process of trying all the possible password candidates is a job. The computation to verify if $H(\text{pwd}_i) = \text{cipher}$ for one specific password candidate pwd_i is seen as a task.

2 Related work

HTCondor is an open-source HTC framework for coarse-grained distributed parallelization of computationally intensive tasks [9]. It requires users to parallelize workflows through file splitting or parameter passing and decomposes the job statically before submission. Dynamic decomposition and scheduling of tasks according to the cluster load during job execution are unsupported by HTCondor. SLURM [10,11] is equipped with excellent scalability and is widely adopted as the resource management and job scheduling system on supercomputers. It uses an exclusive resource-scheduling strategy to allocate resources in the node unit. The SLURM system cannot manage and monitor the job at the task level and detect the task execution status. In the event of a system crash or node downtime, the running job is

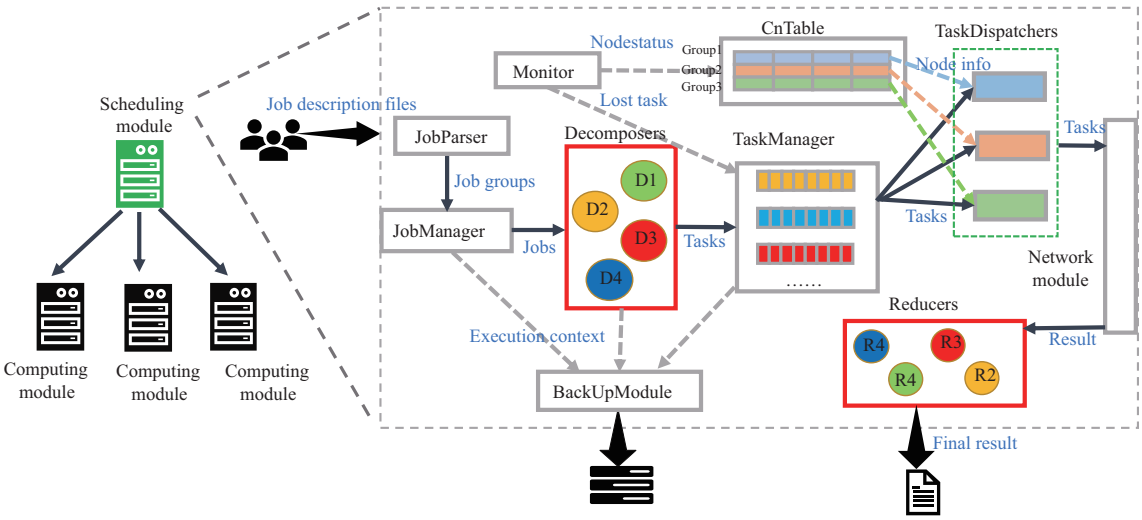

Figure 1 (Color online) Architecture of the scheduling module.

also terminated. The user can only rerun the entire HTC job, which wastes computing resources and time. Raicu et al. [12] designed and implemented a lightweight task-execution framework called Falkon. Falkon's design focuses on multilevel scheduling strategies and streamlined task dispatchers. A multilevel scheduling strategy ensures full utilization of resources and efficient execution of tasks. The streamlined task dispatcher reduces communication overhead and the idle time of computing resources. Raicu et al. [13] optimized Falkon and used C language to replace some functions. Although Falkon has tried to improve the throughput of the framework, it does not schedule the task according to the application and hardware features. Also, Falkon does not effectively address fairness and resource allocation to support multiple users submitting various jobs to a collection of distributed computing resources. Merzky et al. [14] designed and introduced RADICAL-Pilot as a portable, modular and extensible pilot-enabled runtime system for high-throughput scientific applications. The system is implemented in Python, and can be used stand-alone, as well as the runtime for third-party workflow systems. Currently, it is difficult if not impossible to compare their system performance characteristics to ours due to the lack of analogous task implementations and common metrics [14].

3 HTDcr framework design

In general, HTDcr adopts a master-worker architecture similar to HTCondor. The modules deployed on master and worker nodes are called the scheduling and computing modules, respectively.

3.1 Scheduling module

The scheduling module is the cornerstone of the entire system. Its architecture is shown in Figure 1. The data flow on the scheduling module is described as follows. First, the **JobParser** converts the job description files submitted by users into job groups. Next, the **JobManager** uses the breadth-first algorithm of directed acyclic graph to traverse all jobs in a job group and submits jobs to decomposers according to their priorities and dependencies. Then, the **Decomposers** decompose jobs from **JobManager** into tasks and add them to **TaskManager**. Afterward, the **TaskManager** carefully designs various sophisticated queues to maintain and manage numerous tasks and results generated during the execution of jobs. Furthermore, the **TaskDispachers** load the tasks from the task manager and dispatch tasks to worker nodes managed by **CnTable**. Additionally, **CnTable** manages all worker nodes by dividing them into groups. **Reducers** accept results returned from worker nodes, and perform reduction and post-processing according to the user-implemented function.

Task-level execution is mainly supported by the decomposer and task manager. For instance, in a brute-force password guessing application, the user submits a job description file to find a password candidate for a ciphertext generated by MD5. The job search space is all the string composed of 10 lower case letters, upper case letters and digits. In a job-oriented framework, this job, which contains

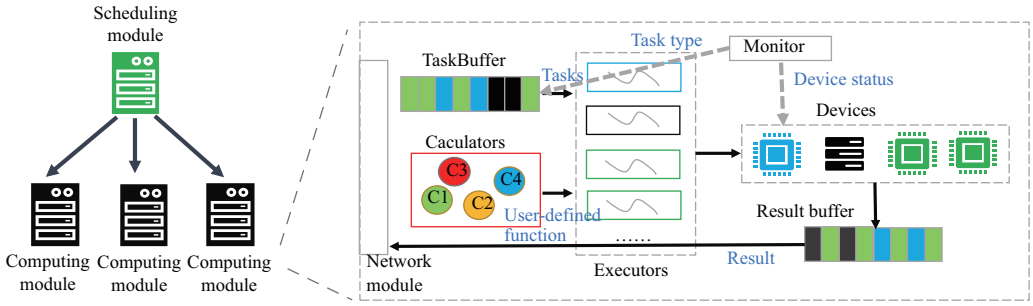

Figure 2 (Color online) Architecture of the computing module.

52^{10} computations of MD5, is assigned to a node or device for computation. In the HTDcr framework, the decomposer divides the whole search space into numerous subspaces. For instance, each subspace contains MD5 computations for 100 different password candidates. Each subspace is called a task. These numerous tasks generated by the decomposer are managed through the task manager. The scheduling module can decide how to dispatch tasks in task manager per scheduling cycle based on the number and status of the computing nodes and devices. In this way, DCR (decompose, compute, reduce) achieves task-parallel execution of the job.

Monitor and **BackUpModule** are designed to improve the system's robustness. The **Monitor** module monitors the executions of all tasks and statuses of worker nodes. It notifies HTDcr to resend tasks when they are lost or node failures occur. By employing task-level monitoring, lost tasks can be automatically resubmitted and calculated. The **BackUpModule** saves the execution context on the scheduling module regularly. All information can be restored after scheduling module is restarted.

3.2 Computing module

The computing module is designed for efficiently executing tasks assigned by the scheduling module. Its architecture is shown in Figure 2. The data flow on the computing module is described as follows. The network module receives tasks sent by the scheduling module and puts them into a task buffer. When tasks are to be executed, **Executors** fetch them from the task buffer and find their corresponding calculator. The calculator is a user-implemented module describing the computational logic. Users can customize the calculation process through the API provided by HTDcr. The device **Monitor** collects the usage information of each computing device and sends heartbeat messages to the scheduling module regularly. With this information, the scheduling module can better schedule tasks.

The decomposer, reducer and calculator are the essential components for concurrently executing programmable execution workflow and are introduced in Subsection 6.1. The task dispatcher is designed for integrating cluster task scheduling with the application and hardware introduced in Section 5. The task manager and the CnTable are implemented for the throughput enhancement and are detailed in Section 4. More details about Monitor and BackUpModule in the scheduling module are explained in the recalculation and checkpoint mechanism for fault tolerance in Subsection 6.2.

4 Throughput improvement

To improve the throughput of the HTDcr framework, there are two major challenges. (a) The inability of the scheduling module to manage numerous tasks and results more efficiently. (b) The inability of the scheduling module to manage many computing resources. To solve these problems, the following was actualized by HTDcr.

4.1 Fine-tuned task management system

The HTC framework creates and terminates numerous tasks and results while executing large-scale and loosely-coupled applications. This operation may cause significant performance degradation and limit the throughput increment of the system. The HTDcr task management system has the following advantages. (a) Task/results preallocation. Since the memory on a single supercomputer node is sufficient (32 GB on Tianhe), the maximum memory space required by the workload is preallocated instead of allocating space

during execution. This preallocation eliminates frequent use of the system memory allocator, which is not so efficient according to our measurements in Subsection 7.3.1. (b) Global task pool. HTDcr adopts a global task pool for objects that are frequently allocated and deallocated. This component puts the recently released objects in the list instead of actually releasing them. If the tasks are reused soon, they are just moved out of the global task buffer. Different types of task queues in the framework are organized as static lists [15] in the task management system. This organization can avoid the time-consuming object initialization and finalization overhead. (c) Contention is usually the highest cost in the massively parallel environment. All operations (such as create, terminate, move, modify, and read) on tasks and results are designed to avoid lock and unlock contention as much as possible.

The task management system of HTDcr is highlighted by two optimization techniques, such as the **private task** and **two-step movement** mechanisms for a massively parallel environment. The frequent lock and unlock operations lead to significant performance penalties or even deadlock. To address this issue, the private task and two-step mechanisms leverage a local temporary buffer to act as an intermediary between different threads. The tasks stored in the temp buffer are called private tasks and are only operated by a thread simultaneously. With the private task mechanism, HTDcr can avoid the following overhead when managing tasks in a massively parallel environment. (a) Since a private task is exclusive to a thread, no thread competition occurs. Therefore, no lock is required when the HTDcr framework operates the private task. (b) Data exchange between the temp buffer and a task queue only needs to lock the task queue without considering the priority order of the lock (only one lock) and deadlock occurrence. (c) Moving multiple elements between a temp buffer and a task queue has a performance gain. The operation may involve the modification of multiple elements. HTDcr modifies the temp buffer instead of the task queue. This way, it decreases contention for critical sections on task queues and alleviates resource competition.

When the tasks are moved and modified between the two task queues (i.e., the source and destination queues) in the global task pool, it is usually divided into three steps. (a) The tasks of the source queue are moved to a temp buffer. (b) The attribute value of the private task is modified in the temp buffer. The private tasks in the temp buffer are moved to the destination queue. The above task movement scheme is called a **two-step movement**. Moving the data in three steps adds some extra operation compared to moving the data directly. However, all the task queues in the global pool are organized as a static list [15], and only the cursor and several attribute values of the task queue are modified to complete the movement. There is zero memory copy in the above operations, so the overhead is almost negligible. Compared to direct data movement, two-step movement has the following advantages. (a) Two-step movement makes it possible to lock step-by-step. In the first and third steps, only the source and destination queues need to be locked, respectively. In the second step, no lock is required. The two-step movement does not cause the deadlock problem. (b) The modification of the task values only in the second step shortens the race condition time of the source and destination queues.

4.2 Hierarchical scheduling and multidispatcher

HTDcr adopts the master-worker architecture. However, this classic architecture has a drawback that the master node may become the bottleneck of the entire system. Especially when the scale of computing resources is huge, the master node manages them inefficiently. This may cause many worker nodes to stay idle for a long time because they cannot obtain tasks from the master node. The main ideas of the HTDcr resource management system are as follows. (a) **Hierarchical scheduling**. After a job is submitted through the HTDcr framework, the scheduling module determines the node the task is distributed to, and the computing modules are used to implement fine-grained resource management in the units of CPU cores/GPU/I/O resources on worker nodes. Hierarchical management avoids the direct management of numerous fine-grained resources and reduces the burden of scheduling modules. (b) **Multi-dispatcher**. Although HTDcr adopts a hierarchical management scheme for computing resources, the scheduling module may become a bottleneck when the number of computing nodes increases to a certain scale. This hinders the further improvement of system throughput. To address this issue, the scheduling module divides the worker nodes into G groups (Figure 1). The value of G can be configured and adjusted according to the total number of nodes. HTDcr launches G threads to generate G dispatcher instances on the scheduling module. Each dispatcher instance manages one node group. This way, each task dispatcher only has to manage $1/G$ of the total nodes, which enables the scheduling module to support more worker nodes participating in task executions. The rough processes of task dispatching on

the scheduling and computing modules are shown shown in Algorithms 1 and 2.

Algorithm 1 Task dispatching on scheduling module

```
Require: G: number of node group;
    jobID: the ID of a job;
    sendNum: the number of tasks to be sent;
 1: sendPerGroup = sendNum / G;
 2: cnTable[] = GetNodes();
 3: strategy = GetDispatchStrategy();
 4: for all pThread in dispatchThread do
 5:     pid = pThread.id;
 6:     nodeGroup = CnTable[pid];
 7:     dispatcher = Dispatchers[pid];
 8:     tasks[] = LoadTasks(sendPerGroup, JobID);
 9:     for all node in nodeGroup do
10:         taskType = GetTaskType(task[i]);
11:         status = CheckNodeStatus(node, taskType);
12:         if status == IDLE then
13:             SendTask(node, tasks, strategy);
14:         end if
15:         if tasks == ∅ then
16:             Break;
17:         end if
18:     end for
19: end for
```

Algorithm 2 Task dispatching on computing module

```
Require: Task: task received from scheduling module;
 1: taskType = GetTaskType(task);
 2: PutTaskInBuffer(taskType, task);
 3: for all pThread in CPUThread do
 4:     if CPUPool is not empty then
 5:         computeTask = LoadTaskFromCPUBuffer();
 6:         SubmmitTask(computeTask);
 7:         SendStatusToScheduling();
 8:     end if
 9: end for
10: for all pThread in GPUThread do
11:     if GPUPool is not empty then
12:         computeTask = LoadTaskFromGPUBuffer();
13:         SubmmitTask(computeTask);
14:     else
15:         SendStatusToScheduling();
16:     end if
17: end for
18: for all pThread in IOThread do
19:     if IOPool is not empty then
20:         computeTask = LoadTaskFromIOBuffer();
21:         SubmmitTask(computeTask);
22:         SendStatusToScheduling();
23:     end if
24: end for
```

5 Task-scheduling improvement

Different workloads can have different behavioral characteristics. Therefore, one-size-fits-all cluster task scheduling may degrade the performance of different applications deployed on the HTC framework. To enable customization and adaptation, HTDcr integrates cluster task scheduling with the application and hardware. HTC applications can achieve higher throughput with more suitable task-scheduling policies. The default scheduling policy on HTDcr is the shortest queue first (SQF) policy. The heartbeat message sent from the computing module to the scheduling module contains the task queue length on the computing module. The SQF policy prioritizes sending tasks to workers with shorter task queues because of the lighter load on these nodes. Through the SQF policy, HTDcr can achieve load balancing.

5.1 Strategy for the shared storage cluster

On a shared storage cluster (such as Tianhe-2), all nodes share storage but not memory. Some applications need to load data from the shared storage into memory, and different tasks may require common data. Reducing data loading from storage and reusing data in memory significantly improve the efficiency of the system. The strategy for shared storage clusters sends tasks that require common data to the same node for computing as much as possible. This way, the data loaded into the memory by the previous task can still be used when the next task arrives. This can avoid unnecessary data loading. The shared storage strategy is suitable for applications that exhibit spatial locality.

As shown in Figure 3, the data accessed by tasks are divided into three parts: datasets 1, 2, and 3. Tasks 1, 3, and 5 access dataset 1. Task 4 accesses dataset 3, and tasks 2 and 6 access dataset 2. First, task 1 is scheduled to node 1, and node 1 loads dataset 1 into the memory simultaneously. Since the data accessed by task 2 differs from the dataset accessed by task 1, task 2 is scheduled to node 2, and it loads dataset 2. Task 3 accesses dataset 1. Dataset 1 has been loaded on node 1, and the data locality can improve the performance of task 3. Thus, task 3 is scheduled for node 1. Task 4 is scheduled to node 3 since dataset 3 is not accessed by any previous task. Tasks 5 and 6 are scheduled to nodes 1 and 2, respectively, since datasets 1 and 2 have been loaded.

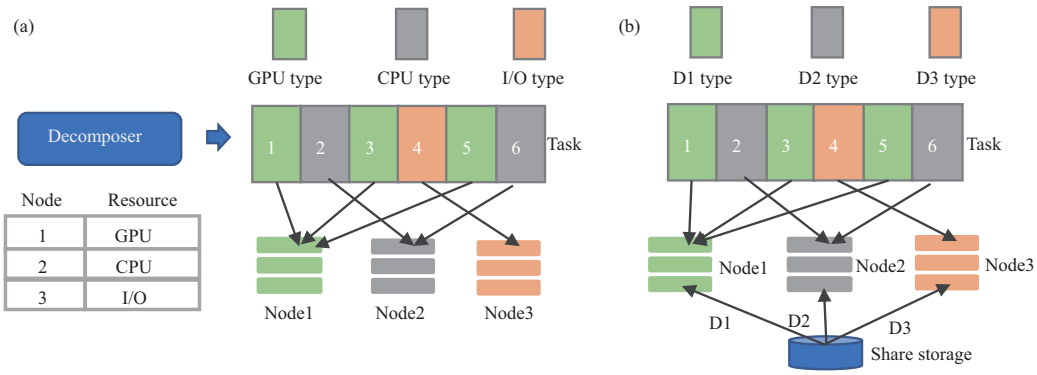

Figure 3 (Color online) Task-scheduling strategy integrating with application and hardware characteristics. (a) Strategy for the heterogeneous architecture; (b) strategy for the share storage architecture.

5.2 Strategy for heterogeneous architecture cluster

This strategy is suitable for jobs comprising different tasks. These tasks can be executed simultaneously on different devices. To execute these different tasks on the best-fit device, each task is assigned a task type according to its required resources by the decomposer. Each task is assigned by the user when the decomposer is implemented. There are three task types available currently: CPU-intensive, GPU-intensive, and I/O-intensive types.

As shown in Figure 3, the job is broken down into three types of tasks by the decomposer. Tasks 1, 3, and 5 are GPU-intensive tasks. Task 4 is an I/O-intensive task, and tasks 2 and 6 are CPU-intensive tasks. The GPU/CPU/I/O load statuses of the worker nodes are regularly fed back to the scheduling module and recorded in a table. Since the GPU load on node 1 is not heavy according to the node information presented in the table, GPU-intensive tasks 1, 3, and 5 are scheduled for node 1. Tasks 2 and 6 are CPU-intensive tasks. The CPU load on node 2 is not heavy, so tasks 2 and 6 are scheduled for node 2. Task 4 is scheduled for node 3 for a similar reason.

The worker nodes on supercomputers are commonly equipped with heterogeneous computing devices, such as CPUs and GPUs. When the computing module is started, it spawns separate threads for each device on the worker node (Figure 2). Suppose there is a worker node equipped with an 8-core CPU and 4 GPUs, the computing module on the worker node generates 13 threads: eight threads are used to execute CPU-intensive tasks, four threads are used to execute GPU-intensive tasks, and a thread is used to execute I/O-intensive tasks. These 13 threads fetch tasks matching their device types from the task buffer continuously. If any resource is available, the threads submit the fetched task for execution immediately. This way, the computing module has 13 tasks executed on one worker node in parallel. With a heterogeneous architecture policy, HTDcr can fully use various resources on the worker node.

6 Improvement of usability

HTDcr needs to do more than execute HTC applications efficiently on supercomputers. The usability of the system is also critical. It needs to provide robust and reliable HTC services, user-friendly programming workflow, and multiuser/multijob support. The following efforts are taken to improve usability.

6.1 Programmable execution workflow

Like HTCondor, deploying HTC applications on HTDcr comprises four phases: (1) discretize the job into smaller tasks; (2) preprocess the inputs; (3) execute tasks on the HTC framework; (4) postprocess the results. Many HTC workloads are actually nontrivial for manual splitting. HTCondor requires users to parallelize the workflow through file splitting or parameter passing and decomposes the job before submission. However, a key difference from HTCondor is that all these phases in the workflow of HTDcr are programmable via the programming model and concise API provided by HTDcr. This design offers two advantages. First, HTDcr has more flexibility in determining how jobs should be decomposed. Second, it allows HTDcr to dynamically decompose and schedule tasks according to the load of the cluster during job execution. To develop applications based on the programming model, users must

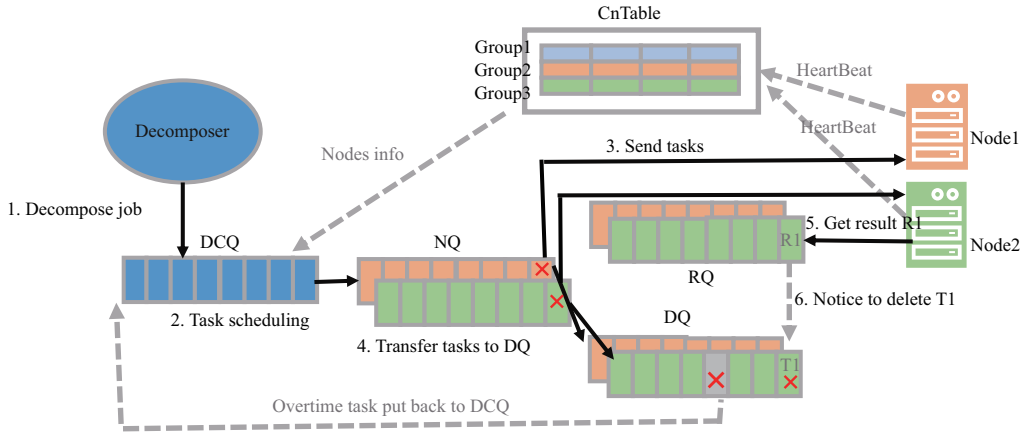

Figure 4 (Color online) Overall architecture of the recalculation mechanism.

define the following three contents. (1) Describe the data members in the task and result. (2) Implement decomposition, calculation and reduction APIs in the decomposer, calculator, and reducer. (3) Create job templates and integrate all involved classes into HTDcr. First, the information required for calculation as the data member of the computation task class is defined. Afterward, the information of the calculation results required for reducing is defined as the data member of the result, similar to how the mapper and reducer [16] are implemented in the Hadoop [17], inherit and implement the decomposer, calculator, and reducer classes predefined by HTDcr.

6.2 Mechanisms for more robust and reliable service

6.2.1 *Recalculation mechanism*

Recalculation mechanism for computing module failure and task loss. The recalculation mechanism is used for computing the module failure and task loss. In a long-term executing process on the large-scale cluster, worker node failures are common. Even if all nodes run without failure, peripheral device failures (e.g., network) may occur, resulting in packet loss of tasks. To solve this problem, HTDcr manages tasks in the scheduling module (Figure 4). The DCQ (decompose queue), NQ (node queue), and DQ (deleted queue) are task queues at different stages in the task management system. The RQ (received queue) is the result queue used for receiving the calculation results from the worker nodes. The scheduling module maintains a task buffer NQ, DQ, and RQ for each worker node.

The task dispatch process is as follows. (a) Decomposers decompose jobs from the job manager into tasks and add them to the DCQ. The decomposer works periodically; the number of tasks decomposed in each cycle depends on the system load. The task management system puts the DCQ tasks into the NQ and sends them to the worker nodes according to the scheduling strategy and the load of each node. (b) When the scheduling module sends a task to the worker node from the NQ, it does not directly discard the task but transfers it to the DQ and saves the current timestamp. (c) After the scheduling module receives results from worker nodes, it deletes the corresponding tasks in the DQ. Therefore, each task will only stay in the DQ before receiving the result. If the residence time of a task in the DQ exceeds the threshold, the monitor determines that the task is lost and moves it from the DQ back to the DCQ. In the DCQ, the task is resent to the worker node. The threshold for the task timeout is estimated as follows: T is the execution time of a task, which can be obtained after the first few results are returned. L is the maximum length of the task queue on the worker node. N is the number of devices on the worker node. Additionally, there is an ongoing task on each device. So, a task can be completed at most $T_{\max} = \frac{TL}{N} + 1$ time on the worker node. Considering some other overheads (such as the performance degradation caused by the overheating of devices and communication delays), the threshold for task timeout is set as $T_{\text{loss}} = \alpha T_{\max}$. (d) If a worker node loses connection to the scheduling module, the monitor confirms that the worker node is unavailable and deletes it from the CnTable. At the same time, all tasks of the lost node in the DQ (regardless of whether they are timeout or not) and NQ of the worker node are considered invalid, and all tasks are moved back to DCQ. This way, incorrect workload results on HTDcr due to the loss of tasks and worker node failure are eliminated.

6.2.2 *Checkpoint mechanism*

Check-point mechanism for the scheduling module failure. The checkpoint mechanism is used for investigating the failure of the scheduling module. The checkpoint mechanism is a relatively mature fault-tolerant mechanism [18]. Different systems have their specific implementations. During the execution of workloads on large-scale clusters, the failure of the scheduling module can result in the collapse of the entire computing system. HTDcr sets up a backup thread on the scheduling module to save the current execution context regularly. The saved context mainly includes as following. (a) Job information submitted by users, including job identification, specific content, and type. (b) Status of all decomposers: large jobs cannot be completely decomposed at once. The submitted job may be in the waiting (not yet decomposed), running (partially decomposed), or decomposed states. Therefore, the context of each job in the decomposer must be retained. (c) Tasks in the buffer of the scheduling module are decomposed, including tasks in the DCQ, NQ, and DQ of each node. The backup information is loaded and checked after restarting the scheduling module. The execution context before failure can be restored, which may lose at most one backup cycle of computing results for the job in progress. The computing module on worker nodes reconnects to the scheduling module and resends its current status.

6.3 Fine-grained resource allocation system

When multiple jobs are deployed on a cluster, the framework can allocate computing resources among different workloads. HTDcr does not use the traditional method of isolating the hardware to each job for resource allocation (entire-occupied allocation). Instead, it controls the proportion of computing resources each job uses by adjusting the dispatching speed of tasks. This way, different jobs can be allocated and scheduled on the shared nodes at a task level. The design principle of resource allocation algorithms is that the more computing resources a job requires, the more tasks of the job are sent in a cycle.

If the number of tasks (total compute resources) that can be sent in a scheduling cycle exceeds the sum of the lower bound of all jobs, the lower bound of each job is first guaranteed. Then, the remaining computer resources are divided equally (resources enough allocation). Otherwise, the computational resources are divided according to the ratio of the lower bound of each job (resources limited allocation). In Algorithm 3, the number of free slots of task buffer on all worker nodes is denoted as C. The job list is denoted as A, and the i-th job is represented as $A[i]$. The maximum and minimum numbers of tasks that a job can send in each cycle are denoted as lb and ub, respectively. The sum of lb of all jobs is denoted as S. If C exceeds S, the computing resources in the system are sufficient to meet the requirements of all jobs. In this case, HTDcr adopts Algorithm 3 for resource allocation; otherwise, it adopts Algorithm 4. In Algorithm 3, HTDcr first meets the minimum resource requirements lb of each job when allocating resources. The remaining resource $C - S$ is equally distributed among all jobs. The resources obtained for each task cannot exceed ub at most. In Algorithm 4, the summation of lb from $A[0]$ to $A[i]$ is denoted as $S_1[i]$, the ratio of $S_1[i]$ to S is denoted as $P[i]$, and the number of resources allocated to $A[i]$ is denoted as $N[i]$. The scheduling module first generates a random number between $[0, 1]$. The computing resource is allocated to $A[i]$ corresponding to the $P[i]$ interval where the random number is located. That is, the Monte Carlo method [19] is used to determine the job to which each computing resource is allocated. Consequently, all computing resources are allocated according to the proportion of lb of each job in S.

Algorithm 3 Resources enough allocation

Require: C: total amount of computing resources;
S: sum of lb for all jobs;
AL: length of the job list;
$A[t_0, \ldots, t_{\mathrm{AL}-1}]$: job list;
Ensure: $M = \{\langle t_i, n_i\rangle | 0 \leqslant i \leqslant \mathrm{AL} - 1\}$: Jobs and resources allocate to Jobs;
1: $M = \emptyset$, $i = 0$;
2: benefit $= (C - S)/\mathrm{AL}$;
3: **while** $i <$ AL **do**
4: $t = A[i]$;
5: $n = \min(\mathrm{lb}(t) + \mathrm{benefit}, \mathrm{ub}(t))$;
6: $M = M \cup \{\langle t, n\rangle\}$;
7: $C = C - n$;
8: $i = i + 1$;
9: **end while**

Algorithm 4 Resources limited allocation

Require: C: total amount of computing resources;
S: sum of lb for all jobs;
AL: length of the job list;
$A[t_0, \ldots, t_{\mathrm{AL}-1}]$: job list;
Ensure: $M = \{\langle t_i, n_i\rangle | 0 \leqslant i \leqslant \mathrm{AL}-1\}$: Jobs and resources allocate to Jobs;
$M = \emptyset$;
for $i = 0$ to AL **do**
  $N[i] = \{0\}$;
  $S_1[i] = \sum_0^i \mathrm{lb}(A[i])$, $P[i] = \frac{S_1[i]}{S}$;
end for
while $C > 0$ **do**
  Generate a random number r in [0, 1];
  $i = 0$;
  while $P[i] < r$ **do**
    $i = i + 1$;
  end while
  $t = A[i]$;
  if $N[i] == \mathrm{ub}(t)$ **then**
    Continue;
  end if
  $N[i] = N[i] + 1$;
  $C = C - 1$;
end while
for $i = 0$ to AL **do**
  $M = M \cup \{\langle A[i], N[i]\rangle\}$;
end for

Table 1 A brief introduction to the workloads in the experiments

Workload	Applications	Usage	Platform
T1	Microbenchmark (diffrent time tasks)	Throughput evaluation	Tianhe-2
T2	Microbenchmark (diffrent time tasks)	Throughput evaluation	Tianhe-2
E1	Microbenchmark (1 s tasks)	Efficiency evaluation	Tianhe-2
E2	Microbenchmark (8 s tasks)	Efficiency evaluation	Tianhe-2
E3	Microbenchmark (64 s tasks)	Efficiency evaluation	Tianhe-2
S1	Brute-force password guessing	Maximum nodes evaluation	Sunway TaihuLight
S2	Microbenchmark (1 s tasks)	Maximum throughput evaluation	Tianhe-2
S3	Microbenchmark (1 s tasks)	Task management system evaluation	Tianhe-2
F1	Brute-force password guessing	Fault tolerance system evaluation	Tianhe-2
R1	Brute-force password guessing	Multi-job resource allocation evaluation	GPU cluster with 4 nodes

7 Experiment

7.1 Experimental setup

7.1.1 *Configuration*

Most of the experiments are conducted on the Tianhe-2 [20]. Each node has two Intel Xeon E2-2692v2 CPUs (i.e., 12 cores each and a total of 24 cores) running at 2.2 GHz. Each node has 64 GB of memory. The nodes are connected with Tianhe Express-2. The scalability test is conducted on the Sunway TaihuLight supercomputer [21] equipped with SW26010 many-core processors. The master node has one Xeon E5-2680 v3 CPU (12 cores) running at 2.50 GHz and 32 GB of memory. The worker nodes have an SW26010 CPU (65 cores) running at 1.25 GHz. Each node has 8 GB of memory. The experimental data for comparison of HTCondor and SLURM is adopted from our previous work [8]. Besides, microbenchmarks are performed with the same configuration for HTDcr.

7.1.2 *Workload*

A brief introduction to the workloads used in the experiments is presented in Table 1. Each section has a more detailed description of the workload configurations. **Microbenchmark** refers to programs that run for a specified time and have no practical use, such as the sleep system call, and are used for performance testing. Multiple microbenchmarks are designed for evaluation.

Brute-force password guessing can be described as follows: in offline password guessing, researchers typically possess a known Hash function $H()$ (such as MD5) and a ciphertext cipher. To determine if the password satisfies $H(\mathrm{pwd}) = \mathrm{cipher}$, a set of password candidates, $\mathrm{pwd}_1, \ldots, \mathrm{pwd}_s$, to be tried in the Hash function should be generated according to some specific methods first. Then, researchers Hash the generated password candidates in turn and compare the Hash results with the knowing cipher. If the Hash result of pwd_i matches with cipher, the origin password is considered pwd_i.

BLAST [22] is a gene sequence query program that can search in the designated gene database according to the submitted gene sequence and output the sequence number and similarity of the most matching gene sequence. BLAST can query different database shards in parallel by splitting the gene database. The entire query job for the gene database is broken down into several small tasks for querying

the data blocks. The BLAST experiment chooses the wgs gene and the other_genomic gene databases with sizes of 195.7 and 239 GB, respectively. The wgs and other_genomic databases are divided into 1957 and 2390 data blocks with sizes of 100 MB, respectively, with 4347 tasks in total.

Rainbow table search [23] adopts a precalculated table for password guessing. It includes four steps and can be roughly divided into two stages. The first stage includes step 1 and step 2, the GPU-intensive chain calculation and CPU-intensive endpoint merging. Because of the different computing types, the two types of tasks can be executed in parallel. The second stage includes step 3 and step 4, the I/O-intensive file reading and the GPU-intensive endpoint matching. Steps in the second stage can only be started after the first stage is completed. In the first stage of executing the rainbow table search with heterogeneous architecture strategy, GPU-intensive step 1 and CPU-intensive step 2 can be executed in parallel and the execution time covers each other. In the second stage, I/O-intensive step 3 tasks are executed in parallel with the GPU-intensive step 4 tasks. The execution time can be covered.

Tasks with running times of 1 and 2 s are defined as task1 and task2. (1) T1 comprises task1, task2, task4, task8, task16, and task32, with a total of 60000 tasks. T1 is chosen to evaluate the system's throughput because it contains various tasks with different execution times, which can simulate the multijob execution in reality. T2 has no calculations and I/O operations and comprises 10000000 "sleep 0" tasks. Essentially, execution overhead per task in T2 is about 0.08 s. T2 was chosen to evaluate the performance of HTDcr when encountering a burst of short-time tasks. (2) The test workloads E1, E2, and E3 comprise task1, task8, and task64, respectively. Each job contains 100000 tasks. E1, E2, and E3 are chosen to evaluate the efficiency of HTDcr. Different execution time tasks are set to observe whether the execution time impacts efficiency. The execution time of each task in E2 is 1 s, the execution time of each task in E2 is 8 s, and the execution time of each task in E3 is 64 s. This covers short-duration tasks, medium-duration tasks, longer-duration tasks, and the evaluation results are more comprehensive. (3) Brute-force password guessing is used as the workload S1. Since S1 is used for evaluating the maximum number of nodes the system can manage, a real-world application has to be used. This way, the user of this application can provide financial support for running the HTDcr for long periods of time on tens of thousands of nodes. The configuration of the workload S2 in this experiment is the same as the configuration of workload T2. S2 is chosen to evaluate the maximum task throughput of HTDcr, very short-time tasks are needed to push the system to the limit of HTDcr. S3 comprises 1000000 short-time tasks to evaluate the performance of the task management system. (4) Since F1 is used for evaluating fault tolerance mechanism of HTDcr, a real-world application must be used in order to verify the correctness of the final result. Brute-force password guessing application is used as the workload F1. (5) BLAST is a gene query program that requires frequent data loading from the storage system. It is suitable for the evaluating lifting effect of the task dispatching strategy designed for shared storage cluster. Rainbow table search is an application with multiple phases having distinct resource necessities. It is suitable for evaluating lifting effect of task dispatch strategy designed for heterogeneous clusters.

7.2 Overall evaluation of HTDcr

7.2.1 *Throughput comparison*

Task throughput can reflect the advantages and disadvantages of the distributed system architecture and the optimization effect. It is an important indicator of the system's ability to handle large-scale HTC applications. In the face of large-scale computing clusters, an inefficient task scheduler significantly degrades the HTC system performance. To compare the throughput of HTDcr with that of HTCondor and SLURM, tests were conducted on Tianhe-2 with 60 nodes and 1200 cores (20 cores per node, and each executor is assigned to a core on a worker node). T1 and T2 are adopted as the experimental workload in this section. The experimental data of HTCondor and SLURM are adopted from our previous work [8].

The results are shown in Figure 5. The theoretical maximum throughput refers to the task throughput achieved under the assumption that the job is executed on a single worker node with 1200 CPUs. The theoretical maximum throughput is calculated as $\text{Throughput}_{\max} = N_t/T_{\text{best}}$, $T_{\text{best}} = T_c/N_c$. N_t represents the total number of tasks, T_{best} represents the theoretical optimal execution time of the job, T_c represents the time required for one core to execute N_t task, and N_c represents the number of processors.

When T1 is submitted through SLURM, the task throughput is 26.74 tasks/s, which is much lower than that of HTCondor and HTDcr. The rationale behind this is that the tasks executed on each node are randomly allocated by SLURM rather than based on the load of nodes and task time, which causes a serious load imbalance. When T1 is submitted to HTCondor and HTDcr, the task throughput is 81.23

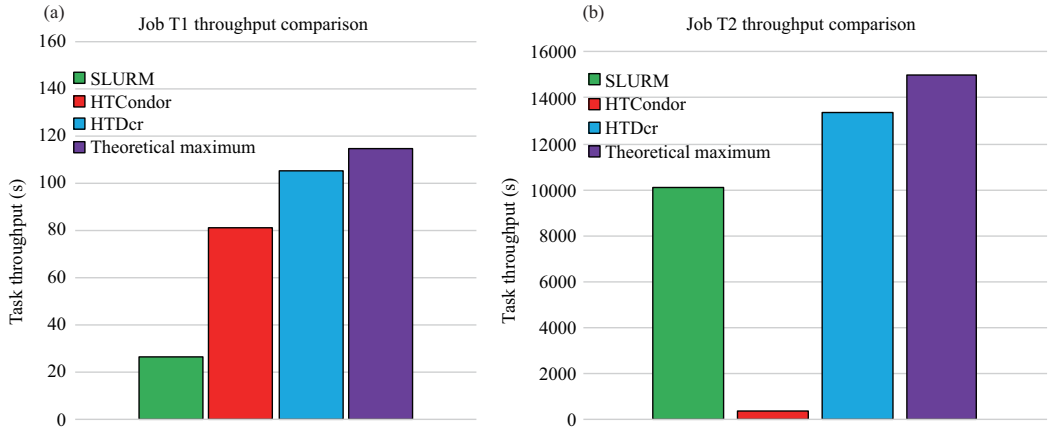

Figure 5 (Color online) Throughput comparison. (a) Job T1; (b) job T2.

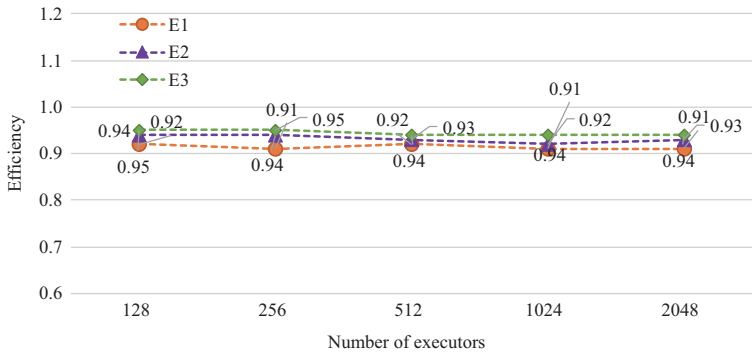

Figure 6 (Color online) HTDcr efficiency.

and 105.15 tasks/s, respectively. The HTDcr throughput is 1.3 times that of HTCondor, which is closer to the theoretical maximum throughput.

Since SLURM cannot directly run such a large-scale job, the experiment is adjusted when submitting the T2 job through SLURM. The task is randomly assigned to each worker node for parallel execution in advance, which is equivalent to concurrently executing the "sleep 0" program on a single worker node. So the SLURM throughput seems to be relatively high when running T2. SLURM frequently creates and destroys processes when performing tasks. However, HTDcr only needs to use threads in the thread pool to obtain tasks from the task buffer, which has a small overhead. Therefore, the HTDcr throughput still slightly exceeds that of SLURM. When T2 is submitted to HTCondor, the task throughput is only 346.26 tasks/s. It is only 1/40 of HTDcr. This trend is mainly because the task scheduler of HTCondor suffers from a severe performance bottleneck when the computing cluster is expanded to a certain scale and the execution time of the task is short. Consequently, many computing processes cannot obtain tasks.

7.2.2 *Comparison of efficiency*

To further explore how task execution time and the number of worker nodes impact the HTDcr performance, efficiency is used to show the evaluation results. The calculation formula for efficiency is given as follows: $E_p = S_p/N$. Speedup is defined as $S_p = T_1/T_n$. Here T_n is the execution time using n executors, and N is the total number of executors. The test workloads E1, E2, and E3 are used herein. The worker nodes gradually increase from 16 to 256 nodes (8 cores per node, and each executor is assigned to a core on a worker node). The number of worker nodes doubles each time.

The experimental results are shown in Figure 6. HTDcr can achieve high efficiencies even with short tasks (91% in the worst case with 2048 executors and 1-s tasks). There is less than a 5% loss in efficiency when the number of executors increases from 128 to 2048. In the worst-case scenario, HTDcr achieves a speedup of 1863 with 2048 executors and task1. With task64 (64-s tasks) and 2048 executors, the speedup is 1905. In contrast, it can be roughly inferred from Figure 5 that HTCondor is inefficient when

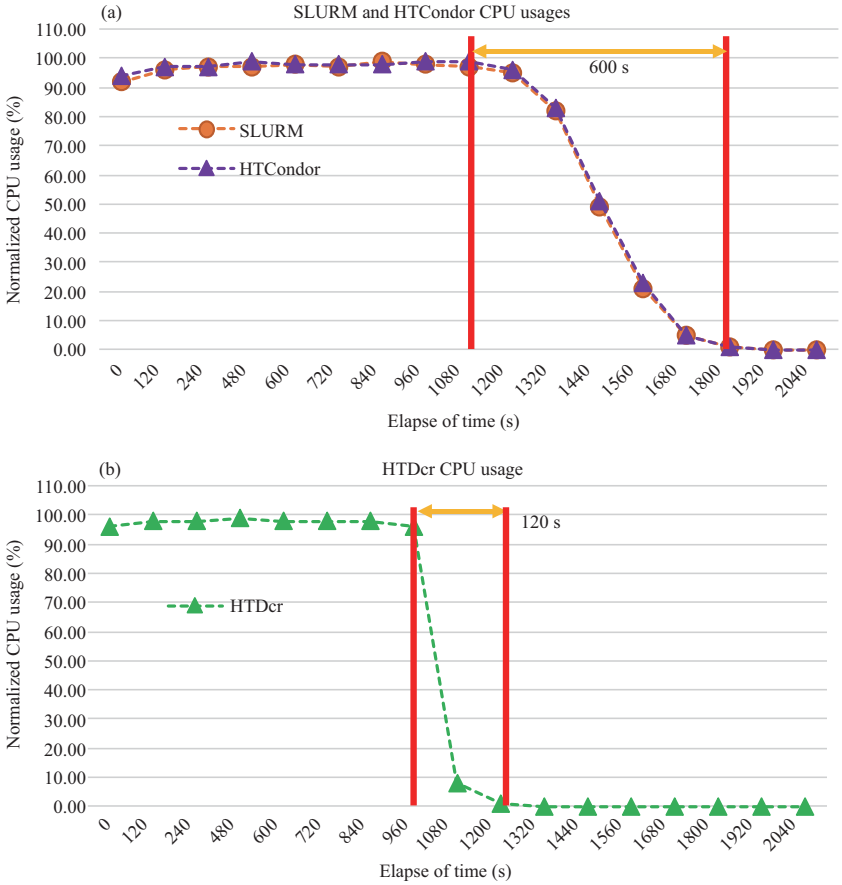

Figure 7 (Color online) Normalized CPU usage of different frameworks during job execution. (a) SLURM and HTCondor CPU usage; (b) HTDcr CPU usage.

facing large-scale and very short tasks. This is also mentioned in Falkon [12].

Figure 7 shows the CPU usage when the E2 load is submitted to HTDcr. The CPU usage information of HTCondor and SLURM in Figure 7 is measured in our previous work [8]. The CPU usage curve of the HTDcr framework is relatively steep, whereas the CPU usage curves of HTCondor and SLURM are more gradual. This is because HTCondor and SLURM have some idle worker nodes before the execution or after the completion of all tasks, respectively. The HTDcr framework uses worker nodes more efficiently. In the later stage of job execution, the CPU usage of all three frameworks is extremely low because most of the tasks are completed, and only a few tasks are still waiting to be executed.

7.3 Evaluations of optimizations for throughput enhancement

7.3.1 *Evaluation on task management system*

To evaluate the performance gains of the task management system, workload **S3** is used. Experiments are conducted on 10 worker nodes. The time spent on basic task operation functions (such as creating, terminating, and moving tasks) is measured, and task operations of the designed task management system and naive task operations are compared. The result is shown in Figure 8. The overall task operation time of the task management system is only 1/3 of the naive task operation time. This greatly improves the affordable throughput of the HTDcr scheduling module.

The maximum task throughput of HTDcr is also evaluated. The configuration of the workload **S2** in this experiment is the same as that of workload T2 in Subsection 4.1. The experiment is conducted on Tianhe-2. Twenty cores are allocated on each node, and the number of nodes is scaled up until HTDcr suffers from a severe performance issue. The worker nodes managed by HTDcr show an obvious load imbalance when the task throughput of HTDcr reaches 30000 tasks/s. This trend means that the scheduling node of HTDcr is the performance bottleneck of the entire system. Therefore, the throughput limit of the HTDcr framework is 30000 tasks/s.

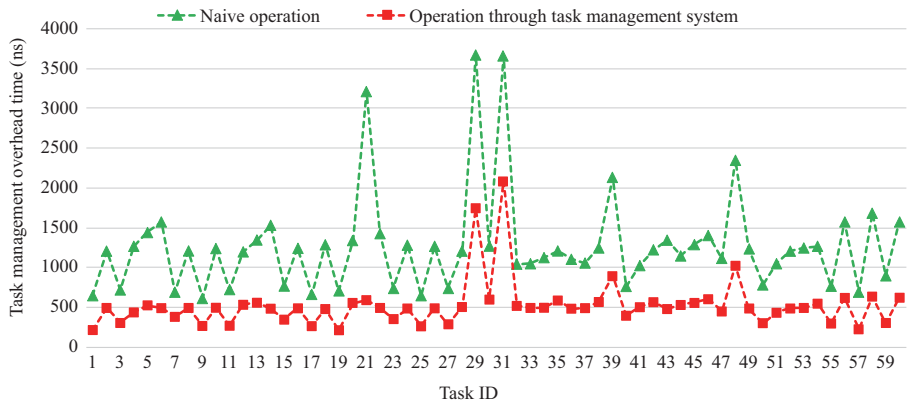

Figure 8 (Color online) Comparison of task operation time. The red and green lines are the latencies of naive task operation and task management systems, respectively.

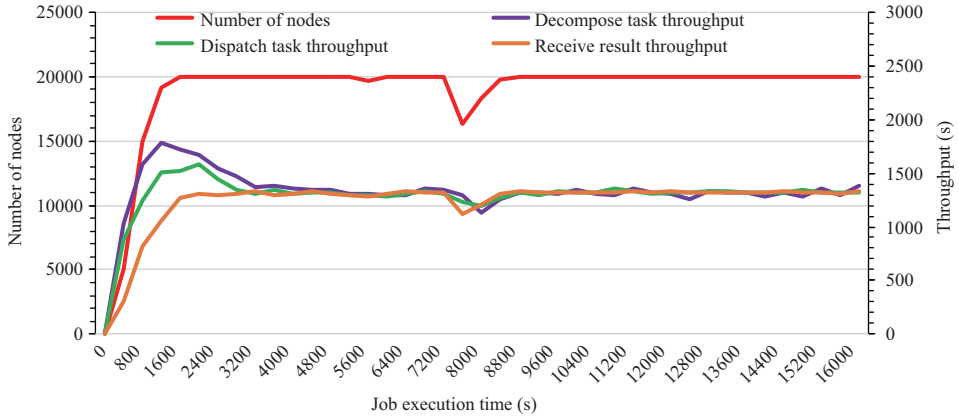

Figure 9 (Color online) Long running on the Sunway TaihuLight.

7.3.2 *Evaluation of scalability of HTDcr*

To test the scalability and robustness of HTDcr, experiments are conducted to push HTDcr to its limits. First, the maximum number of nodes that HTDcr can handle is evaluated. The brute-force password-guessing application is used as the workload **S1** here. The experiment is conducted on the Sunway TaihuLight with 20000 nodes of eight executors per node, giving a total of $20000 \times 8 = 160000$ executors. S1 comprises 29 different jobs. Each job calculates and matches a batch of passwords using different Hash functions (such as MD5 [24] and SHA1 [25]). Every job is broken down into 100-s tasks.

Figure 9 shows that the tasks sent by the HTDcr distributor (green line) are roughly equivalent to the number of tasks decomposed by the decomposer (purple line). After the number of nodes managed by HTDcr stabilizes, the results received by HTDcr (orange line) are at about the same rate as the tasks sent and decomposed by the distributor and decomposer, respectively. The throughput is 1300 tasks/s. The red line represents the number of worker nodes managed by HTDcr, which increases from 0 to 20000 in 1600 s. According to our measurement, the memory usage of the scheduling module is kept at 4 GB, and the memory of a single node on the Sunway TaihuLight is 32 GB. This shows that HTDcr can manage at least 20000 nodes, and there is room for further increase.

7.4 Evaluation of scheduling strategy

Here, two real-world applications are used to evaluate the performance gain of cluster task scheduling, which integrates with application and hardware. The first test is conducted on Tianhe-2 with eight nodes (one executor per node). As shown in Figure 10, for applications, such as BLAST, that require frequent data loading from the storage system, the strategy designed for shared storage has obvious performance improvement over the SQF policy. According to our experiments, the task-scheduling integration with the application and cluster architecture (share storage strategy) increases the performance of the blast application by 1.7×. This is because a shared storage strategy can effectively leverage the memory as a

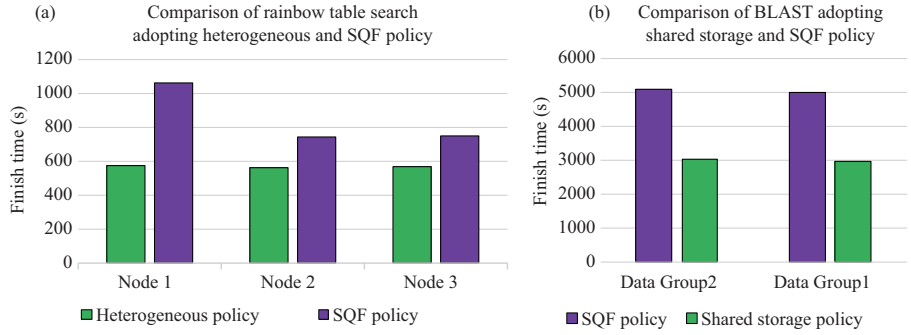

Figure 10 (Color online) Evaluation of task scheduling policy of HTDcr. (a) Evaluation on heterogenerous policy; (b) evaluation on shared storage policy.

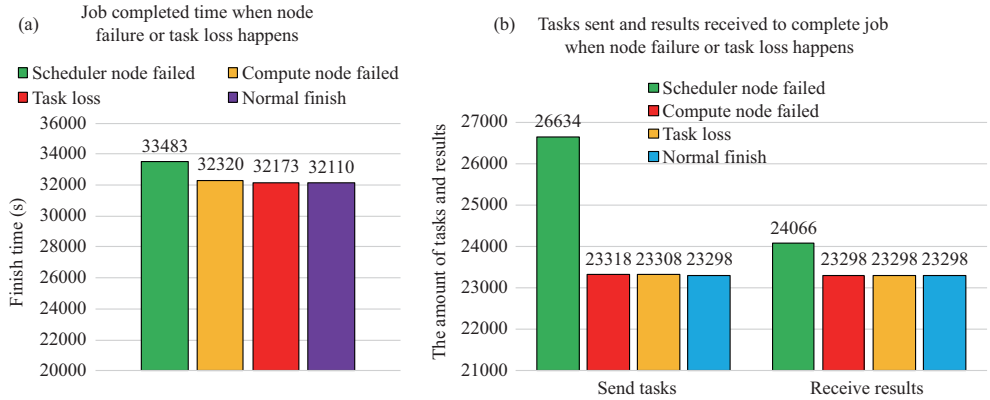

Figure 11 (Color online) Evaluation of fault tolerance mechanism. (a) Evaluation on job completed time; (b) evaluation on tasks sent and result received.

cache for redundant data on the shared storage cluster, which mitigates the performance penalty brought by the I/O. The second test is conducted on Tianhe-2 with three nodes and 24 executors (eight executors per node). The result is shown in Figure 10. The heterogeneous cluster strategy helps the performance of the rainbow table search application increase by 1.9× over the default SQF policy. For applications, such as the rainbow table search, which comprises different types of tasks, the integration of task scheduling with application and hardware has several performance benefits.

7.5 Evaluation of usability of the framework

7.5.1 *Evaluation of fault tolerance mechanism*

This experiment tests the fault tolerance mechanism of HTDcr. The brute-force password-guessing application is used as the workload **F1** in the experiment. The experiment uses F1 as a workload to compute the MD5 Hash function on Tianhe-2 with 256 worker nodes (1 executor per node, and the executor is assigned to run on the V100 GPU of the worker node). Job F1 is divided into 23298 tasks, and the approximate running time of each task is 300 s. The faults are manually simulated and created.

As shown in Figure 11, (1) if no failure or packet loss occurs, the number of tasks sent by HTDcr is equal to the number of results returned, both being 23298. (2) The timeout threshold for 10 tasks is set to 100 s, which is shorter than the preset task loss time. When these tasks are incomplete on the worker nodes, the scheduling module judges these 10 tasks as lost and resends them. So 10 more tasks are sent compared with all results received (23308 vs. 23298) when the job is completed. (3) The computing module processes on two worker nodes are killed and restarted after 30 s. When the scheduling module does not receive heartbeat packets from the computing module processes, these two worker nodes are judged to be down. Furthermore, the uncompleted tasks on these two nodes are resent to other nodes. Since the task buffer size on each node is 10, the final number of tasks sent by HTDcr is 20 more than the number of results received (23318 vs. 23298). (4) The scheduling module process is killed in half of the backup cycle (15 min). When the process is restarted, all tasks in the task buffer of the worker nodes are resent (256×10). The completed tasks (256×3) in the first 15 min of the latest backup cycle

Table 2 Test cases to verify the multijob resource allocation system

Cases	Case1				Case2			
Group	G1	G2	G3	G4	G5	G6	G7	G8
lb	800	800	400	400	800	800	400	400
P (s)	4096	4096	4096	4096	4096	20480	4096	4096
Estimated time (s)	12228	12228	16384	16384	12288	32768	20480	20480
Actual finish time (s)	11033	10759	15000	14917	10668	29530	18485	18675
Relative error (%)	11	14	9	10	15	11	11	10

are recalculated because the results have not been saved into a file. Thus, 3328 tasks are resent, and 768 results are received again, coinciding roughly with the test results (26634 vs. 24066).

The completion time of the F1 workload in four experiments is shown in Figure 11. The task loss and worker node failure slightly affect the job completion time because lost tasks are quickly reassigned to other worker nodes. The time loss caused by the failure of the scheduling module is 1300 s. These include the time loss of unsaved results in the first 15 min of the backup cycle and the time consumption of manually restarting the process. The time loss only accounts for 4% of the total execution time.

7.5.2 *Evaluation of multi-job resource allocation of HTDcr*

This subsection mainly focuses on resource allocation among multiple jobs. The experiment is designed to verify whether the resource allocation mechanism for multiple jobs works well. The brute-force password-guessing application is used as the workload **R1** in the experiment. MD5 is the Hash function used by R1. R1 is deployed on a cluster with four nodes; each node is equipped with 8 NVIDIA GTX 1080, and each executor is assigned to run on the GPU of a worker node. Each worker node has two Xeon E5-2609 v3 CPUs running at 1.90 GHz. Each worker node has 128 GB of memory.

Table 2 shows the test cases in the experiment. The first test case contains four jobs with exactly the same content. Also, lb is the minimum number of tasks that a job can send in a scheduling cycle, which is discussed in Subsection 3.5. P is the time required when a job is performed using all computing resources. The computational complexity of a job can be seen from the value of P. The P value of each job is 4096 s in the first test case. Each job is divided into 2185 tasks, and the approximate running time of each task is 60 s. The second test case also contains four jobs. The P value of G6 in the second test case is five times that of other jobs, which takes 20480 s. G6 is divided into 4370 tasks; the approximate running time of each task is 120 s. The other jobs are the same as the jobs in test case 1. The relative error between the actual (act) and estimated (est) execution times can be calculated as $|\text{est} - \text{act}|/\text{act}$. The smaller the relative error between estimated and actual execution times, the better the performance of our resource allocation mechanism. The estimated execution time is estimated using Algorithms 3 and 4.

The results of our experiments are presented in Table 2. The actual execution time is quite close to our estimated time in all test cases: the maximum relative error is 15%, and the average relative error is 11%. The experiment results of the multijob resource allocation system show that HTDcr can support the simultaneous execution of multiple jobs, and the efficiency loss of the multijob execution is small, which meets the requirement of an HTC framework to deploy multiple jobs.

8 Conclusion

Herein, we propose a job execution framework HTDcr targeting high-throughput applications. We focus on improving the throughput and usability of the framework. Various optimizations and designs are implemented to improve the throughput, including a fine-tuned task management system and hierarchical scheduling. The efforts to improve usability include a programmable workflow (decompose, compute, and reduce) and task-level fault tolerance. The evaluations on both microbenchmarks and real-world applications show that HTDcr can achieve outstanding throughput and scalability on the Tianhe-2 and TaihuLight supercomputers. The evaluation of usability also yields results in line with design expectations.

Acknowledgements This work was supported by National Key R&D Program of China (Grant No. 2021YFB0301300), National Natural Science Foundation of China (Grant No. U1811461), Zhejiang Lab (Grant No. 2021KC0AB04), Major Program

of Guangdong Basic and Applied Research (Grant No. 2019B030302002), Program for Guangdong Introducing Innovative and Entrepreneurial Teams (Grant No. 2016ZT06D211).

References

1 Shendure J, Ji H. Next-generation DNA sequencing. Nat Biotechnol, 2008, 26: 1135–1145
2 Houshmand S, Aggarwal S, Flood R. Next gen PCFG password cracking. IEEE Trans Inform Forensic Secur, 2015, 10: 1776–1791
3 Ward L, Sivaraman G, Pauloski J G, et al. Colmena: scalable machine-learning-based steering of ensemble simulations for high performance computing. In: Proceedings of the IEEE/ACM Workshop on Machine Learning in High Performance Computing Environments (MLHPC), 2021. 9–20
4 Casalino L, Dommer A C, Gaieb Z, et al. AI-driven multiscale simulations illuminate mechanisms of SARS-CoV-2 spike dynamics. Int J High Perform Comput Appl, 2021, 35: 432–451
5 Caporaso J G, Lauber C L, Walters W A, et al. Ultra-high-throughput microbial community analysis on the Illumina HiSeq and MiSeq platforms. ISME J, 2012, 6: 1621–1624
6 Fajardo E M, Dost J M, Holzman B, et al. How much higher can HTCondor fly? J Phys-Conf Ser, 2015, 664: 062014
7 Karo M, Lagerstrom R, Kohnke M, et al. The application level placement scheduler. Cray User Group, 2006, 1–7
8 Yu W, Shen Y X, Li L, et al. Teno: an efficient high-throughput computing job execution framework on Tianhe-2. In: Proceedings of the 20th International Conference on High Performance Computing and Communications, 2018. 408–415
9 Thain D, Tannenbaum T, Livny M. Distributed computing in practice: the Condor experience. Concurr Comput-Pract Exper, 2005, 17: 323–356
10 Yoo A B, Jette M A, Grondona M. SLURM: simple Linux utility for resource management. In: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing, 2003. 44–60
11 Goiri í, Le K, Haque M E, et al. Greenslot: scheduling energy consumption in green datacenters. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2011. 1–11
12 Raicu I, Zhao Y, Dumitrescu C, et al. Falkon: a fast and light-weight task execution framework. In: Proceedings of the ACM/IEEE Conference on Supercomputing, 2007. 1–12
13 Raicu I, Zhang Z, Wilde M, et al. Enabling loosely-coupled serial job execution on the IBM BlueGene/P supercomputer and the SiCortex SC5832. 2008. ArXiv:0808.3536
14 Merzky A, Turilli M, Titov M, et al. Design and performance characterization of RADICAL-Pilot on leadership-class platforms. IEEE Trans Parallel Distrib Syst, 2021, 33: 818–829
15 Hagras T, Janeček J. Static vs. Dynamic list-scheduling performance comparison. Acta Polytech, 2003, 43: 6
16 Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters. Commun ACM, 2008, 51: 107–113
17 Shvachko K, Kuang H, Radia S, et al. The hadoop distributed file system. In: Proceedings of the 26th Symposium on Mass Storage Systems and Technologies (MSST), 2010. 1–10
18 Hursey J, Squyres J M, Mattox T I, et al. The design and implementation of checkpoint/restart process fault tolerance for open MPI. In: Proceedings of the IEEE International Parallel and Distributed Processing Symposium, 2007. 1–8
19 Hammersley J. Monte Carlo Methods. Berlin: Springer, 2013
20 Liao X, Xiao L, Yang C, et al. MilkyWay-2 supercomputer: system and application. Front Comput Sci, 2014, 8: 345–356
21 Fu H, Liao J, Yang J, et al. The Sunway TaihuLight supercomputer: system and applications. Sci China Inf Sci, 2016, 59: 072001
22 Ye J, McGinnis S, Madden T L. BLAST: improvements for better sequence analysis. Nucleic Acids Res, 2006, 34: 6–9
23 Hellman M. A cryptanalytic time-memory trade-off. IEEE Trans Inform Theor, 1980, 26: 401–406
24 Rivest R. RFC1321: The MD5 Message-digest Algorithm. RFC Editor, 1992
25 Eastlake D, Jones P. RFC3174: US Secure Hash Algorithm 1 (SHA1). RFC Editor, 2001