2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th Intl. Conference on Data Science and Systems

Teno: An Efficient High-Throughput Computing Job Execution Framework on Tianhe-2

Wei Yu[1], Yi-xian Shen[2], Lin Li[1], Yun-fei Du[2]✉, Zhi-guang Chen[1, 2],Yu-tong Lu[1, 2]

[1]*College of Computer, National University of Defense Technology, Changsha 410073, China*

[2]*National Supercomputer Center in Guangzhou, School of Data and Computer Science, Sun Yat-Sen University, Guangzhou 510006, China*

***Abstract*—Large-scale and loosely-coupled applications cannot be implemented directly on high-performance computing platforms. At the same time, the deployment and maintenance of high-performance computing and high-throughput computing will result in the waste of computing resources. In order to solve the problem that existing resource management systems cannot make high-throughput computing applications execute efficiently on high-performance computers, we propose, design and implement a high-throughput computing job execution framework Teno without modifying the existing configuration environment of Slurm on Tianhe-2. It uses Slurm to implement fine-grained resource scheduling through the idea of hierarchical scheduling, and optimizes the traditional Master-Worker model, thereby speeding up the high-throughput operation and increasing the effective utilization of cluster resources. Effective fault-tolerance mechanisms such as fault recovery and error retry are also implemented. Finally we design various experiments in Tianhe-2 to test and evaluate the key factors for high-throughput calculations of Teno, Slurm and HTCondor, and analyze in detail why the performance of Teno is over that of the other two.**

Keywords—High-throughput Computing, High-performance Computing, Loosely-coupled, Job scheduling

I. INTRODUCTION

High-throughput computing refers to costing large amounts of computing resources to execute a long-term computing task [1, 2, 3]. European Grid Infrastructure defines high-throughput computing as a computing paradigm that focuses on the efficient execution of enormous loosely-coupled tasks[4]. Compared to high-performance computing, high-throughput computing does not focus on the computing power per second. The key point is making the best of computing resources effectively and providing robust and reliable computing services in the computing environment with unstable and unreliable factors.

Modern scientific computing tends to analyze the big, large-scale, loosely-coupled data, such as high-throughput material computing[5,6], gene sequencing in the biomedical field[7], MARS applications in the field of economic models[4], OOPS applications in the chemical field[9] and so on.

Tianhe-2 is a group of heterogeneous supercomputer located in National Supercomputer Center in Guangzhou with capable of a peak performance of 54.9 PFlops and sustained performance of 33.9 PFlops. Although Tianhe-2[10] has a powerful computing capability, its architecture is mainly designed for the intensive applications implemented by the messaging paradigm (such as MPI). Tianhe-2 can't make use of computing resources effectively, resulting in the execution speed and quality of operations, which unable to satisfy user's requirements.

Slurm[11,12] is the cluster resource management and job scheduling system on Tianhe-2, which is equipped with excellent fault tolerance and scalability, and widely adopted in HPC around the world. It uses an exclusive resource scheduling way and allocates resources with the computing node as a unit. That is because the operation of Slurm requires managing and scheduling of all computing resources and tasks by the central process Slurmctld. However, there are more than 17,000 nodes in Tianhe-2. If scheduled by cores, it will not only lead to a high load on the management process, but also affect the user experience greatly. Large-scale resource manager (LRM) is intent to schedule resources by cores based on Slurm. Therefore, the method of performing high-throughput computing on Tianhe-2 is to submit the job script of the specified process by means of "batch processing", and then distribute the sub-tasks to each computing node for execution randomly. Although this method can improve the utilization of computing resources to some extent, it is more difficult to achieve load balancing in scripts. In addition, since Tianhe-2 cannot monitor each computing task with fine-grained granularity, if an unexpected event such as a Slurm system crash or computing node downtime causing the running job to be terminated, the user can only re-run the entire job. It will waste of computing resources and time. This paper aims to provide a robust and reliable high-throughput computing framework Teno on Tianhe-2, which promotes Tianhe-2 to perform better in loose-coupling high-throughput computing applications. The main contributions of Teno are as follows:

- Realize hierarchical scheduling of computing resources by means of Slurm. The outer layer uses Slurm for scheduling in computing nodes. The inner layer uses the HTCTeno subsystem to execute the kernel dispatch by cores.
- Optimize the task distribution mode from "PUSH" mode to "PULL" mode based on Master-Worker, which greatly boosts the high-throughput operation and increases the adequate utilization of cluster resources.
- Organize and construct the job by task of forest in the engine subsystem, solving the problem of describing large-scale operations.

978-1-5386-6614-2/18/$31.00 ©2018 IEEE
DOI 10.1109/HPCC/SmartCity/DSS.2018.00085

- Design the monitoring subsystem to implement fault recovery and task retry mechanism, ensuring the stability of high-throughput computing and reliable execution.

We evaluate the throughput, efficiency, scalability and fault tolerance by 4 groups of apple-to-apple experiments, which demonstrated that the task throughput of Teno can outperform HTCondor by 10x, the task execution efficiency is 55.08% higher than that of Slurm, and the computing efficiency is improved by 43.12% compared with Slurm. When the task reaches the maximum scale of 10 million, 12,000 processors perform computational tasks with a run time of 16 seconds, and the effective utilization of computing resources can reach more than 90%, which significantly improved utilization of compute nodes.

This paper is organized as follows. Section II briefly introduces and discusses other systems for large-scale computing tasks. Section III shows the architecture design, module components and implementation of Teno are presented. Section IV describes experimental results, demonstrates and compares with other systems to draw conclusions in the real environment. We summarize the paper and point out the direction of further works in Section V.

II. Related Works

HTCondor is a software system that creates a high-throughput computing environment [13, 3, 14]. It effectively uses the computing power of network communication and idle computing resources. It is often used to manage workloads, cluster resources and idle desktop computers. Because it faces unreliable resources, HTCondor can switch between dedicated cluster resource processing mode and personal desktop computer processing mode, manage all computing resources in an integrated manner, and construct reliable computing services on reliable computing resources and unreliable computing resources [15,16]. The workflow management system Pegasus[16,17] implemented on the basis of HTCondor DAGMan[18] maps workflows to directed acyclic graphs first, then converts the abstract graph information to executing workflows that include available storage, available computing resources, input data as well as executing programs and other attributes. DAGMan in HTCondor finally performs the specified sequence of tasks based on the workflow information submitted by Pegasus.

Apache Spark [19,20] is a general-purpose parallel computing framework originally developed by the AMPLab at the University of California, Berkeley. Unlike Hadoop MapReduce [21, 22, 23], which writes intermediate data to disk while the job is running, Spark stores the intermediate results in memory, which is 100 times faster than Hadoop MapReduce. The architectural foundation of Apache Spark is the elastic dataset, which is a simple extension of the MapReduce [24] model, whose purpose is to use an efficient data sharing concept and MapReduce-like operations to handle computational tasks that are not suitable for MapReduce (eg, Iteration). RDD has fault tolerant and parallel data structure features, which allows the user to specify whether data is stored to the hard disk or memory, control the partitioning method of the data, and perform various operations on the data set. In addition, RDD provides a coarse-grained transformation interface that records the data set transformation process without storing intermediate data, thereby achieving efficient fault tolerance.

Ioan Raicu proposed MTC (Many-Task Computing) in grids and supercomputers [25] in 2008. The purpose of many-tasking is to compensate for the differences between the two computational paradigms of high-performance computing and high-throughput computing. Many-task computing is derived from high-throughput calculations. It has large amount of calculations and many tasks. The difference from high-throughput computing is that the former uses a lot of computing resources to complete many calculation tasks in a short time, and the later focus more on long-term, reliable and stable computing cability[26,27].

Ioan Raicu and Yong Zhao design and implement a lightweight task execution framework Falkon[28]. After Microbenchmarks testing, Falkon uses 24,000 actuators to complete 2 million tasks in 2 hours, with a maximum task throughput of 440 tasks/sec.

III. Architecture and Design

Teno is divided into four subsystems: Execution Engine (**DE**), Hierarchical Scheduling (**THUS**), High-throughput Computation (**HTCTeno**), and Monitor (**MON**). This section we will first describe the design of these four subsystems. Based on this, we will describe the operation flow of the entire system. The system architecture is shown as Fig.1.

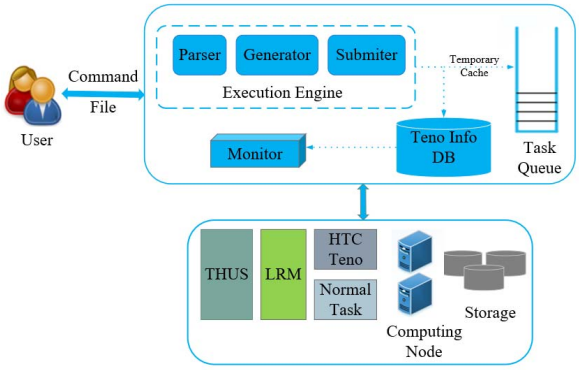


Fig.1 The Diagram of System Architecture

A. DE

Execution Engine subsystem is designed and implemented to solve the problem that the task relationship is difficult to describe and the execution sequence of the task is difficult to control in a large-scale jobs. It directly provides an interface for describing job types and building tasks, and finally describes the entire job in a JSON format string that is more general. For the problem of how to describe the number of millions and even tens of millions of tasks, we have proposed two new concepts after observing a large number of loosely-coupled applications: **Missions** and **Task Forests**.

Mission: A type of tasks that correspond to multiple different input files.

Task forest: The way to describe the dependencies between missions.

DE is further divided into three components based on the function: a job information parser (**Parser**), a task generator (**Generator**), and a task submitter (**Submitter**). Parser provides the user with a programming interface for describing the job information. Generator generates task information in a specified format according to the job description information provided by Parser. Submitter is a multi-threaded program that will perform tasks generated by Generator. Information is sent to the task pool concurrently in the order of priority. Here is their specific work process.

Parser first converts the job information constructed by the user through the interface into a task forest, and then uses the improved breadth-first search algorithm to generate the job description information in a specified order according to the mission priority. Each task node in the task forest has a mission. Since the parent task node can be executed only after it completes the execution, there is a strong dependency relationship between the layers in the forest. But the task nodes in a single layer have no coupling relationship and can be completely parallel. The execution order of the job is from the root node to the child node and finally to the leaf node.

Generator is mainly to generate task information in a specified quantity and format in the order that has been designed according to the job description information and the specified task format generated by Parser, and to give each task a unique identifier. The key point is that Generator does not generate all the task information at one time. Although this can solve the problem of memory overflow through persistent storage, the task information should be read from the disk when Submitter needs it. This frequent reading and writing of the disk will bring a lot of IO overhead. In order to avoid the above problem, after Generator outputs a certain amount of task information, we "suppress" the execution of Generator and record the progress of execution until Generator is explicitly triggered again.

Submitter is a multi-threaded program that submits the tasks generated by Generator concurrently in the priority order to the task pool. It is not only a simple task submitter, but also responsible for scheduling task execution based on the task's priority and mission's dependencies and the use of computing resources.

B. THUS

The primary design principle of Teno is not to modify the configuration environment of Tianhe-2. Based on this, we propose to use hierarchical scheduling. The core idea is that after a job is submitted through Teno, Slurm is used to allocate the entire block of computing nodes to the job, and then build a computing environment that is more suitable for large-scale, loosely-coupled application on these nodes. Thereby a core-based resource scheduling strategy and a task-based work management mechanism are realized.

The function of THUS is to configure the required environment variables, run HTCTeno, and finally build a high-throughput job execution environment on the computing nodes that the cluster resource management system allocates for the job. Because the demand for computing resources for each job is generally different, and the computing nodes assigned to the job by the cluster resource management system are not exactly the same each time, we need to build a high-throughput computing environment dynamically during job execution. Only on the computing nodes that the cluster resource management system allocates for this job, we focus on how to integrate Teno and Slurm. Batch mode on Slurm allows users to distribute work to each computing node through executable scripts. So we make Teno read the environment information required for the job first and then dynamically generate a customized executable script.

The specific work flow of THUS is as follows:

1. Get the head node of the compute node allocated by Slurm, use it as the task pool node and record its host name and port number to communicate with other worker processes.

2. Set the Python path and Python execution program path and environment variables to execute job.

3. Configure and starts the task pool program based on the job size and configuration file entered by the user.

4. Start the task executor on each computing node. The task executor is responsible for concurrently acquiring and executing tasks.

There is a key question that after all programs are completed, Slurm considers the script to have completed execution, thus killing all processes on the computing nodes and retrieving the computing resources. However, in reality, Teno is performing efficient, concurrent loose-coupling tasks. The solution is to run a program that periodically queries the job status on the head node until all tasks completes, then to run the cleanup function and releases resources.

C. HTCTeno

HTCTeno focuses on two key points when handling large-scale, loosely-coupled applications. One is the throughput of the system. The other is the effective utilization of computing resources. So it improves and optimizes the process of distributing tasks from the main process to each working process. HTCTeno separates the management and scheduling tasks from the main process and combines them with the task submitter in the execution engine subsystem. The traditional "Master process" becomes the task submitter and the task pool consists of a distributed message queue. The "PUSH" mode of task distribution is subsequently modified to "PULL" mode. Since there is no main process to uniformly manage computing resources and the state of task execution, there is a task execution and management process called **HTCTenod** on each of the computing nodes.

Slurm assigns computing nodes to jobs, then HTCTeno manages and schedules these nodes by cores. According to the purpose, the computing resources can be divided into two types. One is the **task pool**, which is mainly used to run the distributed message queue, and can adjust the amount of computing resources allocated to the task pool dynamically based on the scale size of the operation. The data in the task pool is not only the task information needed to execute the job but also the task execution status. Since the task submitter needs to determine whether the task can be executed according to the execution status of the dependent task group, the task pool needs to receive the return of the task execution. As a result, the task group's

status information is maintained at regular intervals and provided to the task submitter. Another way to use resources is the **computing process pool** that used to perform tasks. HTCTeno starts HTCTenod process to execute and manage tasks on each node containing a calculation process. On the one hand, the HTCTenod process communicates with the task pool and acquires the task information to be executed. On the other hand, the HTCTenod process manages the computational resources and computational tasks on the computing node, and concurrently executes the task.

We observe that there are three cases in which computing resources are in idle status while Teno executes jobs:

1. When the current task execution is completed, and the HTCTenod process is communicating with the task pool to obtain new tasks.

2. When the task type is data-intensive and there is a large number of IO steps and no other task competes with the current execution.

3. When the job execution is drawing to a close and no tasks in the task pool to be executed.

The third situation is the normal phenomenon of the job execution, so we only analyze and deal with the first and second cases. For the first case, we let the HTCTenod process take a group of tasks each time in the local task queue. When the number of remaining tasks in the task queue is lower than the threshold, the HTCTenod process fetches another group of tasks from the task pool until the job execution completes. Too many tasks in a group may cause result undesired while too few may cause some nodes to have no tasks to execute and all of them are idle. We will later discuss this phenomenon through experiments. The second case is simple. For data-intensive applications, we start more processes concurrently than CPU cores on the compute nodes. The optimal number of processes is related to the specific application.

D. MON

Teno needs to do more than just perform large-scale, loosely-coupled applications on high-performance computing systems. More importantly, it need to provide robust and reliable high-throughput computing services, including fine-grained task management and resource monitoring to achieve failure recovery and error retry mechanism.

The functions of the monitor subsystem are performed based on the task, including the execution status of the monitoring task, the resource usage of the monitoring task, the progress of the recording job, and the reason of the error. Therefore, MON is inseparable from other subsystems. On the one hand, the execution information of the task is obtained through other subsystems, and on the other hand, status information and resource usage of the task are provided for other subsystems.

When Slurm crashes or fails, the computing resources executing the job are all retired and the job is interrupted, but the job execution information has been saved. After the failure is repaired and the computing environment returns to normal, the user can choose to re-execute the job that was interrupted by the exception, or to resume operation after the job is restored to the state before the interruption. If users choose to continue the interrupted job, they need to specify the job ID when submitting the job and execute the job in "recovery" mode.

The key is to find and handle the "corrupted mission". The "recovery" mode directly skips Parser and generates task information based on the recorded job information. However, Generator does not generate all the task information. It traverses the state of the task group one by one until it finds the task group being executed. The task is the very "damage mission." The tasks in the "damage mission" have three states: **Completed**, **Running**, and **Not Generated**. It skips the completed task and re-execute the running task. There is no difference between the processing of the abnormal task and the normal mode. The remaining missions are performed in normal mode.

We implement a task retry mechanism based on fault recovery. During the execution of the task, it may fail because of some accidental factors, and the user may not get the expected result. So we capture the cause of the failure of the task execution and re-inject the task with the relevant error into the task pool according to the user's configuration.

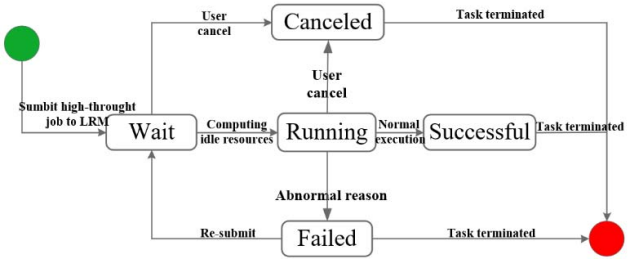


Fig.2 Task State Machine

The state machine of the task is shown in Fig.2. The task waits after the job is submitted until it starts execution. During this time, if the user cancels the execution of the job or a specific task, the task is marked as a canceled state. If there is an idle computing resource, HTCTenod gets the task information from the task pool initiatively, then sends it to the compute node for execution and marks the task as the state of "**running**". If the running task is canceled by the user, the state of the task is modified to "**canceled**". If the task is completed normally, the status is modified to be "**successful**". And if the task fails to execute, the status is modified to "**failed**".

In general, "canceled", "successful" and "failed" all belong to the end state, that is, there is no other process following after the task reaches these states. However, there are exceptions to the "failed" state. If a task encounters an interrupt due to an accidental hardware problem during execution, the task can be re-executed in the task pool. The state of the task group is similar to that of the task. The difference is that the state of the task group can be marked as "successful" only when all the tasks in the task group are successfully executed. But as long as one task in the task group fails to be executed, the mission is failed.

E. System execution flow

The execution flow of Teno is shown in Fig.3. The flow can be described as follows:

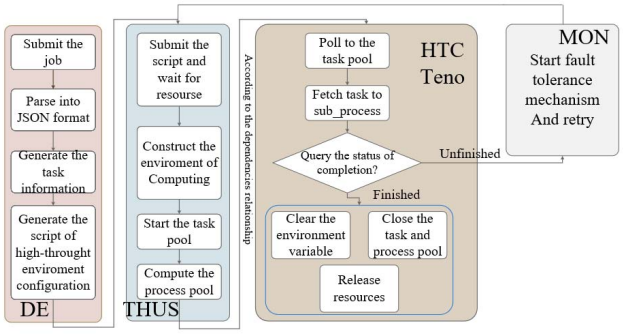


Fig.3 Teno Execution Flow

1. The user uses the interface provided by Parser of DE to build jobs from different types of missions based on their dependencies.

2. Parser replaces the job description class with a more general JSON format string.

3. Generator parses the string and waits for the task committer to trigger to generate specific task information.

4. A high-throughput environment configuration script is generated according to the job configuration, and submitted to the cluster resource manager.

5. The cluster resource manager allocates computing resources and builds a high-throughput computing environment on the computing node, at the same time starts the task pool and the computing process pool.

6. Submitter triggers Generator to form new task information, and then pushes the task to the task pool according to the dependencies between the tasks.

7. HTCTenod process in the computing process pool fetches tasks from the task pool according to the executing status of tasks on the respective computing node and submits them to sub-process to execute.

8. Mon monitors job execution in real time and starts the failure recovery or error retry mechanism when the job is interrupted unexpectedly

9. The above process is executed in a loop until all the missions in the job are finished. Then the environment variables are cleared, the task pool and the calculating process pool are closed, and the task resources allocated by the cluster resource manager are released.

IV. Experiments and Analysis

We will test the performance of Teno in three aspects: task execution throughput, efficiency, and scalability. In addition, the system's fault tolerance mechanism will also be verified. All experiments were conducted on Tianhe-2. The test workloads of different indicators are also different. We will introduce the workload, experimental procedures and comparison methods in each test.

A. Task Execution Throughput

In order to fully measure distribution speed and execution efficiency of the task, we design two jobs *Job_t_1* and *Job_t_2* as the workload of the test in this section. The *Job_t_1* has no calculations and IO at all and consists of 10 million "sleep 0" programs.

We define **Task1** with a running time of 1 second, **Task2** with a running time of 2 seconds, **Task4** with a running time of 4 seconds, and so on.

Job_t_2 consists of Task1, Task2, Task4, Task8, Task16, and Task32, each with 10,000 tasks, totaling 60,000 tasks. The size of computing resources is limited to 60 computing nodes and 1200 cores. At the same time, we use HTCondor and Slurm to perform the same workload. A total of 10 experiments are conducted in this test, and the average value is finally taken. The experimental results are shown in Fig.4.

Since the throughput of task execution is not only related to the number of tasks, the number and status of computing resources, but also affected by the task itself, the results of *Job_t_1* and *Job_t_2* are not directly compared in this experiment. From Fig.4 (a), we can see that when the *Job_t_1* is submitted and executed by Slurm, the average throughput of the task reached 10101.01/s, far more than that of Teno and HT Condor. That is because when we use Slurm to execute large-scale applications, the task is randomly distributed to each computing node in advance and executed in parallel, while the tasks in *Job_t_1* are all the "sleep 0" program without calculation and communication, which is equivalent to executing "sleep 0"concurrently on a single compute node. Under the same conditions, the task throughput of HTCondor is only 346.26/s, mainly because of the task distributor speed bottleneck. That is, when the available computing resources reach a certain scale while the task execution time is short, a large number of computational processes do not acquire tasks and are idle. Teno performs *Job_t_1* with a task throughput of 3378.38/s in the same experimental environment, nearly 10 times that of HTCondor, achieving the desired effect of improving the task distribution process. It is only 1/3 of that of Slurm, mainly because it takes a lot of time for Generator to build task information and communicate between task pool and computing process pool. This is also our further work to optimize.

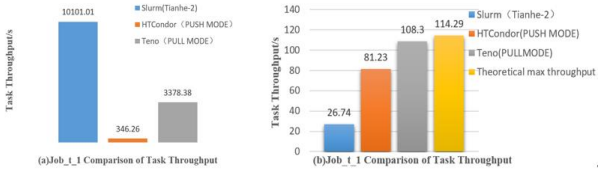


Fig.4 Tests for System Maximum Throughput

Compared to *Job_t_1* that simply focuses on the design and implementation performance of the system architecture, *Job_t_2* focuses more on the capabilities of system's resource scheduling and load balancing. The difference between these two workloads can be seen in Fig.4 (b). In addition to comparing the three systems, Fig.4 (b) also gives the theoretical maximum throughput values. The theoretical maximum throughput refers to the task throughput that can be achieved by executing jobs in parallel when a single computing node with N CPUs. The theoretical maximum throughput is calculated as:

$$\text{Throughput}_{max} = N_{task}/T_{best} \quad (1)$$

$$\text{T}_{best} = \text{T}_c/N_{cpu} \quad (2)$$

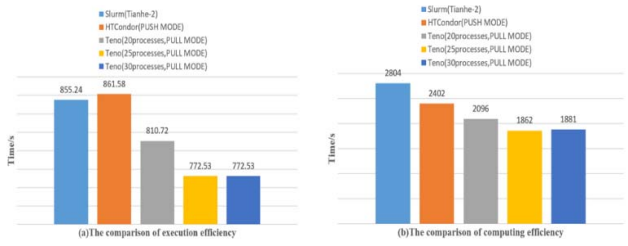
Among them, N_{task} represents the total number of tasks, T_{best} represents the best theoretical execution time of the job, T_c represents the time required for a single processor to execute N_{task} tasks, and N_{cpu} represents the number of processors.

Slurm reaches the task throughput of 27.78/s on executing workload *Job_t_2*, which is far lower than that of HTCondor and Teno. That is mainly because when we use Slurm to perform large-scale operations, the tasks executed by each computing node are randomly assigned, and nodes cannot exchange tasks with each other, serious load imbalance is caused. In the practical application scenario, the user cannot obtain the exact execution time of each task in advance and reasonably cannot allocate the task according to the performance of the computing node. When *Job_t_2* is submitted and executed through HTCondor and Teno, the task throughput reaches 81.23/S and 106.3/S, respectively, where Teno is closer to the theoretical task throughput.

B. *Efficiency*

Efficiency includes two aspects, one is the efficiency of the task execution, and the other is that of the use of computing resources. We choose the actual molecular docking application as the workload *Job_e_1* in this test. The experiment is to use *Job_e_1* to load and analyze 100,000 molecular protein structural files concurrently on 500 computing nodes and 10000 processors. Because of the special configuration of Tianhe-2, when we test Slurm, we first randomly divided 100,000 molecular protein structural files into 500 copies, each of which was 200. Then we use a simple calculation script to perform multiple processes concurrently on each node to execute their corresponding task, until the detection of the molecular protein file is completed.

This experiment divides Teno into three versions for testing: Teno1 performs 20 processes concurrently to execute tasks, Teno2 performs 25 processes, and Teno3 performs 30 processes. They are all performed on HTCTenod. In addition, *Job_e_1* is submitted and executed on HTCondor and Slurm.



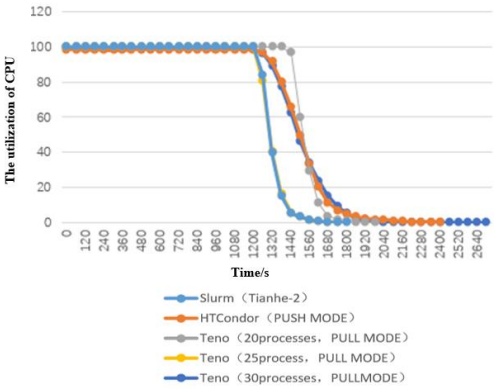


Fig.5 System Efficiency Test of Teno

A total of five rounds of experiments are performed, and 100,000 random samples are taken from the molecular protein structure library in each round. The final experimental results are average of these five rounds, as shown in Fig.5. According to Fig.5(a) and Fig.5 (b), the average completion time of the *Job_e_1* running on Slurm is slightly lower than that of HTCondor under the same environment. That is because on the one hand, the former has no task scheduling process. On the other hand, the execution efficiency of HTCondor drops seriously when the scale of calculation reaches a certain level. However, the calculation efficiency of Slurm is still lower than that of HTCondor. The reason is that when Slurm performs loosely-coupled applications, tasks of each computing node are randomly allocated. Although the scheduling overhead is saved, it will lead to load unbalancing among computing nodes, which will reduce the effective utilization of computing resources. The task execution efficiency and computational efficiency of Teno optimized by PULL model are 11.44% and 33.59% higher than that of Slurm, respectively, and can increase 12.09% and 22.31% compared with HTCondor.

Fig.5(c) describes the use of computing resources by each system when executing *Job_e_1*. We find that the CPU utilization curve of Teno is steeper, while Slurm and HTCondor is more gradual, indicating that before all the tasks start to execute or after all the tasks are completed, both HTCondor and Slurm have some idle computing resources. In contrast, Teno is more efficient in the use of computing resources. The end of the curves show that the effective utilization of computing resources is very low at the end of all operations in all the systems. That is because most of the tasks have been executed and only the remaining tasks are still running. But the design and the setting of Tianhe-2 make these free computing resources unavailable for other users, which is our next focus to optimize.

C. *Scalability*

In order to study further how task execution time and computing resources size impact the performance of Teno, we use efficiency to represent the experimental results in this section of the experiment. The calculation formula for efficiency is:

$$E = T_{\text{best}} / T_{actual} \quad (3)$$

Among them T_{best} represents the best theoretical job execution time, T_{actual} represents the actual job execution time.

The test workloads *Job_s_1*, *Job_s_2*, *Job_s_3*, *Job_s_4*, *Job_s_5*, *Job_s_6* consist of Task1, Task2, Task4, Task8, Task16, and Task32, respectively. Each of workloads contains 100,000 subtasks, and the computational resources are gradually increased from 1200 processors to 12,000. The processor is increased by 1200 at a time.

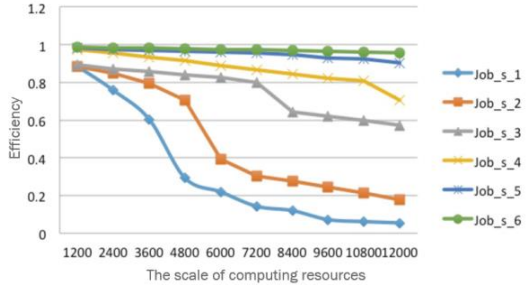


Fig.6 Scalability Test Results of Teno

The experimental results are shown in Fig.6. It can be concluded that the efficiency curve of *Job_s_1* abruptly decreases when the compute resource scale increases to 4800 processors. The efficiency curve of *Job_s_2*, *Job_s_3*, *Job_s_4* get the same results when they are located on 6000 processors, 8400 processors, and 12000 processors, respectively. Before and after this drop point, the efficiency is basically a steady decrease.

Through log analysis and calculation, the main reason of this phenomenon is that when the size of computing resources increases, the number of connections of task queues also increases, which leads to a sharp drop in the response speed of the task queue and the continuous transmission of task information. When the total time of task information transfer becomes larger and larger in the total task processing time, the efficiency becomes smaller and smaller. The phenomenon of drop point indicates that the response speed of the task queue will reach a performance bottleneck while the computing resources are increasing, which will cause a dramatic increase in response time. Moreover, we found that with the increase in the length of the task execution, the "backward" behavior occurs at the location of the drop point.

D. Fault Tolerance Mechanism

We use the molecular docking application and 10,000 molecular structure files as the test workload *Job_r_1* in this section. The computational resource size is 100 nodes and 2000 processors. The purpose of this experiment is to test the correctness and performance of the fault recovery mechanism. So we need to manually simulate the fault. There are four objects in the experiment. They are *Job_r_o_1*, *Job_r_o_2*, *Job_r_o_3*, and *Job_r_o_4*. *Job_r_o_1* simulates the situation that Teno runs *Job_r_1* normally. *Job_r_o_2* simulates the situation that *Job_r_1* is canceled by the user during the execution and then is executed again. *Job_r_o_3* simulates the situation that *Job_r_1* is recalled the compute resources by Slurm during the execution and then is executed again. *Job_r_o_4* simulates the situation that job execution process of *Job_r_1* is forced to be killed during the execution and then the job is continued to execute. During each subject's execution, the "exception" occurred at a random time. The experimental results are shown in Fig.7.

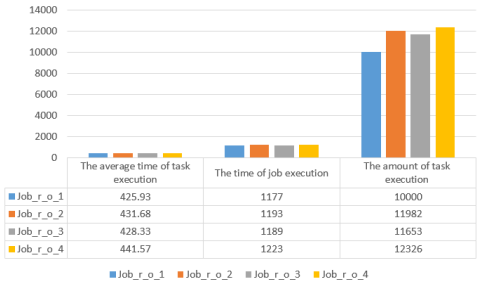


Fig.7: Failure Recovery Mechanism Test

The results show that the fault recovery mechanism of Teno system can quickly and stably resume the fault operation and re-execute it without affecting the task execution efficiency as much as possible, and avoid part of the waste of computing resources accordingly.

We can also see that the average completion time of the tasks in the four execution scenarios are not significantly different, indicating that the failure recovery mechanism does not affect the execution efficiency and the distribution speed of the tasks. But from the perspective of job completion time, the completion time of the failed job is much longer than that of the normal job.

There are two reasons for this phenomenon. On the one hand, the operation of failure job is submitted twice, so the resource allocating and scheduling overhead is also twice that of the normal execution. On the other hand, the second submission job needs to re-execute the task that has not been completed before the failure. That is, the execution flow of the job is partially overlapped. This is also the point we need to study and improve in the future.

Compared with the failure recovery mechanism, the task retry mechanism is more difficult to test. Therefore, we can only design the corresponding test workload *Job_r_2* for the task retry mechanism. *Job_r_2* contains a Python program, which has five kinds of possible operation. They are (1)finishing program and throwing an IOError exception, (2)finishing program and then throwing an Serer exception, (3)finishing program and throwing a MemoryError exception, (4)finishing program and throwing RuntimeError exception, and (5)completing program normally. The five kinds of situations in the program running have an equal probability to occur.

The workload *Job_r_2* has a total of 1 million Python programs, and the experiment is performed on 1200 processors. We update system configuration of Teno to retry the tasks with IOError, OSError, and MemoryError type exceptions, with a maximum of three retries. We also perform the same experiment on Slurm and HTCondor. The results are shown in Fig.8.

From the results, it can be seen intuitively that, the number of successfully executed tasks of *Job_r_1* that are submitted and executed through Teno, is 2.18 times that of Slurm and

HTCondor. The task retry mechanism achieves the desired results.

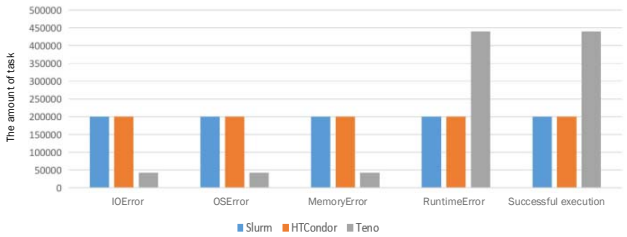


Fig.8 Task Retry Mechanism Test

V. CONCLUSION

We propose a robust and stable high-throughput computing framework Teno. The framework runs on Tianhe-2 and does not need to modify the configuration of the cluster resource manager, Slurm. The hierarchical scheduling subsystem uses Slurm to implement. The high-throughput computing subsystem is researched and improved on the traditional Master-Worker model. The task forest of the execution engine subsystem is organized and constructed to solve the problem that it is difficult to describe large-scale operations. The failure recovery and task retry mechanism of monitoring subsystem ensure the stable and reliable execution of high-throughput calculation operations.

Through different experimental tests, the Teno system's task throughput is nearly 10 times that of HTCondor, the task execution efficiency is 11.44% higher than that of Slurm, and the computational efficiency is increased by 33.59% compared to Slurm, respectively. When the maximum task size reaches tens of millions level, using 12000 processors to execute computing tasks with a running time of 16 seconds, the effective utilization of computing resources can reach over 90%.

In the following work, we plan to optimize the task queue to improve its scalability and response speed, to improve the scheduling strategy so that the effective utilization of computing resources can be improved and optional scheduling algorithms for different types of applications can be provided, and to strengthen the function of the monitoring module to strictly limits the behavior of each task.

ACKNOWLEDGMENT

This work is supported by National Key R&D Program of China 2017YFB0202201, National Natural Science Foundation of China U1611261 and 61433019, the Program for Guangdong Introducing Innovative and Enterpreneurial Teams under Grant NO. 2016ZT06D211.

REFERENCES

[1] Livny M. High Throughput Computing[J].

[2] Tevfik Kosar, George Kola, and Miron Livny. A framework for self-optimising, fault-tolerant, high performance bulk data transfers in a heterogeneous grid environment. In Proceedings of the 2nd International Symposium on Parallel and Distributed Computing, Ljubljana, Slovenia, October 2003.

[3] Taylor S. High Performance Computing of Hydrologic Models Using HTCondor[J]. Brigham Young University, 2013.

[4] Gagliardi F. The EGEE European grid infrastructure project[C]// International Conference on High Performance Computing for Computational Science. Springer, Berlin, Heidelberg, 2004: 194-203.

[5] Liu Yijun, Yang Xiaoyu, Ren Jie1, Wang Zongguo. A Task Management System with High Availability for High-Throughput Material Simulation [J]. E-science Technology & Application, 2015, 6(6): 74-82.

[6] Carter E A. Challenges in modeling materials properties without experimental input[J]. Science, 2008, 321(5890): 800-803.

[7] J Shendure, Hanlee Ji. Next-generation DNA sequencing. Nature Biotechnology 26, 2008, pages 1135-1145.

[8] D. Hanson. “Enhancing Technology Representations within the Stanford Energy Modeling Forum (EMF) Climate Economic Models”, Energy and Economic Policy Models:A Reexamination of Fundamentals, 2006.

[9] PL protein library, http://protlib.uchicago.edu/, 2008.

[10] Liao X, Xiao L, Yang C, et al. MilkyWay-2 supercomputer: system and application[J]. Frontiers of Computer Science, 2014, 8(3): 345-356.

[11] GreenSlot: Scheduling Energy Consumption in Green Datacenters, Inigo Goiri, et. al. SuperComputing, 2011.

[12] S. M. Balle and D. Palermo. Enhancing an Open Source Resource Manager with Multi-Core/Multi-threaded Support. Job Scheduling Strategies for Parallel Processing, 2007.

[13] Miron Livny, Jim Basney, Rajesh Raman, and Todd Tannenbaum, "Mechanisms for High Throughput Computing", SPEEDUP Journal, Vol. 11, No. 1, June 1997.

[14] Elisa Heymann, Miquel A. Senar, Emilio Luque, and Miron Livny, "Adaptive Scheduling for Master-Worker Applications on the Computational Grid". in Proceedings of the First IEEE/ACM International Workshop on Grid Computing (GRID 2000), Bangalore, India, December 17, 2000.

[15] Rajesh Raman, Miron Livny, and Marvin Solomon, "Policy Driven Heterogeneous Resource Co-Allocation with Gangmatching", Proceedings of the Twelfth IEEE International Symposium on High-Performance Distributed Computing, June, 2003.

[16] Deelman E, Vahi K, Rynge M, et al. Pegasus in the cloud: Science automation through workflow technologies[J]. IEEE Internet Computing, 2016, 20(1): 70-76.

[17] Deelman E, Vahi K, Juve G, et al. Pegasus, a workflow management system for science automation[J]. Future Generation Computer Systems, 2015, 46: 17-35.

[18] Anderson E J, Linderoth J. High throughput computing for massive scenario analysis and optimization to minimize cascading blackout risk[J]. IEEE Transactions on Smart Grid, 2017, 8(3): 1427-1435.

[19] https://spark.apache.org/

[20] Zaharia M, Chowdhury M, Franklin M J, et al. Spark: Cluster computing with working sets[J]. HotCloud, 2010, 10(10-10): 95.

[21] White T. Hadoop: The definitive guide[M]. " O'Reilly Media, Inc.", 2012.

[22] Shvachko K, Kuang H, Radia S, et al. The hadoop distributed file system[C]//Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on. IEEE, 2010: 1-10.

[23] Vavilapalli V K, Murthy A C, Douglas C, et al. Apache hadoop yarn: Yet another resource negotiator[C]//Proceedings of the 4th annual Symposium on Cloud Computing. ACM, 2013: 5.

[24] Hindman B, Konwinski A, Zaharia M, et al. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center[C]//NSDI. 2011, 11(2011): 22-22.

[25] Raicu I, Foster I T, Zhao Y. Many-task computing for grids and supercomputers[C]//Many-Task Computing on Grids and Supercomputers, 2008. MTAGS 2008. Workshop on. IEEE, 2008: 1-11.

[26] Katz D S, Armstrong T G, Zhang Z, et al. Many-task computing and blue waters[J]. arXiv preprint arXiv:1202.3943, 2012.

[27] Schlagkamp S, Da Silva R F, Deelman E, et al. Understanding user behavior: from HPC to HTC[J]. Procedia Computer Science, 2016, 80: 2241-2245.

[28] Raicu I, Zhao Y, Dumitrescu C, et al. Falkon: a Fast and Light-weight tasK executiON framework[C]//Proceedings of the 2007 ACM/IEEE conference on Supercomputing. ACM, 2007.