Bridging the Gaps between Many-task Computing and Supercomputers

ZHAO ZHANG

July 11, 2011

Contents

Abstract

Many Task Computing, an emerging programming paradigm on supercomputers, embraces many applications in such domains as biology, economics, and statistics, as well as data intensive computations and uncertainty quantification. Its high inter-task parallelism and intense data processing features place new challenges on the existing hardware-software stack on supercomputers. Those new challenges include resource provisioning, job scheduling, load balancing, data management, and resiliency. In this paper, we identify Many-Task Computing middleware gaps between the applications and supercomputer's hardware-software stack by examining their characteristics. Based on this analysis, we propose AME, an Anyscale MTC Engine with a special focus on scalability. We describe the AME framework and present performance results for both synthetic benchmarks and real applications. Our results show that AME is a highly scalable MTC engine on petascale machines, and a strong candidate for exascale machines and beyond.

Chapter 1

Introduction

Many-Task applications [20] involve the automatic execution of a number of other programs, where job dependencies are resolved by POSIX-compatible files. Such applications usually feature high inter-task parallelism, and invoke more than one existing program. The file flow patterns vary among applications and can be depicted by file flow graphs. These patterns have been well studied [19, 18].

Many current supercomputers have some common features. From a hardware perspective, there are a large number of multi-core compute nodes that have ram disks but no local disks, one or several low-latency networks, and a shared file system. At the operating system level, the compute nodes might or might not have a full Linux kernel and have POSIX-compatible access to the shared file system. As supercomputers are administrated by various institutions, they have different scheduling policies. The granularity of scheduling, which can be interpreted as the smallest number of compute nodes that users can request, differs among machines. A specific example is the IBM BG/P at Argonne, which has 40960 quad-core PPC 450 compute nodes, each with a shared 2 GB memory. The BG/P has 5 networks: one low-latency high-bandwidth 3D torus network for message passing, one highly scalable collective tree network for data collection, one global interrupt network for low latency barrier and interupts, one control network that could access the RAM of all compute nodes and IO nodes, and an Ethernet network that interconnects IO servers, IO nodes, and login nodes. The ANL BG/P has a GPFS configuration of 128 file servers yielding 3072 TB of storage. The scheduling granularity is 64 compute nodes, because of the hardware layout. (The prior BG/L system had a granularity of 32 nodes.)

Naively applying Many-Task application to the existing supercomputer hardware-software stack will result a series of problems, such as low machine utilization, low scalability, overload of file system, and more. Examining

the characteristics of Many-Task application and the physical layout of the supercomputers, we identify the following gaps from a distributed system perspective:

- Resource Provisioning: A first gap lies between the static resource provisioning scheme and the variable running time of the jobs. Well known job schedulers such as PBS features a static scheduling scheme, where it is not feasible to release part of computing resources while the job is not complete yet. A second gap is between the scheduling granularity and the potential/eventual lack of jobs ready to be run. IBM BG/P and BG/L have a scheduling granularity at 64 compute nodes and 32 compute nodes respectively. As we have seen, at some stage of the application, the number of jobs will be far less than the scheduling granularity, leading to low utilization at that stage.

- Job Dispatching: Most of the existing supercomputer schedulers suffer overhead at the minute level when starting/terminating allocations. In the case of jobs that run for seconds, this dispatching overhead will dominate the time needed to run the application.

- Load Balancing: The existing supercomputer hardware-software stack does not provide load balancing functionality for the Many-Task Computing paradigm, while such applications requires this feature in order to obtain high machine utilization during the execution.

- Data management: The gap in this category is a key gap between Many-Task applications and supercomputer hardware. The I/O system, including the shared file system, is not capable of handling the large number of metadata operation requests from a Many-Task application.

- Resiliency: The lacking of resiliency mechanism on existing parallel programming language, such as MPI, also places a challenge for Many-Task applications. Various failures may occur when a Many-Task application is running. Hardware failures and operating system failures are not recoverable at the Many-Task Computing engine level. When those failures happen, the obstacles to recovery are identifying finished jobs, inferring the job dependencies of the unreturned and failed jobs, and re-establishing the states of various services of the runtime system.

In some cases, Job Dispaching and Load Balancing are interlevead in one schduler. A centralized job dispatcher that sends the longest job to the next

available compute node also balances the load among the compute nodes in the mean time of dispatching. In the following discussion, the term dispatcher describes the the schdulers' role in the job dispatching scenario, while the term load balancer my refer to the schdulers' load balancing role or some independent load balancing service. In this paper, we address three categories of the above gaps: Job Scheduling, Load Balancing, and Data Management.

- To address scheduling gaps, we take advantage of previous lessons from Falkon [13] regarding multi-level scheduling. In addition, we evaluate the idea of centralized dispatching vs. decentralized dispatching.

- To address the load balancing issue, we have two design alternatives. One of them is a centralized dispatcher with a pull model, the other is a decentralized dispatcher with a push model plus work-stealing.

- For the data management gap, we have classified data according to their usage pattern in [20]. They are common input, unique input, output and intermediate data. In this paper, we focus on the intermediate data handling scheme. We present a distributed memory coherence protocol that resolves job dependencies at runtime. Along with a distributed hash table (DHT) based design, this particular management scheme features high scalability. We also demonstrate how to use this scheme as a primitive to benefit the file flow patterns of Many-Task applications.

The rest of the paper is organized as following: Part II discusses both previous works in this domain and related works with intriguing ideas from other domains. In Part III, we demonstrate the high level design of AME and the communication among modules of the system. We show both the benchmark design and performance results with explanation in Part IV. In Part V, two extreme cases of file flow patterns of Many-Task applications are examined. Conclusion are drawn in Part VI, and future work is envisioned in Part VII.

Chapter 2

Related Work

2.1 Job Dispatching

Regardless of the programming paradigm, the program source code needs to be translated to some machine code that could be executed workers(or CPU). We look around on how well known programming language or library dipatch jobs to workers in the Parallel and Distributed Programming context.

MPI leaves this function to programmers. It is a common case that, for embarassingly parallel applications, MPI programs include the code for all the tasks that may be run, and each compute node does its part of the work which is identified by the worker's rank. This scheme is the most scalable of the ones we covered, but it requires the compute nodes to load redundant information. Every compute node needs to load the compiled binary, and finds out its own task. Pegasus/Condor [8] uses a centralized job dispatcher, the submit host, which keeps a shadow of every single job. It tracks lifetime state change of the jobs. Thus its scalability is limited to the capacity of the submit host. Also, it consumes a lot more memory than the MPI case. Falkon [13] explores a 3-tier architecture: a first tier submit host, a group of second tier dispatchers, and a group of third tier workers. Nevertheless, Falkon's job view is the same as Condor, and limits the scalability of short jobs (O(1) s). AME's dispatcher takes advantage of Falkon's 3-tier design, and changes the job view from a single job to a text file at the first layer. AME's dispatcher does not monitor job status, which results in higher scalability than Falkon; it is a tradeoff between scalability and the detailed job status monitoring.

2.2 Load Balancing

For those embarassingly parallel applications, the running time of jobs may vary. Load Balancing is necessary for the run to achieve optimal utilization.

Supercomputers do support for High-Throughput Computing applications, such as IBM BG series' HTC mode and Cray XE6's Cluster-Compatibility Mode. Thus supercomputers provide load balancing for HTC computation. But MTC applications' huge amount of jobs could go beyond the load balancing capability(memory, CPU) of the scheulers and load balancers of existing supercomputers. The MPI standard does not provide such functionality; ADLB [12] (and other higher level schemes) aims to solve this bottleneck in the MPI scenario. In the parallel programming languages Cilk, and Parlog [6], researchers have put effort in the work-stealing algorithm that tries to remedy the starving situation that may occur. But none of the existing work-stealing algorithms has ever been proved to scale up to 100,000 cores. AME wants to address this problem via a local stealing scheme, where each compute node cannot access every other compute node in the allocation. This is because some of the supercomputers' communication networks are in torus topology(IBM BlueGene Series, Cray XE6), which features a hop-by-hop transmission network, so by limiting the stealing scope, we can avoid high-latency round-trips. Also, this can balance the stealing workload across all the compute nodes in the allocation.

2.3 Data Management

Data Management is a key component of many parallel and distributed computing programming systems.

Related work on data management ranges from the operating system to distributed computing middleware: ZOID [9] works with the computer node OS kernel to accelerate the IO throughput from computing resources to persistent storage. GPFS [15], LUSTRE [7]and PVFS [5] aim to provide scalable global persistent storage for supercomputers. GridFTP [2], MosaStore [1], and Chirp [16] provide data management primitives on grids and clusters at workflow runtime. ROMIO [17] is designed as the I/O support for MPI, which can also be viewed as the Data Management Module. MPI also leaves this feature to programmers. In most MapReduce scenarios, the data to be processed are assumed to reside on the compute nodes. HDFS [4] (Hadoop Distributed File System) places three replicas of each data chunk over the compute nodes. Other work tries to isolate the data storage and processing. HDFS's scalability is mainly limited by its single management node archi-

tecture. Results show it scales up to the level of 1,000 compute nodes, but not more. AME's data management system is designed as a MTC runtime support on supercomputers. It differs from the persistent storage by the lifetime of the data it manages. With the DHT based design, theoretically, it could scale up to any number of compute nodes.

2.4 System Resilience

System Resilience is considered quite differently among different parallel and distributed computing systems.

MPI does not provide any resilience features: when an MPI program fails, the user needs to restart the run. Condor uses a checkpointing scheme for resilience. MapReduce treats system failure as a norm rather than an occasional accident, thus it duplicates jobs as it takes node failure as a normal situation rather than an exception in the commodity clusters. We are still working on how AME should address this issue.

Chapter 3

Design

3.1 Overview

The AME system tackles three categories of gaps: job scheduling, load balancing, and data management. It consists of five modules: a provisioner, a submitter, a group of decentralized dispatchers, a group of DHT-based Data Lookup Services, and one worker per compute node. The provisioner is in charge of resource requests and releases. Currently, it is using a static resource provisioning strategy. The submitter is the only central point of the AME system. A submitter submits workflow descriptions to a number of dispatchers. The decentralized dispatcher takes the responsibility of uniformly dispatching a number of jobs to all compute nodes. The DHT-based data location lookup service implements a distributed memory coherence protocol, and provides file state and location lookup interfaces. In addition to running a job, the worker is capable of querying and updating the state and location of data, and stealing jobs from neighbors. Figure 3.1 shows the overview of the whole system.

The submitter runs on the login node. We select some of the compute nodes as the dispatchers, and the workers run on compute nodes. The submitter communicates with the job dispatcher via POSIX files on the shared file system. The submitter feeds the dispatchers with files containing job descriptions. The communication between dispatchers and the workers is over the interconnect network. Each dispatcher only has a local view of the jobs in its allocation. Communications among workers are over the interconnect network for the purpose of file transfer and load balancing. All communications use TCP/IP over the interconnect network.

By employing the idea of a DHT-based distributed memory coherence protocol, AME supports the feature of out-of-order execution. The submitter

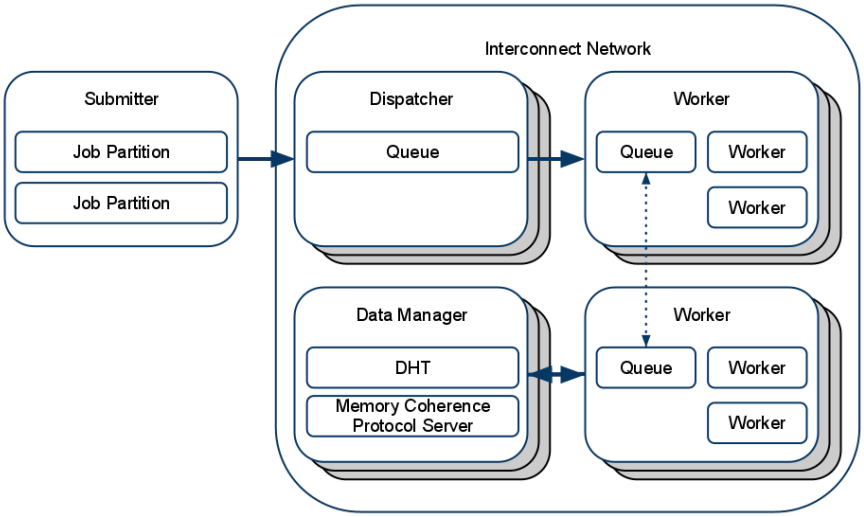

Figure 3.1: Overview of AME

submit alljobs regardless of data availability. Job dependencies and coherence are resolved by the protocol, thus preserving the execution order of the workflow. Another feature of the AME system is that the system utilization can scale up linearly in the unit of a pset (one I/O node with 64 compute nodes under its control). As the number of files increases proportionally with the number of jobs, we will have more distributed memory coherence protocol (DMCP) servers. Assuming we have a fixed ratio of DMCP servers to compute nodes, the number of file records that each server keeps remains the same. Thus the query and update workload does not increase with the number of jobs. In an ideal case where all jobs run for an identical time, the utilization of each pset remains constant as the system scales and number of jobs increase.

3.2 Dispatcher

The AME dispatcher has a 3-tier architecture. At the highest level, the submitter is a central point, it bundles the available jobs and sends them to the second level dispatchers proportionally to the number of workers. The

second level dispatchers send jobs to the workers in its range also in a uniform way. It keeps a record in memory for every job. Before a job is sent to worker, the second level dispatcher puts a tag in the job in order to mark the source of the job. Thus when a job is stolen across tree networks, the second level dispatcher could know where to route the result.

In a general view, AME uses supercomputers' interconnect network for inter compute nodes communication. It divides the allocation into serveral partitions. One of the compute node in each partition works as a second level dispatcher. Another compute node in each partition works as a data manager, while the remaining compute nodes work as workers.

3.3 Data Manager

3.3.1 Distributed Memory Coherence Protocol

We introduce this distributed memory coherence protocol (DMCP) to record file state transition. The DMCP protocol is implemented at two places, the worker and the DHT-based data location lookup services.

In a worker, the state transition logic tracks the state change of every file that is related to the jobs on the worker. There are four states in this protocol. For each file, its state is one of the following: INVALID, LOCAL, SHARED, REMOTE. INVALID indicates that this file is not available anywhere in the system, and it is expected to be generated by some job. LOCAL means this file is available on the local disk or the memory of this compute node. SHARED files are in shared file system. REMOTE files are available on some other compute node. There is a state transition from INVALID to REMOTE when an intermediate file is produced and its state is updated. Upon an update from INVALID to REMOTE, the protocol initiates a broadcast. It broadcasts the file location to all workers that have requested this file. After the intermediate file is copied from the producer to consumer, its state is update from REMOTE to LOCAL. State transition from Local to Shared only happens when an output file is written from local disk back to shared file system. As Many-Task applications have the multiple-read single-write pattern, there is no further state update once an intermediate file becomes LOCAL to the worker. Also, there is no more state transition to the output files that are moved to shared file system with states updated. Figure 3.2 shows the state transition of DMCP on compute nodes. Figure 3.3

The DMCP on the DHT-based lookup service have three states. INVALID, VALID, SHARED. The state INVALID indicates the file in not generated yet. The state VALID maps to the REMOTE and LOCAL states

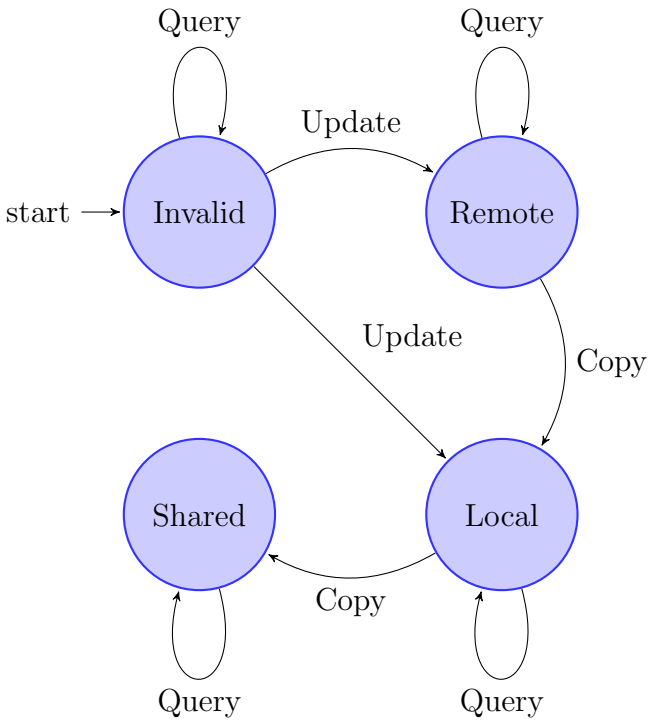

Figure 3.2: State Transition of Distributed Memory Coherence Protocol on Workers

in the implementation on workers, it does not track the location of the file in the interconnect network. The SHARED state shows that the data is copied to global file system. When an intermediate or output file is produced on workers, the state of the file is transitted from INVALID to VALID. Once an output file is copied from worker to global file system, the state of the file moves from VALID to SHARED.

The DMCP is used but not limited for POSIX-compatible file state transition tracking. It could also be used to track state change in in-memory data in other HPC programming paradigm.

3.3.2 DHT Based Data Lookup Service

Each of the Data Lookup Server has a in-memory hash table. The key for the hash table is a file name as a string. Each file name is unique in the namespace of one execution of Many-Task application. The associated value stores the status of the file, the location of the file, and an address list. The address list keeps track of the workers that request this file.

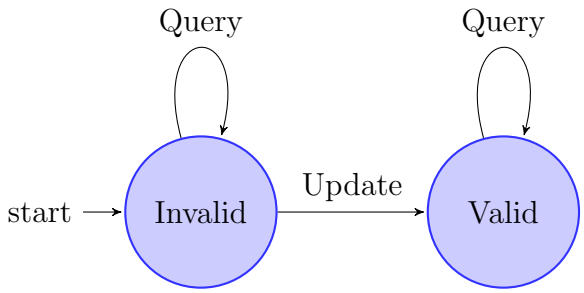

Figure 3.3: State Transition of Distributed Memory Coherence Protocol on Workers

We use a static approach for the DHT design, which is also known as consistent hash. Assuming there are no DHT servers coming and going during execution of one workflow, the information on all related files is distributed to all Data Lookup Servers by a hash function. The following equations are used to compute the target server for a given file:

```
Server_Rank = The rank of the server
Server_Num: The total number of servers
File_Name: The string of a file name
Hash_Value: The return value of
            the hash function

Hash_Value = Hash(File_Name)
Server_Rank = Hash_Value % Server_Num
```

In this way, the records are uniformly distributed on all Data Lookup Servers. And a worker uses the same way to find out which server to query for a given file.

3.3.3 Hashed Output Directory

To reduce the overhead produced by the Many-Task applications on the global file systems, we adopt the approach described in our previous paper [20]. A pre-created hashed output directory can significantly reduce the metadata server overhead by avoiding the locking mechanism in shared file systems.

3.4 Worker

The worker's main function is to execute jobs. In addition, it also has functions to enable job scheduling, work-stealing, etc. It keeps several data structures in memory: a queue that stores jobs received from the dispatcher, a ready-queue that stores all jobs that are ready to run, a result queue that keeps the results for finished jobs, a job hash map with job ID as key and job description as value to store jobs that have unavailable data, and a reverse hash map with file name as key and job ID as value. In the job hash map, there are also a pair of values that indicate the number of available input files and the total input files.

The DMCP protocol is implemented as follows: By looking at local data information, a file is either marked a LOCAL or INVALID. If a file is marked as INVALID, the worker will initiate a query to the Data Lookup Services. The services could return two results, either INVALID or REMOTE. In the case of a worker receiving INVALID for an input file, the Data Lookup Service links the IP address with the file, and worker will continue to check on the next input file. While if the worker receives the REMOTE message, the worker will add one to the count of available input files for the job. When a worker receives a broadcast message from the Data Lookup Services, it means some file has been updated to REMOTE. The worker will add one to the count of available input files of the job, and if the job is ready to run, the worker will launch a thread to execute it. When the job is executed, the worker will query the Data Lookup Service again to get the location of the file, then copy it from remote site, and the state of that file will be changed from REMOTE to LOCAL. After the execution, the worker will mark the output file as LOCAL. If the file is an output file, it will be moved from worker's RAM to global share file system, and the state of the file will changed from LOCAL to SHARED.

The worker has 10 active threads. The *fetcher* fetches jobs from dispatcher, and pushes them into the job queue. The *committer* pops results from the result queue and send them to scheduler. The *thief* steals jobs from the ready queues of its neighbors. The *victim* accepts the thief's requests. The *job mover* checks the availability of input files of the jobs in the job queue. If all input files are ready, the job is moved to the ready queue. Otherwise, the job is pushed to the job map. The *receiver* accepts broadcast messages from DHT servers. Upon every received message, it first finds the corresponding job ID in the reverse hash map, then adds one to the available input file count in the job map. If the available input file count equals to the total input file count, then the job is ready, and moved to ready queue. The other 4 threads are used to run the jobs.

Chapter 4

Design Alternatives

4.1 Job Scheduling

Centralized vsḊecentralized: In a centralized design, the submitter keeps states of jobs (it monitors the state transition of jobs.) It requires some amount of memory to store the return state, queuing time, running time, etc. In a decentralized design, the submitter only keeps the number of jobs for each dispatcher, initializes the dispatchers, and waits until all dispatchers return. It is easier for a centralized submitter to find out the status of jobs, and to rerun jobs that have failed or not returned. Removing the management logic from the submitter has two advantages: it relieves the submitter from using too much memory, and it places a lighter load on the submitter. Thus, it could enhance scalability.

4.2 Load Balancing

4.2.1 Push vs. Pull

In a Pull Model, dispatchers hold jobs in their queues, and workers pull jobs from them when they are idle. In a Push Model, the dispatcher initialize the communication to workers and pushes a job to a worker if it is idle. One limit of this algorithm is the central placed job load balancer, As Figure 5.3 shows, the centralized load balancer(dispatcher) can not linearly scale up with 64-second-long jobs. Even if the jobs are long enough to achieve high efficiency, the capacity of the centralized Scheduler will be limited by the CPU and the Memory of the scheduler.

The time-to-solution of a Push Model with known job duration is no worse than 1.5x of the optimal solution [11]. The Push model, in this design, pushes

all jobs to all available workers as uniformly as it can, as we are assuming the job dependencies can be resolved by the DMCP protocol. There is a good chance that a starvation situation occurs, where some of the workers are busy running with extra jobs are queued, while other workers are idle. To remedy this problem, we introduce the work-stealing algorithm.

4.2.2 Global Stealing vs. Local Stealing

A stealing scheme where each worker could steal from any other worker in the allocation, called Global Stealing, has been demonstrated [6]. Local Stealing is when each worker can only access part of the other workers' work. For example, a worker might only steal jobs from its neighbors. The latter scheme has a chance to incur a hot-spot situation, where a worker has a lot of jobs in the queue, while all its neighbors are busyand other, more-remote workers are idle. Here, no other workers would steal jobs from the hot-spot.

4.3 Data Management

4.3.1 Data Aware Scheduling vs. Distributed Coherence Protocol

Falkon has a data-aware scheduling feature [14]. Instead of moving the data to jobs, it route the jobs to the workers who have the data. The advantage of this scheme is that, for large data, which is more expensive to move than a job, it will lower the system overhead. It also places some challenges on the submitter, one of which is memory consumption. Memory limitations will limit the scalability of the whole system. In order to avoid the disadvantages of Data Aware Scheduling and to avoid a centralized point, the Distributed Coherence Protocol could be used. Each server keeps information about a certain part of the data, distributed by a hash function. When a worker queries or updates the state of a file, it could use the same hash function to find out the corresponding server in at most O(1) time.

4.3.2 Location Lookup vs. Data Store

To support intermediate data caching, either we could use a location lookup to find out where the data is, then move it in a peer-to-peer style, or we could build an intermediate file system on-the-fly. A location lookup service would have a smaller amount of data movement as it only would need to copy from the source to destination, while the data store would double the amount of

data movement, with one copy needed from the source to the data store, and another copy needed from the data store to the destination. Also, the data store scheme has to maintain the metadata consistency if it is distributed.

Chapter 5

Performance Evaluation

Our testbed is the IBM Blue Gene/P deployment at Argonne National Laboratory. It has 40960 quad-core compute nodes, each with 2 GB memory. A pset is a group of 64 compute nodes and one IO node corresponding to this group of nodes. Within the pset, the IO node and compute nodes communicate via the tree network. Each pset has a private IP space, so compute nodes can not communicate across psets via the tree network. A rack consists of 16 psets, thus 1024 compute nodes or 4096 cores. Compute nodes can reach each other via the 3D torus network, which is a global network for compute nodes in the same allocation.

5.1 Scheduling without Data

To show the AME dispatcher's performance, we use a suite of synthetic jobs. Each job runs for the same length. The lengths are 0 seconds, 1 seconds, 4 seconds, 16 seconds, 64 seconds. And we run 16 jobs on each core. Figures 5.1 and 5.2 show the different dispatching rates for the Centralized and Decentralized scheduler. The dispatching rate of the centralized scheduler increases linearly up to 8 psets, that is 512 compute nodes (2048 cores). From there, the increase slows down significantly due to the dispatcher's limited ability to manage traffic over sockets. For the decentralized scheduling case, the performance keeps increasing linearly up to 8192 compute nodes (32768 cores). The reason for this linear scalability is that the submitter partitions the job description file and only issues control traffic to the dispatchers, instead of sending jobs to them. The dispatch rate will stop increasing linearly at some point, either because the system hits the bottleneck of GPFS read performance, or the bottleneck of socket management.

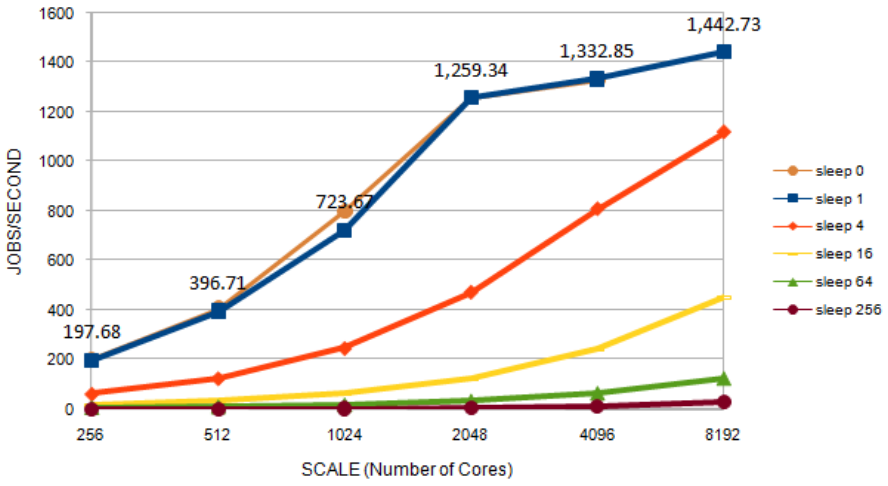

Figure 5.1: Dispatching Rate of Centralized Scheduler

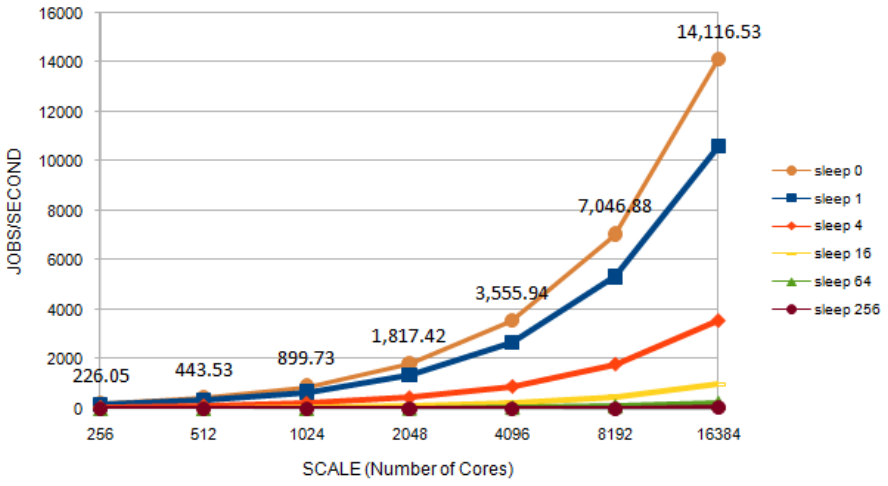

Figure 5.2: Dispatching Rate of Decentralized Scheduler

Figure 5.3 and 5.4 shows the workload efficiency in both of the centralized and decentralized schedulers. The efficiency is computed as

$$efficiency = \frac{job\_length * jobs\_per\_core * num\_cores}{time\_to\_solution * num\_cores} \tag{5.1}$$

We can tell from Figure 5.3 and 5.4 how long should the job be in order to achieve certain efficiency without concern of data. For the case of 2 racks, which is 2048 compute nodes (8192 cores), with the centralized scheduler, the job length needs to be longer than 16 seconds to achieve the efficiency of 90%. While in the decentralized case, the job length could be at least 4 second long to achieve 90% efficiency, which allows domain scientists more flexibility of job length when they design their workflows on supercomputers.

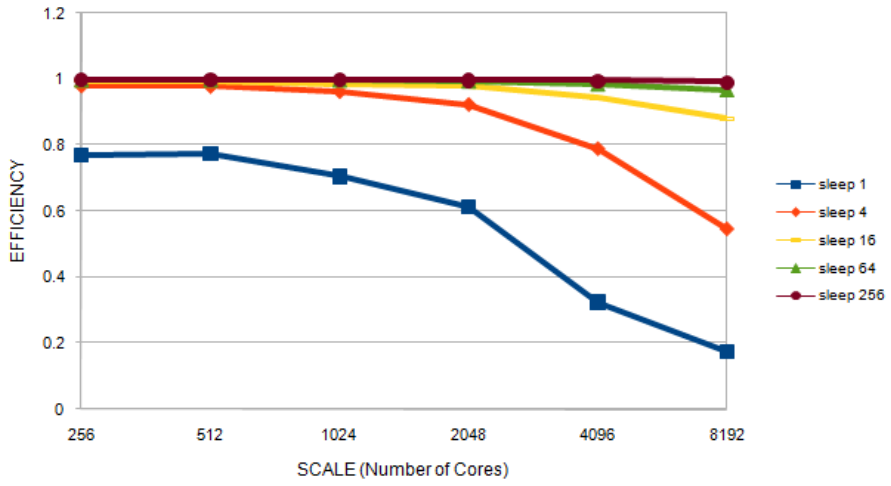

Figure 5.3: Efficiency of Centralized Scheduler

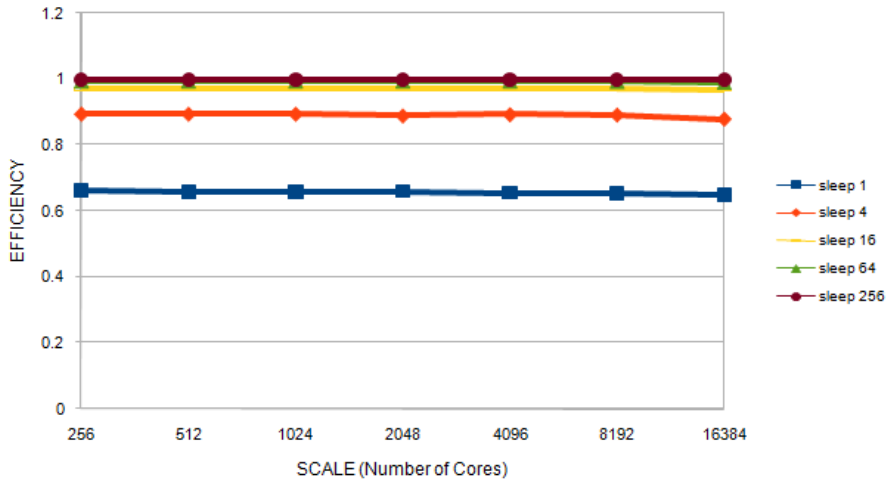

Figure 5.4: Efficiency of Decentralized Scheduler

5.2 Scheduling with Data

In the below tests, we use the same setting as the test suite in the above section with only one difference on job dependency. Among the 16 jobs each core receives, we add a job dependency between a pair of jobs, thus we have eight pair of jobs. The first job of each pair will run for the same job length, and output a file of 10 bytes. The second job of each pair of take the file from previous step as an input file, then runs for the same job length. We use the decentralized scheduler with the DHT-based DMCP services to conduct the tests.

Figure 5.5 shows the time to solution of the designed tests at various scale. Though some overhead is introduced by the intermediate data handling scheme, it remains almost constant as the scale goes up thanks to the consistent hashing scheme as shown by Figure 5.6. One thing to note is that the intermediate data management overhead decrease as the job length increases, this is because, longer running job will release the DMCP servers from traffic contension.

To show the impact of file size variation in the AME system, we select file size from 1 KB, 1 MB to 10 MB. The job length is set to 16 seconds, and each core runs 8 jobs. And there are 4 rounds of data transfer involved in the test. As the available memory on each compute node is no more than 600 MB, in this workload, each compute node has 16 cached files that are produced by its 4 cores, and another 16 files transferred from other compute nodes, with a total size of 320 MB. The ideal time-to-solution of this test

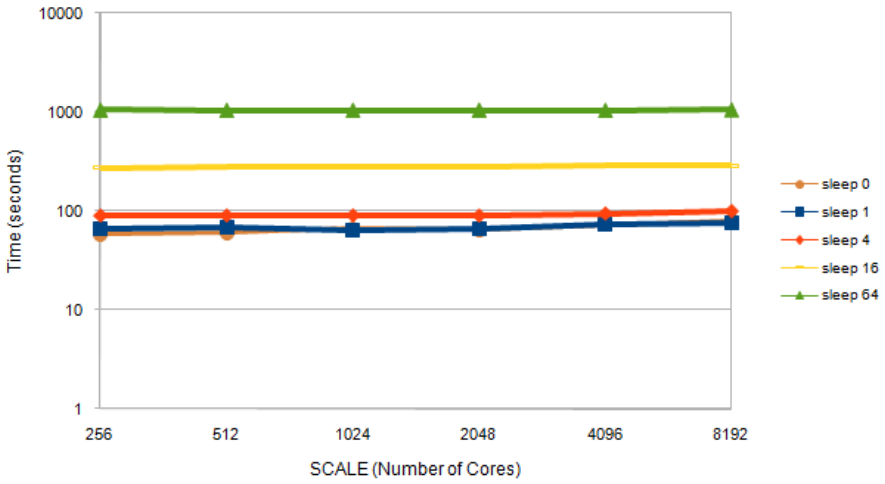

Figure 5.5: Time to Solution with Intermediate Data

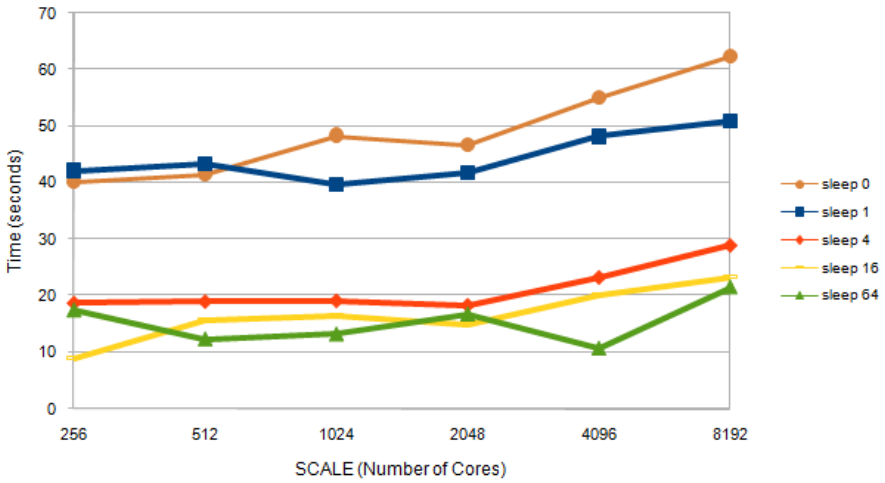

Figure 5.6: Overhead Introduced by Intermediate Data Handling Scheme

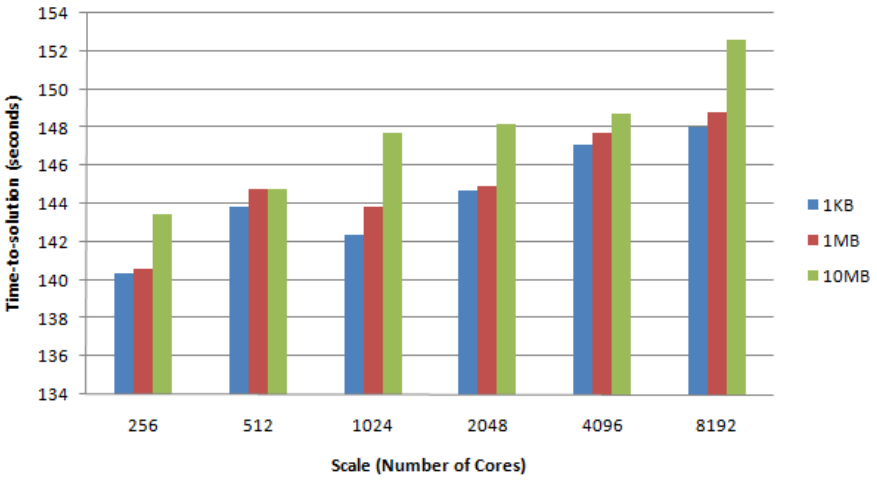

Figure 5.7: File Size Impact on Time-to-solution

should be 128 seconds. The overhead comes from two sources: job scheduling and intermediate data management. We see two trends in Figure 5.7: One is that within the same scale, the larger file size places more overhead, but the overhead is not significant. The other trend is that with the same size, larger scale places larger overhead. The overhead comes from the file transfer in the torus network, as it is a hop by hop transmission network, larger scale requires more hops to transfer the data. From 256 cores to 8192 cores, the overhead for 4 rounds of data transfer is ∼8 seconds.

5.3 Data Management Overhead Analysis

The overhead of intermediate data management comes from four potential sources: torus network congestion, Data Location Lookup Service queuing, hash table synchronization in the Data Location Lookup Service, and the CPU-saturated OS. The workers use two operations to access the Data Location Lookup Service. The first one is to *query* the state of some intermediate data. The other is to *update* the state of a piece of data. The first round of traffic is all workers querying intermediate data that has not yet been generated. The next eight rounds of operations are update operations. When the workers finish the jobs, they update the records of the corresponding data. Upon updates of the records, the Data Location Lookup Service broadcasts the data location to the workers who queried it. Then comes the next eight rounds of query operations to look up the locations of intermediate data.

	Query-Queuing	Query	Update-Queuing	Update
Average	144.31 ms	0.30 ms	2.45 ms	0.36 ms
Stddev	14.24 ms	7.15 ms	0.085 ms	0.14 ms

Table 5.1: Statistics of Data Location Lookup Service

Note that for each piece of data, a worker has two chances to determine its location: through the broadcast from the Data Location Lookup Service, or from the results of a second round query. This ensures that workers can find the correct location of data that is needed by a stolen job. In this case, there is no work-stealing involved, so the data location is broadcast to workers.

Table 5.1 shows the queuing time and processing time respectively at the Data Location Lookup Service side. We can tell that queuing and processing at server side is not a major source of overhead. The queuing time for query takes longer because the query traffic is congested when the workload starts.

At the worker side, as shown in Figure 5.8, the average time needed for a query is 148.35 ms (of which 144.31 ms is the queuing time), while the average update operation takes 3.05 ms (of which 2.45 ms is the queuing time). The query operation takes longer due to the queuing at Data Location Lookup Service side. Upon receiving the intermediate data location, the workers will initiate transfers to the remote peer to get the data. As Figure 5.8 shows, the data transfer takes 0.2 seconds on average, with standard deviation of 0.24 seconds. The latency comes from the CPU-saturated OS. Comparing between the rounds of data transfer, latter rounds are spread across a longer range of time, with a lower workload on the CPU, and thus lower data transfer latency. In the tests involving data, there are 8 threads in a worker running actively on a quad-core power CPU. Shorter jobs put a heavier load on thread switching on the Power CPU, thus producing a larger overhead.

5.4 Load Balancing

We choose the set of 15344 DOCK6 jobs in [3] as our test sample to show the load balancing performance of AME. We simulated this load balancing feature, and compare the time to solution of the same workload among three algorithms: centralized dispatcher+FIFO, decentralized dispatcher+PUSH,

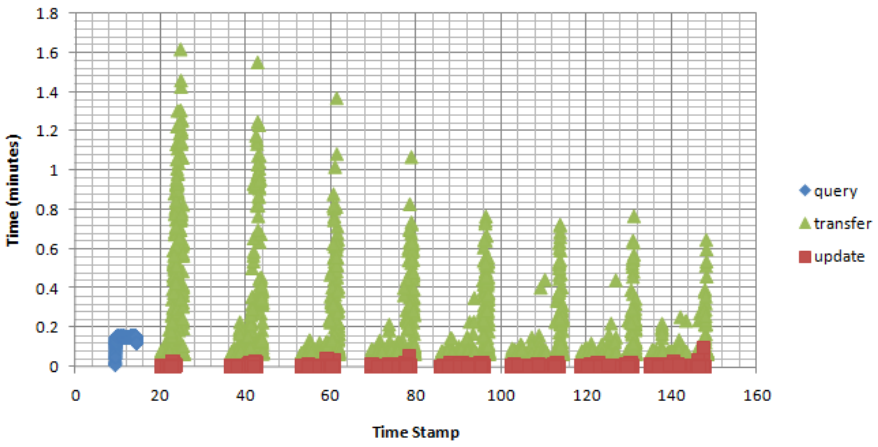

Figure 5.8: End-End Operation Time from Worker Side

and decentralized dispatcher+PUSH+work-stealing. The same workload runs at multiple scales to show the scaling performance of all three algorithms. Figure 5.9 shows the running time distribution of the workload: the average running time is 878.57 seconds, with a standard variation of 445.18 seconds.

As Figure 5.10 shows, the centralized dispatcher+FIFO algorithm is the best among all scales. The decentralized dispatcher+PUSH scheme trails by 14.05%, 29.07%, 33.22%, 52.76%, and 46.21% respectively at the scale of 256 cores, 512 cores, 1024 cores, 2048 cores and 4096 cores. With the work-stealing algorithm, the decentralized dispatching+PUSH scheme trails the centralized dispatcher+FIFO algorithm by 0.44%, 3.79%, 7.40%, 16.55%, and 11.00%. As for the local-stealing algorithm, it behaves almost identical to the decentralized dispatcher+PUSH scheme due to the hotspot situation. The hotspot situation is defined as a situation where a worker is saturated with extra jobs, and all its neighbors are busy, thus no other workers could steal jobs from this saturated worker.

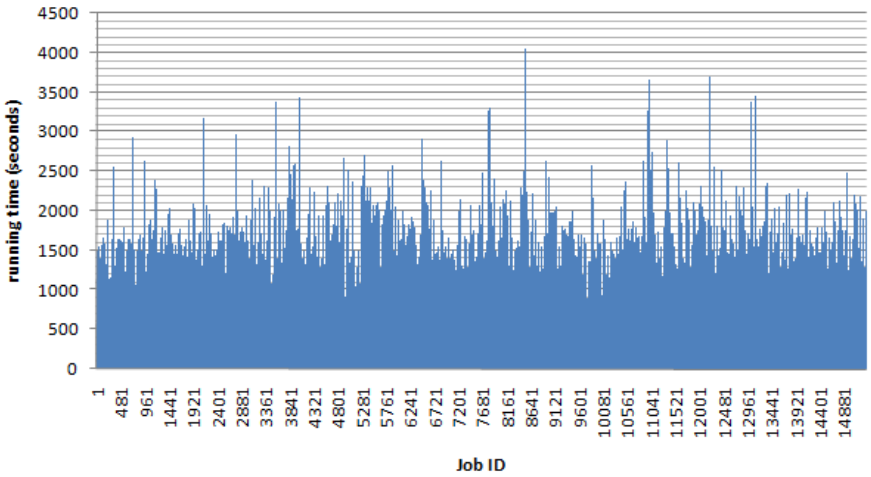

Figure 5.9: Running Time Distribution of the 15344 DOCK6 Job Workload

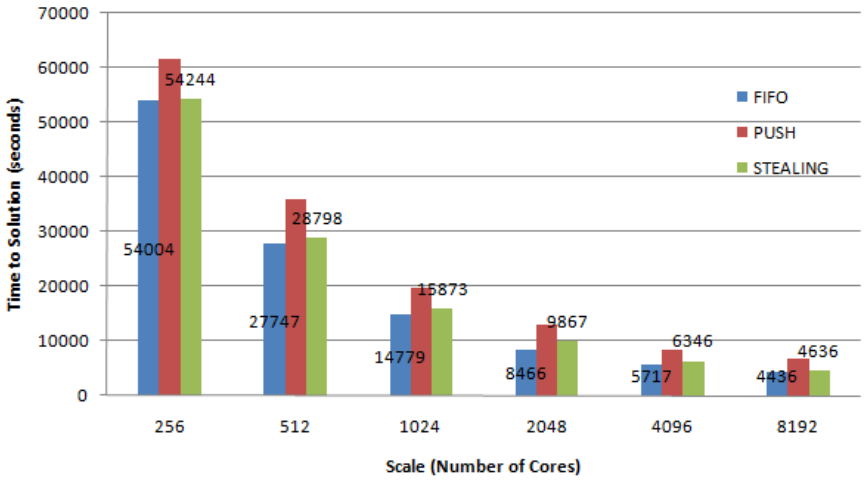

Figure 5.10: Performance of Load Balancing Algorithms

Chapter 6

Application Experiments

Montage is an astronomy application that composes small images from telescope into one large image. It has been successfully run over supercomputers and grids, with MPI and Pegasus respectively [10]. The Pegasus version of the Montage workflow has 9 stages, 3 of which involve steps that can be executed in parallel. In the AME version of Montage, we use 7 stages. Stage 1 is mProject, which takes in raw input files and outputs reprojected images. Stage 2 is mImgtbl, which takes the metadata of all the reprojected images, and generates a summary image table. Stage 3 is mOverlaps, which analyses the image table, and produces a metadata table describing which images overlap along with a job list of mDiffFit tasks (one for each pair of overlapping images). The fourth stage, mDiffFit, has jobs that take as input two overlapping output files from Stage 1 and fits a plane to the overlap region. Stage 5, mConcatFit, is similar to Stage 2; it gathers all output data from the previous stage (coefficients of the planes), and summarizes them into one file. mBgModel, Stage 6, analyses the metadata from Stage 2 and the data from Stage 5, creating a set of background rectification coefficients for each image, then generates a mBackground job list for the next stage. The last stage of the current workflow is mBackground, which actually applies the rectification to the reprojected images. Note the mBackground stage is the only stage where we move data from the compute nodes to GPFS; in all other stages, the data remains only on the compute nodes. The current version of AME cannot run the last stage of Montage, mAdd, due to its computation and data pattern. mAdd reads output files from mBackground, and writes an aggregated file, the size of which is close to the sum of the sizes of the input files. Instead, we would just run the Montage-provided sequential or MPI version of mAdd using the data that AME stored in GPFS in the previous stage. Improving this will be future work.

We ran a test of Montage that produces a 6 x 6 mosaic centered at galaxy

	# of jobs	1 core (s)	512 cores (s)	speedup
mProject	1319	21220.32	56.53	375.38
mDiffFit	3883	35960.12	95.32	377.27
mBackground	1297	9815.92	64.44	152.33

Table 6.1: Performance Comparison of AME and Single-node Execution

	GPFS (MB)	AME (MB)	Saving(%)
mProject-input	2800	2800	0%
mProject-output	5500	0.36	100%
mDiffFit-input	31000	0	100%
mDiffFit-output	3900	0.81	100%
mBackground-input	5200	0	100%
mBackground-output	5200	5200	0%
total	53600	8001.17	85.72%

Table 6.2: Comparison of Data Transfer Amount between GPFS and AME Approaches

M101. It has 1319 input files, each of which is 2 MB. Stage 1 outputs 1319 4-MB files. We ran the 2nd and 5th stage with the AME built-in reduction function. Stages 3 and 6 runs on the login node, as they analyze summarized files, and generate new jobs. Stages 1, 4, 7 each run in a parallel manner; they process the input/output data with the data management scheme we described in previous sections. Each job in Stage 7 writes a file of 4 MB size. We compare the performance of the 512-core approach with a single node execution to show speedup, as in Table 6.1. The time is measured in seconds. We haven't run the mAdd calculation, as this would not show any improvement.

The 1-core data is estimated from the performance of the login node, which is 4x times faster than a compute node. The mBackground stage has a lower speedup because it moves the output from compute nodes to GPFS. If we can run mAdd in a MTC style, then we could reduce this consumption by transferring data among compute nodes, and only port the mAdd output to GPFS. The mImgtbl and mBgModel stages are done with the AME built-in reduction function. The processing times are short, 9.6 and 14 seconds respectively. In this test, we reduce the data movement from compute nodes to GPFS by 45.6 GB, as shown in Table 6.2.

Chapter 7

Conclusion

None of the existing parallel programming language models were designed for exascale systems. Some of them, like MPI, might have a lower barrier to scaling up to exascale with some optimization, but some of them are themselves limited by their architecture. Nevertheless, to scale up a programming paradigm to the order of millions of CPU cores, we need to solve some common issues, which are perfectly covered by our five categories of gaps: resource provisioning, job scheduling, load balancing, data management and system resiliency.

AME is a Many-Task Computing engine that is designed for ultrascale supercomputers, with the focus on scalability. Using the principle of avoiding a central point, AME's schedulers dispatch jobs in a partially distributed manner, AME's intermediate data management scheme employ a linear scalable solution theoretically up to any scale, and AME's load balancing scheme relies on a work-stealing algorithm that is fully distributed across the allocation.

The benchmarks show that AME performs as expected. Scheduling performance increases linearly up to 16,384 cores. We are confident that performance will keep scaling up linearly until it hits the read performance bottleneck of the GPFS configuration. Even though the intermediate data management scheme introduces extra overhead, the overhead remains constant in the benchmark tests up to 8,192 cores. The benchmark of the work-stealing algorithm of AME shows that it performs no worse than the FIFO algorithm, and with future research on this topic, we will have other solutions to solve the large scale load balancing issue.

AME emphasizes its scalability on ultrascale machines with all of its jobs schedulers, data managers and load balancers. In the scheduling test on 16,384 cores, AME ran 262,144 jobs with variable job lengths. And in the data management test on 8,192 cores, the total number of files managed was

32,256. With 10 MB per file, the total file size was 323 GB.

AME improves running the Montage.The workflow that produces a 6x6 mosaic using 512 cores on BG/P handles 53.6 GB data in total. AME reduces data movement between compute nodes and GPFS from 53.6 GB to 8 GB, and significantly improves the utilization of the allocation during the run by reducing the computing cycles that are wasted during the data transfer.

Chapter 8

Future Work

Two main aspects of this work are going to continue. One of them is scaling up the intermediate data management scheme, and comparing it with alternative solutions such as intermediate data store. The other one is investigating the load-balancing algorithm, where we will design and evaluate more practical approaches for this problem.

One of the assumptions of this work is that we are scaling the POSIX semantics to peta-scale. We will also investigate and evaluate other approaches, such as other memory access techniques.

For the intermediate data management scheme, we will design and evaluate a file replication scheme based on the usage of the data across compute nodes.

To address the reliability of the system, we need to provide domain scientists with resilience features because the workflow run can fail during any part of the run. For this, failed and unreturned jobs could be retried explicitly by the scientists or automatically by the system.

Integrating the engine with existing parallel scripting language like Swift is another challenging area of work. We will identify the primitive semantics of parallel scripting languages and build them into the AME system. One basic question is how to support dynamic branching in the engine. Rather than solving that immediately, integrating AME with Pegasus is lower hanging fruit, as the workflow description in Pegasus could be directly translated to AME job descriptions.

With larger scale testing, we will answer a further question, which is a basic assumption of this work: Will congestion on the interconnect network dramatically increase as the scale increases? If so, we need to determine a geographic-aware algorithm to select the location of DMCP servers to minimize the traffic congestion.

Last but not least, we will collaborate with domain scientists from all

research areas to run more MTC applications with AME. AME will be the bridge between their MTC applications and high-end supercomputers.

Chapter 9

Acknowledgment

This work was partially supported by ExM award from DOE under contract number DE-SC0005380. We thank Kamil Iskra, Kazutomo Yoshii, and Harish Naik from the ZeptoOS team at the Mathematics and Computer Science Division, Argonne National Laboratory, for their effective and timely support. We also thank the ALCF support team at Argonne. Special thanks to Professor Rick Stevens of Department of Computer Science, the University of Chicago for his enlightening class.

Bibliography

[1] Samer Al-Kiswany, Abdullah Gharaibeh, and Matei Ripeanu. The case for a versatile storage system. *SIGOPS Oper. Syst. Rev.*, 44:10–14, March 2010.

[2] Bill Allcock, Joe Bester, John Bresnahan, Ann L. Chervenak, Ian Foster, Carl Kesselman, Sam Meder, Veronika Nefedova, Darcy Quesnel, and Steven Tuecke. Data management and transfer in high-performance computational grid environments. *Parallel Comput.*, 28:749–771, May 2002.

[3] Tim G. Armstrong, Zhao Zhang, Daniel S. Katz, Michael Wilde, and Ian Foster. Scheduling many-task workloads on supercomputers: Dealing with trailing tasks. In *Proceedings of Many-Task Computing on Grids and Supercomputers, 2010*, 2010.

[4] Dhruba Borthakur. HDFS architecture. http://hadoop.apache.org/common/docs/r0.20.0/hdfs_design.pdf.

[5] Philip H. Carns, Walter B. Ligon, III, Robert B. Ross, and Rajeev Thakur. PVFS: a parallel file system for linux clusters. In *Proceedings of the 4th annual Linux Showcase & Conference - Volume 4*, pages 28–28, Berkeley, CA, USA, 2000. USENIX Association.

[6] Jim Crammond. Scheduling and variable assignment in the parallel parlog implementation. In *Proceedings of the 1990 North American conference on Logic programming*, pages 642–657, Cambridge, MA, USA, 1990. MIT Press.

[7] Stephanie Donovan, Gerrit Huizenga, Andrew J. Hutton, Andrew J. Hutton, C. Craig Ross, C. Craig Ross, Linux Symposium, Linux Symposium, Linux Symposium, Martin K. Petersen, Wild Open Source, and Philip Schwan. Lustre: Building a file system for 1,000-node clusters, 2003.

[8] James Frey, Todd Tannenbaum, Miron Livny, Ian Foster, and Steven Tuecke. Condor-G: A computation management agent for multi-institutional grids. *Cluster Computing*, 5:237–246, 2002. 10.1023/A:1015617019423.

[9] Kamil Iskra, John W. Romein, Kazutomo Yoshii, and Pete Beckman. ZOID: I/O-forwarding infrastructure for petascale architectures. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 153–162, New York, NY, USA, 2008. ACM.

[10] Daniel S. Katz, Joseph C. Jacob, G. Bruce Berriman, John Good, Anastasia C. Laity, Ewa Deelman, Carl Kesselman, and Gurmeet Singh. A comparison of two methods for building astronomical image mosaics on a grid. In *Proc. 2005 International Conference on Parallel Processing Workshops*, pages 85–94, 2005.

[11] Jon Kleinberg and Eva Tardos. *Algorithm Design.* Addison Wesley, 2005.

[12] E. L. Lusk, S. C. Pieper, and R. M Butler. More scalability, less pain: A simple programming model and its implementation for extreme computing. *SciDAC Review*, 17, 1 2010.

[13] Ioan Raicu, Yong Zhao, Catalin Dumitrescu, Ian Foster, and Michael Wilde. Falkon: a Fast and Light-weight tasK executiON framework. In *Proc. IEEE/ACM Supercomputing 2007*, pages 1–12, 2007.

[14] Ioan Raicu, Yong Zhao, Ian T. Foster, and Alex Szalay. Accelerating large-scale data exploration through data diffusion. In *Proceedings of the 2008 international workshop on Data-aware distributed computing*, DADC '08, pages 9–18, New York, NY, USA, 2008. ACM.

[15] Frank Schmuck and Roger Haskin. GPFS: A shared-disk file system for large computing clusters. In *In Proceedings of the 2002 Conference on File and Storage Technologies FAST*, pages 231–244, 2002.

[16] Douglas Thain, Christopher Moretti, and Jeffrey Hemmes. Chirp: a practical global filesystem for cluster and grid computing. *Journal of Grid Computing*, 7(1):51–72, 2009.

[17] Rajeev Thakur, William Gropp, and Ewing Lusk. Data sieving and collective I/O in ROMIO. *Frontiers of Massively Parallel Processing, Symposium on the*, 0:182, 1999.

[18] Michael Wilde, Ian Foster, Kamil Iskra, Pete Beckman, Zhao Zhang, Allan Espinosa, Mihael Hategan, Ben Clifford, and Ioan Raicu. Parallel scripting for applications at the petascale and beyond. *Computer*, 42:50–60, 2009.

[19] Justin M. Wozniak and Michael Wilde. Case studies in storage access by loosely coupled petascale applications. In *Proc. 4th Annual Workshop on Petascale Data Storage*, pages 16–20, 2009.

[20] Zhao Zhang, Allan Espinosa, Kamil Iskra, Ioan Raicu, Ian Foster, and Michael Wilde. Design and evaluation of a collective I/O model for loosely coupled petascale programming. In *Proceedings of Many-Task Computing on Grids and Supercomputers, 2008*, pages 1–10, 2008.