ScalScheduling: A Scalable Scheduling Architecture for MPI-based Interactive Analysis Programs

Jiangling Yin, Andrew Foran, Xuhong Zhang and Jun Wang
EECS, University of Central Florida, Orlando, Florida 32826
jyin, xzhang, jwang@eecs.ucf.edu

***Abstract*—In today's large scale clusters, running tasks with high degrees of parallelism allows interactive data visualization/analysis to complete in seconds. However, conventional, centralized scheduling poses significant challenges for these interactive applications. As the amount of data to be processed grows, it becomes too heavy to move across the network. Thus, data processing tasks should be scheduled such that the amount of transferred data is minimized, i.e., realizing data locality computation. To implement this, a scheduler process should collect and analyze data distribution metadata prior to making scheduling decisions, which usually causes milliseconds or seconds of latency. Such scheduling delay is unacceptable for interactive data applications.**

In this paper, we present a Scalable Scheduling Architecture for conventional interactive data programs and refer to it as ScalScheduling. ScalScheduling is proposed to reduce task scheduling latency, while ensuring the worker processes achieve a high degree of data locality computation and load balance in heterogeneous environments. In our proposed architecture, each worker process uses a novel Modulo-based priority method to schedule its local tasks independently. Multiple scheduler processes are employed according to the number of worker processes to resolve the issue of concurrent requests and assign remote tasks with respect to load balance. We perform experiments using thousands of parallel processes, and the experimental results show the benefits of our proposed scheduling architecture as well as its potential for future oversize task scheduling problems on large-scale clusters.

I. Introduction

With the expansion of information scale, HPC and scientific applications are producing increasingly massive data sets [1].These datasets can be analyzed using parallel distributed computing techniques [2], requiring cooperation among different processes to complete a data processing job. Currently, interactive parallel applications such as Paraview [3] and mpiBLAST [4] are developed with the MPI programming model, where a data processing job is partitioned into many small tasks and parallel processes run as workers to execute the tasks.

When running parallel data applications on commodity clusters, the data input time is a major performance factor. An ideal way to reduce the data input time is to take data distribution into consideration and ensure the data analysis processes achieve data locality computation, such as MapReduce applications [5]. Our previous work VisIO [6] employs a data-centric scheduling algorithm that helps to co-locate parallel MPI processes on nodes with the requested data achieving linear scaling of I/O bandwidth, solving the data movement issue. However, this data-centric method requires the scheduler to collect the metadata of data distribution information, and incurs an extra cost referred to as the *arithmetic overhead*, which is the time cost of finding a suitable task for a worker by checking the metadata. Quincy [7] is also a locality scheduling method and it usually takes over a second to compute a scheduling assignment. Unfortunately, waiting seconds on scheduling is unacceptable for interactive data visualization/analysis, which need to finish in short periods [8], [9].

Additionally, to analyze oversized datasets in heterogeneous environments, data processing tasks usually need to be dynamically assigned to a large number of parallel processes for load balance consideration. This could accumulate a huge arithmetic cost on the scheduler due to the potentially massive amount of concurrent task requests leading to long wait times for the task-requesting processes and resulting in overall performance degradation.

In this paper, we present a scalable scheduling architecture to solve the above issues for interactive data applications based on the message passing programming model and refer to it as ScalScheduling. ScalScheduling is proposed to reduce task scheduling latency, while ensuring the worker processes achieve a high degree of data locality computation and load balance in heterogeneous environments. In our proposed architecture, each worker process uses a novel Modulo-based priority method to schedule its local tasks independently. Multiple scheduler processes are employed to resolve the issue of concurrent task requests from worker processes. The number of scheduler processes depends on the number of worker processes. Moreover, to minimize the number of task request messages, an efficient message exchange protocol is built for task request and approval between worker and scheduler processes.

By load testing the ScalScheduling prototype on PRObE's Marmot 256-core cluster, we found that ScalScheduling improves the overall performance of MPI-based data-intensive applications in comparison to existing scheduling schemes. Unlike the exponentially increasing task latency of traditional monolithic schemes, ScalScheduling achieves a lower task scheduling latency even as the number of parallel processes increases. We also run experiments with different task distributions to show the high degree of data locality computation achieved through our proposed scheduler architecture.

The rest of this paper is organized as follows: Section 2 discusses background and motivations. Section 3 presents our proposed framework. Section 4 shows performance results and analysis. Section 5 discusses related work and Section 6 concludes the paper.

II. Background and Motivations

In large scale clusters, a data processing job is partitioned into a large number of small tasks and executed in parallel by thousands of separate processes. As a result, an efficient method for task scheduling becomes essential, otherwise disorganized tasks assignment can cause significant performance degradation due to network congestion, poor load balance, or task redundancy. In this section, we discuss the scheduling methods for parallel task-based applications.

Dynamic scheduling can usually facilitate a flexible runtime environment, allowing tasks to be assigned to idle processes. As shown in Figure 1, task request messages from idle process are passed to the scheduler and wait in a message queue to be assigned. Currently, most cluster management systems, such as Hadoop, and HPC systems like Torque [10], use some form of dynamic monolithic scheduling. The original Hadoop MapReduce scheduler utilizes a periodic 'heartbeat' and pre-defined delays on the order of seconds when scheduling tasks. Mesos [11] delegates scheduling to framework schedulers and employs batching to handle high throughput, introducing a seconds-order scheduling delay. Condor [12] uses a combination of centralization and distribution for high throughput computing environments. However, these general purpose schedulers are proposed for scheduling batch programs sharing the cluster resources and are not efficient for interactive data anaylsis/visualization.

Fig. 1. The Monolithic Scheduling architecture. Task request messages are passed to the scheduler and enter a message queue, where they wait until they are assigned to an idle process.

For MPI-based data interactive programs, tasks requests/ assignments are more time-sensitive and usually require multiple message exchanges between the scheduler and the worker processes. A single data-centric scheduler could result in poor performance as the scheduling overhead scales with the problem size and number of parallel processes causing high scheduling latency. Methods like mpiBLAST-PIO [4] adopt a hierarchical architecture where a large-scale cluster is organized into several equal-sized partitions and in each partition an independent master is selected as the scheduler. However, data may need to be pulled between partitions over the network because of the wide distribution, which is not ideal for large-scale parallel data-intensive applications.

In this paper, we design a novel scalable scheduling architecture featuring fast task assignment as well as locality aware scheduling to achieve a high degree of local data computation and load balance.

III. ScalScheduling: Design and Implementation

In this section, we first present the software architecture of ScalScheduling. Then, we discuss the design and algorithm behind the architecture.

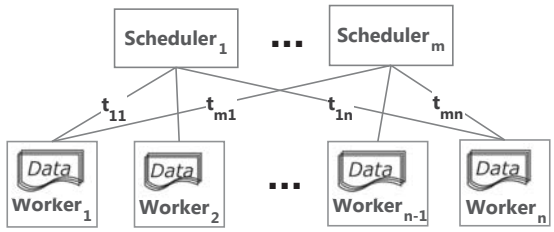

Fig. 2. The ScalScheduling architecture diagram. Each worker process schedules a task for its own based on their local data and sends a task approval request message to the scheduler for task assignment.

A. Software Architecture

In order to minimize program execution time, it is important to let the worker process get a task assignment as soon as possible once it finishes a task. As we discussed, to achieve data locality computation the scheduler process must check the data distribution in order to find the most suitable task for the worker process in a data-centric context. This increases the arithmetic overhead leading to a non-scalable cost for task scheduling. Additionally, a scheduler method which utilizes a single process will amplify the scheduling latency when a large number of working processes ask for task assignments simultaneously.

To address the above issues, we propose a multi-scheduler architecture as depicted in Figure 2. At the top level, several processes are chosen as schedulers, each is supposed to keep track of a partition of the data processing tasks. At the bottom level, each process runs as a worker, executing data processing tasks. In order to reduce the arithmetic overhead of schedulers, each worker process uses a novel Modulo-based priority method to schedule its own local tasks as long as local data exists, exploiting data locality computation. In other words, we essentially amortize the arithmetic overhead over all processes.

Unfortunately, allowing workers to schedule their own local tasks can potentially result in load imbalance due to several heterogeneity issues common on commodity clusters. Firstly, the data to be processed is widely distributed among the cluster nodes. There are always some nodes with more data and local tasks than others. Secondly, the execution time of a data processing task may vary a lot even with similar data sizes [4]. To overcome this issue, we allow scheduler processes to assign remote tasks to worker processes as they run out of local tasks. The multi-scheduler architecture can eliminate the scheduling contention caused by a large number of worker processes asking for task assignments simultaneously.

In order to maximize system utilization, it is also important to choose an appropriate number of schedulers according to the scheduling workload, which in turn depends on the number of worker processes and the request frequency of each worker. Methods like performance profiling on historical executions can be used to find the proper number of schedulers. For instance, in our experiment, we find a good scheduler-to-worker ratio to be approximately 1:512 by comparing the program performance over a gradually varied number of scheduler processes.

B. ScalScheduling Design and Methodology

In this section, we will present the workflow and methodology of task request/assignment in ScalScheduling.

There are two challenges to overcome when ScalScheduling is employed to eliminate the bottleneck of monolithic scheduling.

One big challenge is the necessity of additional scheduling control for avoiding redundant task execution, especially in environments like Hadoop Distributed File System, where each data fragment is replicated and distributed across data nodes and is likely to exist on multiple data nodes. The lack of global task distribution information gives rise to the risk of task execution redundancy in worker processes' self-scheduling of local tasks. To address this issue, we divide the tasks into several subsets based on the task IDs, and we let each scheduler be responsible for one of these subsets. When scheduling a local task, the worker must communicate with the proper scheduler process for permission before actual execution of the task. This ensures the requested task has not been executed by any other processes, eliminating task execution redundancy.

Another challenge is determining a method for distributing the scheduling overhead across the schedulers as evenly as possible, in order to avoid communication contention. Based on the fact that in a large problem, each worker is likely to have local tasks coming from a large range of task IDs, we propose a Modulo-based priority method as detailed in Section III-C. The goal of this method is to allow the local tasks to assume a different position in the execution order of different worker processes, thus distributing the communication messages associated with task scheduling among multiple schedulers at any specific time point.

The interactive workflow between schedulers and workers is shown as follows. From a worker's perspective there are two basic execution modes:

• If there are unprocessed tasks with local data on the worker's hard drive, the worker tries to schedule a local task.

1) The worker process selects a candidate task using the Modulo-based priority method and sends a local task execution request to a proper scheduler.
2) The scheduler replies with an approval message or $NULL$ to the worker.
3) The worker decodes the reply message and either executes the approved task or tries to schedule some different task if the request is not approved.

• If there are no remaining local tasks, the worker enters the remote execution mode.

1) The worker sends a remote task assignment request to a scheduler, which is chosen using a probabilistic decision based on the number of unassigned tasks controlled by each scheduler.
2) The scheduler schedules a task for the worker by considering the system load balance and sends an allocation message to the worker.
3) The worker receives the message and executes a assigned remote task, or sends another remote task request to a different scheduler.

In the workflow, the workers always try to execute a task with local access first. This is very important and useful since the I/O latency can be reduced by avoiding pulling data over the network. On the other hand, it is also necessary to let the schedulers schedule tasks with load balance consideration for remote task requests, since the overall parallel execution is decided by the longest running process.

C. Modulo-based Priority Method

In this section, we will present the methodology of task request/assignment in ScalScheduling.

To eliminate the bottleneck of monolithic scheduling, we use multiple schedulers to keep track of all data processing tasks. Generally, after the initialization of a data processing application, a worker process has many local tasks to choose from. One challenge is to ensure that workers do not simultaneously select the same task even with no awareness of the global task status information. Additionally, the candidate tasks from different workers at a time should be evenly distributed among the scheduler processes in order to get a quick response.

To address the above issues, we let each worker use a Modulo-based method to assign their local tasks priorities such that the same task will have different priorities on different workers. This can reduce the conflict of execution requests for the same task from different workers and distribute the communications overhead over all schedulers. Let n denote the number of workers, f denote the number of tasks, and m denote the number of schedulers. The priority of the z^{th} task on the i^{th} worker is computed using Equation 1, where % represents the modulo operator as in the C Programming Language.

$$x = \lceil \frac{z}{n} \rceil, y = (z-1)\%n + 1, b = \lceil \frac{f}{n} \rceil$$
$$prio(z) = b \times ((y+n-i)\%n) + (x+b-i\%m)\%b \quad (1)$$

We show an example in Figure 3 to see how the Modulo-based method works. We simulated an HDFS dataset of around 600MB (about 10 fragments), each with 3 copies. We assume each chunk is related to a data processing task. Figure 3(a) shows the tasks with local data related to each worker process. Since each chunk has 3 copies, a task may be local to 3 processes. Without using the Modulo-based method, Workers 1, 3, and 5 may try to schedule task t_1 (task 1) but only one of them can succeed. On the other hand, (b) shows the tasks sorted using the Modulo-based method, where the priority increases from left to right and two schedulers s_1 and s_2, each control 6 tasks. Considering the two rightmost tasks on each worker's priority queue, we can see that the only possible conflict is between worker 4 and 5 on task 11.

D. Maximizing the Efficiency of Communications

In this section, we firstly discuss how to ensure high permission rate for local task execution requests. Then we present the detailed message exchanges in the ScalScheduling architecture.

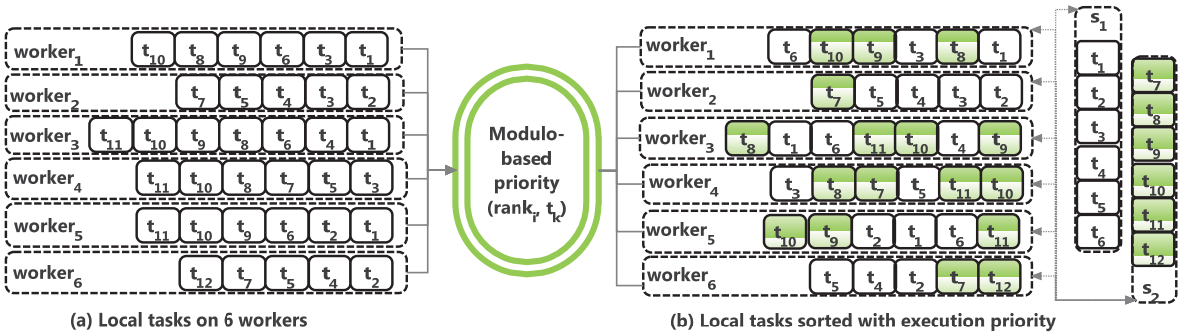

Fig. 3. A simple example of local tasks sorted with the Modulo-based method. (a) is the tasks with local data related to each worker process; (b) the tasks sorted using the modulo-based method, where the priority increases from left to right; s_1 and s_2 are the two schedulers each controlling 6 tasks.

1) Communication Cost: In our proposed architecture, worker processes schedule local tasks without knowing global task distribution information. To avoid task execution redundancy, a worker process needs to get execution permission before performing a local task. However, if a worker selects only one task in each request sent to the scheduler there is a relatively high chance that the selected task has already been executed by some other worker process. This will decrease parallel execution performance since multiple communications may be necessary for one task assignment.

A naive solution to address the above issue, is to simply send all the unprocessed local tasks on a worker to a scheduler as a task request. Among these candidate tasks, the scheduler identifies one that has not been executed and sends the execution permission to the worker. Thus the worker can get local task execution permission using only one communication as long as there exist local tasks not yet executed. Unfortunately, this method is not efficient, the message sent to the scheduler is unnecessarily large. Additionally, requiring a scheduler to look through the possibly long list of unprocessed tasks defeats the purpose of amortizing the arithmetic overhead.

To improve the chance of obtaining a local task assignment as well as keeping the request message short, we let each worker process select τ local tasks as candidates in each request. We assume that the total number of data processing tasks is f and the total number of remaining un-executed tasks at a given time is g. The probability that all τ candidate tasks are executed by others is approximately $(\frac{f-g}{f})^{\tau}$. Thus, the chance to get a local task execution permission from τ candidate tasks is

$$P_{\tau} = 1 - (\frac{f-g}{f})^{\tau}. \quad (2)$$

This function shows that the chance of a worker getting a permission increases exponentially as τ grows.

Furthermore, the actual probability of getting a local task execution permission is even larger than the value given by (2) in our design since we use a *kept-list* mechanism. Each time the scheduler approves one task to a worker process, the unapproved tasks in the request message are put into a kept-list associated with that worker if they are not yet assigned to any other processes. Thus, a worker can still get a local task assignment from the kept-list even if all the τ candidate tasks are executed. In fact, according to our experiments, the chance of a worker getting a local task execution permission is more than 95% using the kept-list mechanism with $\tau = 2$.

TABLE I. THE MESSAGES IF LOCAL TASKS EXISTS.

The messages sent to scheduler			
Task1		**Task 2**	**Message meaning**
ID		ID	Two candidate tasks sent
ID		$NULL$	One candidate task sent
$NULL$		$NULL$	Request a Local task
The messages received from scheduler			
Tag	**Task1 status**	**Task 2 status**	**Message meaning**
ID	—	—	Task(ID) Assigned
α	Approval	Kept by scheduler	Task1 Assigned
β	Approval	Completed	Task1 Assigned
γ	Completed	Approval	Task2 Assigned
χ	Completed	Completed	No local tasks kept by scheduler

2) ScalScheduling Messages: We now present the detailed message for $\tau = 2$, that is, a worker sends two local tasks as candidates to a scheduler in each local task execution request. The messages sent to a scheduler are described in Table I. To avoid repeatedly sending a task to a scheduler, each task can be sent only once by a worker process. The scheduler can approve at most one task for the worker even if neither of the candidate tasks have been executed. The unapproved task will be stored in a kept-list assosciated with the requesting worker. If the workers cannot find 2 unprocessed local tasks controlled by a single scheduler, a worker process can send a single local task ID together with a $NULL$ ID to the scheduler. Furthermore, if no local task that has not been sent to a scheduler can be found on a worker process, the worker process sends two $NULL$ IDs to a scheduler process, which can respond with an unfinished local task maintained in the kept-list.

There are five kinds of messages the scheduler may reply to the worker with, as shown in Table I. When the kept list is not empty, the scheduler will find an unprocessed task from the kept-list associated to the worker and send its task ID to the worker. If the kept list is empty and neither of the candidates have been executed, the scheduler will approve the first one for the worker and keep the second one in the kept-list, responding with message tag α. If the kept list is empty and both candidates are executed, the scheduler will reply with a $NULL$ message to the worker with tag χ. The tags β and γ refer to the approval of the first or second task to the worker, and signify that the other one has been executed.

Once a worker process runs out of local tasks, it will enter the remote execution mode. A remote task request will be sent

TABLE II. THE MESSAGES IF NO LOCAL TASKS EXISTS.

The messages sent to scheduler	
Tag	**Message meaning**
ν	Remote Task Request
The messages received from scheduler	
Tag	**Message meaning**
ID	Assign Remote Task(ID)
$NULL$	All Tasks Completed

to a scheduler chosen with criteria detailed in the following subsection, and the scheduler will assign a task to the worker by considering system load balance. As listed in Table II, the messages sent during the remote execution state from workers have tag ν, and the message received from a scheduler is either a task ID, or $NULL$ if all tasks under its control are assigned.

E. ScalScheduling Algorithms

In this section, we will present the worker and scheduler algorithms.

1) Worker Algorithm: The worker algorithm is presented in Algorithm 1. T_i is the set of local tasks on Worker Process i. The worker firstly sorts its local tasks with the Modulo-based method. While $|T_i| \neq 0$, the worker sends a message to a scheduler according to Table I. In the remote execution mode, to get a remote task assignment the worker process needs to decide a proper scheduler to send the task request to. This is an important step when executing in a large scale cluster since improper decisions may lead to a large number of worker processes overloading a single scheduler by simultaneously sending remote task requests.

In our proposed architecture, we let the scheduler send the number of uncompleted tasks under its control when the two candidate tasks are $NULL$ in a local task request or in reply to a remote task request. Since a worker knows the number of tasks that have not been completed on each scheduler, it can use this information to make a probabilistic decision to choose a scheduler when sending a task request. Assume there are m schedulers and the number of unfinished tasks on scheduler k is n_k, the probability of a worker sending a remote task request to scheduler k is $P_k = \frac{n_k}{\sum_{0 \leq q \leq m} n_q}$. In this way, task requests are distributed evenly among all schedulers.

2) Scheduler Algorithm: With the goal of decreasing the scheduling latency, ScalScheduling shifts a lot of the workload to the workers, thus the scheduler algorithm is actually very simple; the algorithm is presented in Algorithm 2. We have already described the handling of local task execution request in Section III-D2. It should be noted that when both candidate tasks are $NULL$ in a local task request or a remote task is requested, the reply message of the scheduler will also contain n_k, the number of unfinished tasks controlled by that scheduler.

If a remote task request is received from Worker Process i, to achieve load balance across the application, the scheduler will randomly choose a task from the longest kept-list K_{kx} and assign it to the requesting worker process.

Algorithm 1 Worker Algorithm on Worker i

Let m be the number of scheduler processes;
Let U_k be the set of unprocessed tasks controlled by Scheduler k;
Let T_i be the set of unprocessed local tasks on Worker Process i;
Let $n_k = |U_k|$, $T_{ki} = U_k \bigcap T_i$;
Steps:
//Modulo-based priority
for Task $z \in T_i$ **do**
  $x = \lceil \frac{z}{n} \rceil, y = (z-1)\%n + 1, b = \lceil \frac{f}{n} \rceil$;
  $prio(z) = b \times ((y+n-i)\%n) + (x+b-i\%m)\%b$;
end for
while $|T_i| \neq 0$ **do**
  $ID1 = NULL$, $ID2 = NULL$;
  Find a Task z with maximum $prio(z)$ in T_i
  Let k be the scheduler controlling z
  if $|T_{ki}| = 1$ **then**
    Assign Task z to $ID1$;
  else
    //$|T_{ki}| > 1$
    Assign two tasks with maximum priority in T_{ki} to $ID1$ and $ID2$;
  end if
  Send task request with $ID1$ and $ID2$ to Scheduler k;
  Receive message from scheduler k;
  Decode message, Update T_i, T_{ki} and n_k;
  Task_execution();//developer implementation
end while
//Probability-base selection
while $\sum_{0 \leq q \leq m} n_q \neq 0$ **do**
  Let $P_k = \frac{n_k}{\sum_{0 \leq q \leq m}(n_q)}$ for all k;
  Send a remote task request to Scheduler x with probability P_x;
  Receive message from Scheduler x;
  Decode message, update n_x;
  Task_execution(); //developer implementation
end while

IV. EXPERIMENTS

A. Experimental Setup

We conducted comprehensive testing for our proposed ScalScheduling on Marmot. *Marmot* is a cluster of the PRObE on-site project [13] and housed at CMU in Pittsburgh. The system has 128 nodes / 256 cores and each node in the cluster has dual 1.6GHz AMD Opteron processors, 16GB of memory, Gigabit Ethernet, and a 2TB Western Digital SATA disk drive.

In the cluster, MPICH [1.4.1] is installed as the parallel programming framework on CENTOS55-64 with kernel 2.6. To study the properties of data distribution on multi-way replication storage systems, we installed Hadoop 0.20.203 as the distributed file system, which is configured as follows: one node for the NameNode/JobTracker, one node for the secondary NameNode, and other nodes as the DataNode/TaskTracker.

B. Evaluating ScalScheduling

In this section, we firstly describe how real workloads could use our proposed scheduler, then we will specially build an experimental environment which focuses on testing the performance of scheduling methods.

To process a dataset, parallel applications such as mpiBLAST [4] or ParaView [3] will first read a meta-file, which points to a serial of data files, to initialize the list of tasks. Then

Algorithm 2 Scheduler Algorithm on Scheduler k

Let U_k be the set of unprocessed data tasks controlled by Scheduler Process k, $n_k = |U_k|$;
Let K_{ki} be the set of unprocessed data tasks kept by Scheduler k and received from Worker Process i (called as kept-list in the former discussion);
Steps:
if A local task request is received from Worker i **then**
  if $|K_{ki}| \neq 0$ **then**
    Encode the reply message with a task from K_{ki};
  else
    Encode the reply message with α, β, γ or χ according to Table I;
  end if
  if $(Task1 = NULL \ \& \ Task2 = NULL) \ \& \ |K_{ki}| = 0$ **then**
    Encode message with n_k;
  end if
  Reply message to Worker i;
  update U_k and K_{ki};
end if
if A remote task request received from Worker i **then**
  Find x such that $|K_{kx}| = \max_{1 \leq q \leq n} |K_{kq}|$;
  Get a task from K_{kx};
  Encode message with the task and n_k;
  Reply message to Worker i;
  update U_k and K_{ki};
end if

the tasks will be assigned to the MPI processes for parallel execution. It's easy to incorporate our proposed scheduler into these type of parallel applications. Instead initializing the tasks on the scheduler, we let all the processes read the metadata file and find their local tasks. Then they can request tasks assignment permission using our proposed scheduling.

In order to make an effective comparison, we developed a benchmark application which focuses on testing the performance of task scheduling. First, we implement a monolithic scheduling architecture as described in Section 2 for comparison, referred to as "MonolithicScheduling". Specifically, a single MPI process as the scheduler calls a locality function to build the locality information class like that in mpiBLAST [4]. We refer to the locality information as metadata. The scheduler uses the metadata to dynamically assign tasks to worker processes. We use the locality algorithm in our previous work [6], [14] to find a proper task for a request. For our ScalScheduling benchmark, we adopt a flexible number of schedulers, where an extra scheduler is added for each additional 512 worker processes. Moreover, the worker rather than the scheduler calls the locality function to find the local tasks for itself. We use the locality library described in our previous work [15].

In our benchmark program, we test different random data distribution schemes on Hadoop Distributed File System with a chunk size of 64 MB and 3 replicas. Since we are more concerned with the scheduling and communication performance of ScalScheduling, in our test programs each MPI worker process is set to have a random task processing time at second scale. The total number of tasks to be processed is 10 times the number of worker processes. The task processing

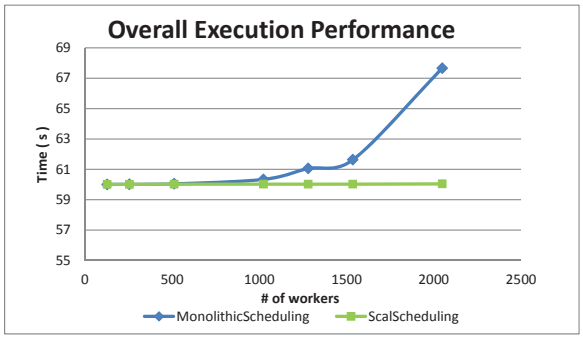

Fig. 4. The overall execution comparison of ScalScheduling to a Monolithic Scheduling scheme. The key observation is that ScaleScheduling scales with increasing number of worker processes. By contrast, the execution time of Monolithic Scheduling increases quickly after the number of worker processes reaches 1536.

time was chosen as a reasonably realistic time frame for the completion of a task based on experience with data processing applications [8], [9], [4], [7], [16]. Throughout the execution of our benchmark application, we randomly generated message exchanges between MPI processes to simulate a realistic data processing MPI appplication. Each worker is allowed to randomly send and receive messages besides messages associated with task assignment.

1) Scalable Performance: First, we present the processing time comparison of our benchmark programs using ScalScheduling and MonolithicScheduling on Marmot for a variable number of processes, shown in Figure 4. From the figure, we find that the benchmark program using ScalScheduling obtains similar processing times as the one using MonolithicScheduling when the number of worker processes is smaller than 512. However, with an increasing number of worker processes, the processing time increases quickly for the benchmark program using MonolithicScheduling. This could be explained by the effect of the arithmetic overhead and communication contention happening on the single scheduler, which causes some worker processes to wait longer for task allocation. In comparison to MonolithicScheduling, ScalScheduling uses 2, 3, and 4 schedulers with respect to the number of worker processes being 1024, 1536 and 2048. With the arithmetic overhead shifted to the worker processes, and the communication and scheduling cost being shared by multiple schedulers, the overhead is eliminated and the processing time is much smaller.

To gain some insights on how expensive scheduling overhead is, we illustrate *task request latency* in Figure 5. A task request latency is the total time span between a task request and an approval. The metric depicted in the figure is the average delay time among all worker processes. We find that the total delay time of ScalScheduling reaches around 20ms before the metadata sizes rise to 20Kb and increases slightly to 50ms for metadta around 40Kb. As the size of metadata increases to more than 80 Kb, the delay time of MonolithicScheduling grows exponentially while that of ScalScheduling still stays around 50 ms. This shows that ScalScheduling can achieve scalability for data-intensive applications.

2) Scheduler Efficiency Tests: For data-intensive applications, the centralized scheduler usually schedules a task to

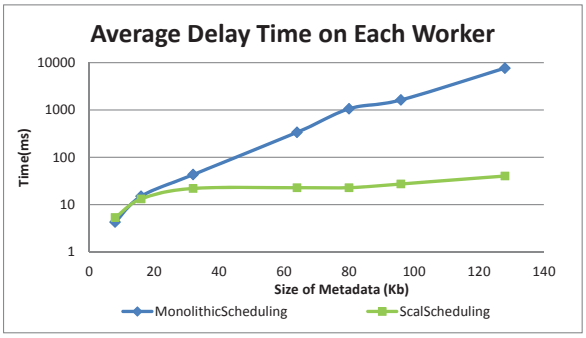

Fig. 5. Average delay time comparison between ScalScheduling and a Monolithic Scheduling architecture. We see much shorter delay times for ScalScheduling than Monolithic Scheduling when the size of data processing tasks are more than 300GB.

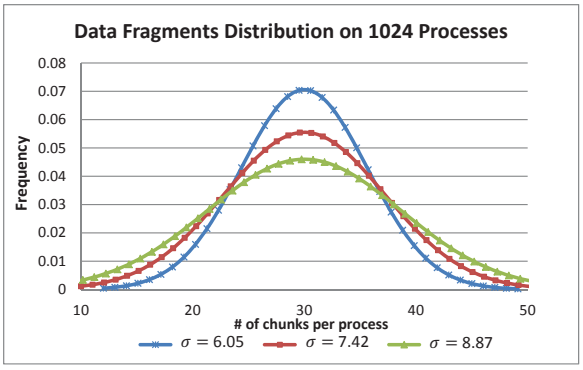

Fig. 6. Three kinds of data distribution on 1024 processes with $\delta = 6.05, 7.42$ and 8.82.

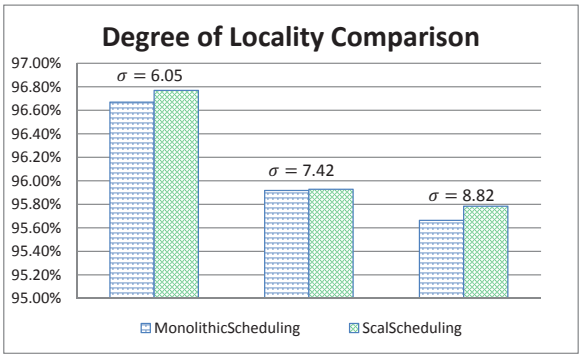

Fig. 7. Degree of data locality comparison on three kinds of data distribution on 1024 processes with $\delta = 6.05, 7.42$ and 8.82.

a worker process by looking at the data on that process for locality computation. However, in our proposed scheduling architecture, we allow each worker process to schedule local tasks for themselves instead of using central control. Thus, to analyze the efficiency of our scheduling on data locality computation, we simulate several data distribution schemes as shown in Figure 6 to test the effect on the degree of data locality achieved. There are three cases with standard deviation values of $\delta = 6.05, 7.42$ and 8.82 and the average number of data fragments local to each process is 30. For data distribution, a smaller δ implies the data is more evenly distributed.

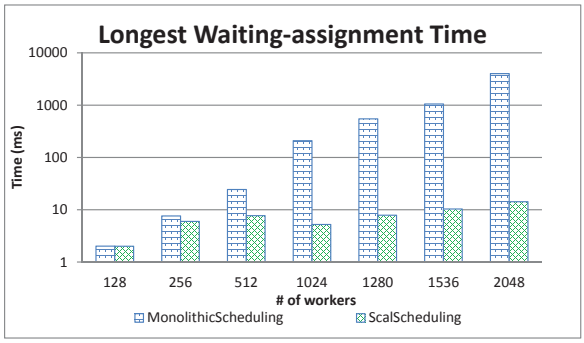

Fig. 8. The longest waiting-assignment time comparison of MonolithicScheduling and ScalScheduling with varying number of processes.

The priority of our ScalScheduling architecture is to achieve data locality. To explore how effectively ScalScheduling works (i.e., to what extent parallel processes are scheduled to access local data), we ran multiple experiments for each case to check how much data was processed locally with 1024 processes and 30,720 data fragments. Figure 7 illustrates the comparison of data locality computation between MonolithicScheduling and ScalScheduling. ScalScheduling achieved slightly higher degrees of data locality computation in all three cases. As seen from the Figure 7, in all three cases, ScalScheduling has more than 95.6% of data fragments read from local hard drive.

We also measure the longest waiting time for a task assignment as shown in Figure 8. In general, as the longest wait time increases, so does the parallel execution time. From the figure, we find that for MonolithicScheduling the longest waiting-assignment time greatly grows with the increasing number of processes. This also could explain why monolithic scheduling achieves worse overall performance.

V. Related Works

Static scheduling reduces communication/scheduling overhead by scheduling all tasks initially, eliminating the need to track task completion and schedule tasks dynamically. Kwok *et. al.* [17] presents a review of classic static algorithms for multiprocessor systems. Unfortunately, scheduling all tasks initially makes for a very inflexible runtime environment [16], as execution time can be unpredictable and lead to load imbalance, especially on heterogeneous clusters.

Dynamic scheduling algorithms are usually adopted to achieve load balance for minimizing parallel execution time. To reduce the communication/scheduling cost, Self-scheduling algorithms [18], [19], [20] divide the total number of tasks into chunks, which are then assigned to workers. However, these algorithms are not applicable for data intensive applications since the data processing tasks related to data can not be arbitrarily divided into chunks. Otherwise, more data may potentially need to be transferred to worker processes through the network, which will cause network congestion.

With resource allocation for parallel processes in heterogeneous environments, hierarchical methods are usually proposed to improve the system efficiency and utilization.

Chronopoulos *et. al.* [21] employs a hierarchical Master-Slave architecture to deal with scheduling and load balancing of a system that consists of heterogeneous computers. Varisteas *et. al.* [22] proposed a space-sharing, two-level, adaptive scheduler to solve the problem of resource contention, created by multiple parallel applications running simultaneously. mpiBLAST-PIO [4] adopts a hierarchical architecture, where a large-scale cluster is organized into several equal-sized partitions and an independent master is selected in each partition. However, these methods will run into network contention issues without considering data locality. Quincy [7] could increase the throughput up to 40% through reducing the volume of data transferred across the cluster. Delay scheduling for MapReduce [23] showed that enabling a higher degree of data locality computation can largely increase the throughput and overall performance. Different from these methods, ScalScheduling is proposed to reduce the scheduling latency for parallel interactive application and also consider the data locality during scheduling.

VI. Conclusion

In this paper, we propose a scalable scheduling architecture to support task request/assignment for large number of worker processes running in parallel data interactive applications, allowing them to achieve lower task scheduling latency. To reduce the arithmetic cost associated with checking the metadata of locality information related to worker processes, we let each worker process to use a Modulo-based method to decide their own local task execution order to exploit data locality computation. By scaling the number of schedulers according to the number of worker processes, the cost of task request/assignment is further reduced on each scheduler. We proposed a new message exchange protocol for task request/assignment, which could improve the efficiency of communication and avoid redundant task assignment. By testing our ScalScheduling prototype system on a 256 core cluster, we saw significant performance improvement over monolithic scheduling architectures. These improvement will be enhanced even further by increasing the cluster size.

VII. Acknowledgments

This work is supported in part by the US National Science Foundation Grant CNS-1115665, CCF-1337244 and National Science Foundation Early Career Award 0953946.

This work is conducted at a PRObE staging cluster-128-node Marmot cluster, which is supported in part by the National Science Foundation under awards CNS-1042537 and CNS-1042543 (PRObE).

References

[1] "Genomes to life project proposal," www.genomes2life.org/SNL-ORNL-GTL-Proposal.doc.

[2] C. Wu and A. Kalyanaraman, "An efficient parallel approach for identifying protein families in large-scale metagenomic data sets," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, ser. SC '08. Piscataway, NJ, USA: IEEE Press, 2008, pp. 35:1–35:10.

[3] J. Ahrens, B. Geveci, and C. Law, "Paraview: An end-user tool for large data visualization," *The Visualization Handbook*, vol. 717, p. 731, 2005.

[4] H. Lin, X. Ma, W. Feng, and N. F. Samatova, "Coordinating computation and i/o in massively parallel sequence search," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 4, pp. 529–543, Apr. 2011.

[5] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[6] C. Mitchell, J. Ahrens, and J. Wang, "Visio: Enabling interactive visualization of ultra-scale, time series data via high-bandwidth distributed i/o systems," in *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, May, pp. 68–79.

[7] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: fair scheduling for distributed computing clusters," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 261–276.

[8] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 2–2.

[9] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis, "Dremel: interactive analysis of web-scale datasets," *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 330–339, Sep. 2010.

[10] G. Staples, "Torque resource manager," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, ser. SC '06. New York, NY, USA: ACM, 2006.

[11] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: a platform for fine-grained resource sharing in the data center," in *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, ser. NSDI'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 22–22.

[12] D. Thain, T. Tannenbaum, and M. Livny, "Distributed computing in practice: the condor experience: Research articles," *Concurr. Comput. : Pract. Exper.*, vol. 17, no. 2-4, pp. 323–356, Feb. 2005.

[13] G. Gibson, G. Grider, A. Jacobson, and W. Lloyd, "Probe: A thousand-node experimental cluster for computer systems research," vol. 38, no. 3, June 2013.

[14] S. Sehrish, G. Mackey, P. Shang, J. Wang, and J. Bent, "Supporting hpc analytics applications with access patterns using data restructuring and data-centric scheduling techniques in mapreduce," *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no. 1, pp. 158–169, Jan. 2013.

[15] J. Yin, A. Foran, and J. Wang, "Dl-mpi: Enabling data locality computation for mpi-based data-intensive applications," in *Big Data, 2013 IEEE International Conference on*, Oct 2013, pp. 506–511.

[16] I. Riakotakis, F. Ciorba, T. Andronikos, and G. PapaKonstantinou, "Self-adapting scheduling for tasks with dependencies in stochastic environments," in *Cluster Computing, 2006 IEEE International Conference on*, 2006, pp. 1–8.

[17] Y.-K. Kwok and I. Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," *ACM Comput. Surv.*, vol. 31, no. 4, pp. 406–471, Dec. 1999.

[18] S. F. Hummel, E. Schonberg, and L. E. Flynn, "Factoring: a method for scheduling parallel loops," *Commun. ACM*, vol. 35, no. 8, pp. 90–101, Aug. 1992.

[19] T. Tzen, T. H. Tzen, L. M. Ni, and L. M. Ni, "Trapezoid self-scheduling: A practical scheduling scheme for parallel compilers," 1993.

[20] I. Banicescu, V. Velusamy, and J. Devaprasad, "On the scalability of dynamic scheduling scientific applications with adaptive weighted factoring," *Cluster Computing*, vol. 6, no. 3, pp. 215–226, 2003.

[21] A. T. Chronopoulos, S. Penmatsa, N. Yu, and Y. Du, "Scalable loop self-scheduling schemes for heterogeneous clusters," *IJCSE*, vol. 1, no. 2/3/4, 2005.

[22] G. Varisteas, M. Brorsson, and K.-F. Faxen, "Resource management for task-based parallel programs over a multi-kernel. : Bias: Barrelfish inter-core adaptive scheduling," in *Proceedings of the 2012 workshop on Runtime Environments, Systems, Layering and Virtualized Environments (RESoLVE12)*, 2012, pp. 32–36, qC 20130116.

[23] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling," in *Proceedings of the 5th European conference on Computer systems*, ser. EuroSys '10. New York, NY, USA: ACM, 2010, pp. 265–278.