AME: An Anyscale Many-Task Computing Engine

Zhao Zhang*, Daniel S. Katz†, Matei Ripeanu‡, Michael Wilde†§, Ian Foster*†§
*Department of Computer Science, University of Chicago
†Computation Institute, University of Chicago & Argonne National Laboratory
‡Department of Electrical and Computer Engineering, University of British Columbia
§Mathematics and Computer Science Division, Argonne National Laboratory

***Abstract*—Many-Task Computing (MTC) is an emerging programming model whose relevance on supercomputers is increasing, driven by applications in biology, economics, and statistics, and by paradigms such as data intensive computations and uncertainty quantification. However, its high inter-task parallelism and data-intensive processing capabilities pose new challenges to existing supercomputer hardware-software stacks. These challenges include resource provisioning; task dispatching, dependency resolution, and load balancing; data management; and resilience. This paper examines the characteristics of MTC applications which create these challenges, and identifies related gaps in MTC middleware for extreme-scale systems. Based on this analysis, we propose AME, an Anyscale MTC Engine, which addresses the scalability aspects of these gaps. We describe the AME framework and present performance results for both synthetic benchmarks and real applications. Our results show that AME's dispatching performance linearly scales up to 14,120 tasks/second on 16,384 cores with high efficiency. The overhead of the intermediate data management scheme does not increase significantly up to 16,384 cores. AME eliminates 73% of the data transfer between compute nodes and the global filesystem for the Montage astronomy application on 2,048 cores. Our results indicate that AME scales well on today's petascale machines, and is a strong candidate for exascale machines.**

***Keywords*-Many-Task Computing; scheduling; load balancing; data management; supercomputer systems.**

I. Introduction

As computers have become more powerful, the simulations and data processing applications that use them have become increasingly resource hungry, and, at the same time more complex. Simulation complexity has increased in the number of dimensions (from 1D to 2D to 3D), in the set of equations being simulated (from one equation, to multiple equations in one domain, to multiple equations in multiple domains), and in the number of time scales being studied simultaneously. Similarly, data-intensive applications are being composed from increasingly complex analyses. In both cases, achieving increased scientific productivity demands the integration of an increasing number of such applications into larger meta-applications. This can be achieved by adding additional layers around the initial application, as is done in optimization, uncertainty quantification, or parameter sweeps. Such meta-applications can be considered many-task computing (MTC) applications, as they are assembled from diverse task patterns, each of which may be as simple as a procedure call or as complex as a complete standalone application. Each distinct task has unique data dependencies, and the entire MTC application is often viewed as a directed graph of these dependencies. In many cases, the data dependencies take the form of files that are written to and read from a file system shared between the compute resources; however, tasks in MTC applications can also communicate in other manners. MTC data dependency patterns vary among applications, several of which have been characterized by Wozniak et al. [1], [2]

Today's most powerful supercomputing systems, many of which are being used to run MTC applications (e.g., IBM BG/P; Cray XE and XT; Sun Constellation) have a set of common features that include: a large number of multicore compute nodes that may have RAM-based filesystems but no local persistent storage; additional service nodes for interactive tasks, compilation, and job submission; one or more low-latency communication networks; and a globally shared persistent file system. Compute nodes generally have POSIX-compliant access to the shared file system, and some systems provide a complete Linux kernel. The scheduling granularity (i.e., the smallest number of compute nodes that can be allocated to a job) varies, but the minimum resource allocation unit can be 64 or more nodes, as is the case for the Intrepid BG/P supercomputer deployed at Argonne.

The resource management stack of these large computing resources, and indeed the machines themselves, have not been designed for MTC applications. These machines mostly run MPI-based applications and have been optimized for such "HPC" workloads. Thus it is natural that the existing hardware/software architecture is inadequate for MTC applications. Our experience [3] confirms that naïvely running MTC applications on the existing hardware/software stack will often result in a series of problems, including low machine utilization, low scalability, and file system bottlenecks. To address these issues, rather than aiming for a complete re-engineering of the resource management stack, this paper explores optimization avenues within the context of existing, heavily-adopted resource management systems: we propose and evaluate scheduling and data-storage mechanisms that address or alleviate the scalability, performance, and load-balancing issues mentioned.

More concretely, we propose AME, an Anyscale MTC Engine, designed to be compatible with existing supercomputer hardware/software stacks. AME features a linear task submission rate, linear performance for task dependency resolution, and linear data transfer performance for a commonly used data-dependency "pipeline" pattern. By plugging in AME,

MTC applications can fully benefit from the computational capacity of today's supercomputers.

Examining the characteristics of MTC applications and the resource management stack of current supercomputers, we identify six gaps:

① *Resource provisioning*: A first gap lies between the static resource provisioning scheme generally used and the variability in the run times of MTC tasks. Well known schedulers such as PBS offer a static scheduling solution, where it is not possible to release some of the computing resources while the job is still running. The result of coarse scheduling granularity on current machines is that in some stages, the number of ready-to-run tasks of an MTC application is lower than the scheduling unit, leading to low utilization.

② *Task dispatching*: Most existing supercomputer schedulers incur many seconds to several minutes of latency when starting and/or terminating allocations. This is an unacceptable overhead for MTC applications that have tasks durations of a few seconds or less.

③ *Task dependency resolution*: At the scale of today's largest machines, which are approaching 10^6 cores, task dependency resolution must be done in parallel, yet no such scheme has previously existed for MTC applications.

④ *Load balancing*: To obtain high machine utilization, MTC applications require workload-specific load balancing techniques.

⑤ *Data management*: MTC applications often exhibit an intensive I/O and data management load that overwhelms a supercomputer's I/O subsystems (in particular, its shared file systems), which are not provisioned and sometimes not even designed for this type of highly-concurrent and metadata-operation-intensive workload.

⑥ *Resilience*: The lack of resilience mechanisms in current resource management frameworks poses another challenge for MTC applications. Various failures may occur while an MTC application is running. In current systems, hardware and operating system failures at the node level lead to canceling an entire allocation. Such failures are not recoverable at the MTC engine level. When these failures occur, the challenges of recovery include identifying completed tasks, inferring the task dependencies of incomplete and failed tasks, and re-establishing the states of various services of the runtime system. These capabilities are not provided by the resource management stacks of current supercomputers.

In some cases, task dispatching (②) and load balancing (④) are interleaved within a scheduler. For example, a centralized task scheduler that sends the longest task to the next available compute node also balances the load among the compute nodes. In the following discussion, the term *dispatcher* denotes the scheduler's role in the task dispatching scenario, while the term *load balancer* refers to the load balancing role of either the scheduler or an independent load balancing service.

In this paper, we address three of the above gaps, presenting solutions for task dispatching (②), task dependency resolution (③, and data management (⑤). Resource provisioning (①) is usually closely coupled with system administration policy, such as minimum allocation unit and job limit per user. In some cases, specific node allocations are based on a network topology, and partially deallocating resources is not feasible; thus dynamic provisioning can not be applied. Load balancing (④) and resilience (⑥) will be addressed in future work. Our approach can be summarized as follows:

- To address the task dispatching gap (②), we take advantage of previous lessons from Falkon [4] and design and evaluate a multi-level dispatcher. In addition, we evaluate the tradeoffs between centralized and decentralized dispatching designs.
- Our solutions to address the task dependency resolution (③) and data management (⑤) gaps are closely coupled. We present a Distributed Data Availability Protocol (DDAP) that supports the common "single-write multiple-read" pattern of MTC applications. This protocol also supports task dependency resolution by tracking file existence states at runtime. The protocol is closely tied to a location service and is implemented on top of a distributed key-value store. To address the data management gap (⑤), we classify data passing according to the usage patterns described in Zhang et. al.[3] as common input, unique input, output, and intermediate data. Our focus here is on the optimized handling of intermediate data, and our data availability protocol and lookup service supports this optimization.

The rest of the paper is organized as follows. In §II, we discuss both previous work in this domain and related work with intriguing ideas from other domains. In §III, we present the high level design of AME and the communication among modules of the system. We present the benchmark design in §IV, and explain performance results in §V. Specifically, in §V-A, we evaluate AME task dispatching performance by comparing two design alternatives. Taking the result as a baseline in §V-B, we further evaluate the scalability and the impact of file size on the intermediate data management scheme. We conduct an in-depth overhead analysis in §V-B. Lastly, we compare two intermediate data placement alternatives, and evaluate their scalability and overheads in §V-C. In §VI, we examine a real-world MTC application that exhibits diverse data-flow patterns. We conclude in §VII, and summarize future work in §VIII.

A major contribution of this work is AME as a whole. AME enables the execution of the new class of MTC applications on supercomputers, with good performance and high system resource utilization. A second contribution is the Distributed Data Availability Protocol (DDAP), which resolves task dependencies in MTC applications that exhibit the single-write multiple-read pattern. DDAP is a protocol that is independent of any specific data format (e.g., the data could be a POSIX file or a in-memory structure). Along with a distributed key-value

store and a lookup service, DDAP resolves task dependencies in a distributed manner with linear scalability (in a weak-scaling sense). AME supports MTC applications developed on top of the Pegasus [5] and Swift [2], [6] workflow specification tools, through a translator that converts a Pegasus workflow description or a Swift script into an AME task description.

II. Related Work

Related and previous work are categorized here with respect to the set of gaps we address (defined in §I).

A. Task Dispatching

Regardless of the programming paradigm, the MTC application specification needs to be translated to machine code that can be executed. Different programming frameworks, in both the parallel and distributed program contexts, have different solutions to dispatch tasks to workers. MPI leaves this to the programmers. In general, MPI programs include the code for all the tasks that may be run, and each compute node does its part of the work, as identified by the worker's rank. This scheme is the most scalable of the ones we found, but it requires the compute nodes to load redundant information: every compute node needs to load the full compiled binary. Pegasus/Condor [5] uses a centralized task dispatcher, the submit host, which keeps a shadow of every single task. It tracks the lifetime state changes of the tasks. Thus its scalability is limited to the capacity of the submit host. Additionally this solution consumes a lot more memory on the submit host than the MPI case. While Falkon [4] uses a three-tier architecture (a first-tier submit host, a group of second-tier dispatchers, and a group of third-tier workers), it still tracks task status on the single first-tier submit host, thus the scalability of running short tasks (O(1) s) stops growing linearly at some point (which is dependent on the specific system, e.g., on the BG/P it is at 4096 cores.) AME's task dispatching mechanism employs the same three-tier architecture as Falkon, but AME's dispatcher does not monitor the status of each task, which results in higher scalability than Falkon; it makes the other choice in the tradeoff between scalability and detailed task status monitoring.

B. Task Dependency Resolution

Pegasus/Condor [5] lets users explicitly compose a workflow, while with the parallel scripting language Swift [2], [6], workflow composition is implicit and dynamic. Nevertheless, both systems use a a centralized submit host to resolve task dependencies. AME can parse either a Pegasus workflow description and or a Swift script, convert them to AME ad-hoc task descriptions, and implicitly resolve the dependencies at runtime in a distributed manner.

C. 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 [7] works with the computer node OS kernel to accelerate I/O throughput from computing resources to persistent storage. GPFS [8], LUSTRE [9], and PVFS [10] aim to provide scalable global persistent storage for supercomputers. GridFTP [11], MosaStore [12], and Chirp [13] provide data management primitives on grids and clusters at workflow runtime. MPI I/O, including the ROMIO [14] implementation, is designed to support parallel I/O in MPI; it can be viewed as a 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 [15] (Hadoop Distributed File System) places three replicas of each data chunk across the compute nodes. Other work tries to isolate the data storage and processing. HDFS's scalability is mainly limited by its single management node architecture. AME's data management system is designed to support MTC applications. It differs from persistent storage in terms of the lifetime of the data it manages. Using a scalable DHT-based design (described in §III), it could theoretically scale up to any number of compute nodes.

III. AME Design

The AME system currently tackles three gaps, as previously discussed: task dispatching, task dependency resolution, and data management. AME consists of five modules: a provisioner, a submitter, a group of decentralized dispatchers, a group of DHT-based Data Location Lookup Services (DDLLS) servers, and one worker per compute node. The provisioner is in charge of resource requests and releases. It currently uses a static resource provisioning strategy. The submitter is the only central point in the AME system. The submitter submits workflow descriptions to a number of dispatchers. The decentralized dispatchers uniformly dispatch tasks to all workers. The key-value store based DDLLS implements a distributed data availability protocol, and provides file state and location lookup interfaces. In addition to running a task, a worker is capable of querying and updating the state and location of data, and stealing tasks from neighbors. Fig. 1 shows the overview of the whole system.

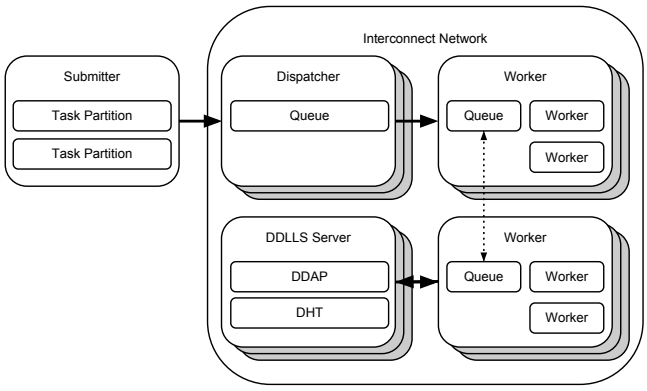

Fig. 1. AME overview

The submitter runs on the login node while task dispatchers,

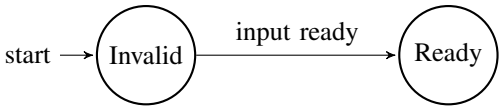

Fig. 2. Task state transition diagram

DDLLS servers, and workers run on compute nodes. The submitter communicates with the task dispatcher via POSIX files on the shared file system. These files contain task descriptions. All communications among dispatchers, DDLLS servers, and workers are through the supercomputer's interconnect network. The dispatchers only have a local view of the tasks in their allocations; they talk to a designated group of workers to dispatch tasks and collect task status. The DDLLS servers exchange messages with workers to provide the data location and data state change. Workers communicate among themselves for data transfer.

One important notion in AME is an AME-unit. Each AME-unit comprises a set of nodes that contains one dispatcher, a group of workers, and one or several data managers. Each individual node could contain the dispatcher and/or one or more data mangers and/or a worker.

AME's distributed data availability protocol (DDAP) is used to resolve task dependencies. The submitter submits all available tasks, regardless of data availability, and AME guarantees that the tasks are launched in an order that satisfies the dependencies. A feature of the AME system is that support services can be scaled up to match the overall system scale and workload intensity. For example, by maintaining a fixed ratio of DDLLS servers to compute nodes, the number of file records that each server manages remains stable regardless of system scale. Thus the per-node query and update workload will not increase with the number of tasks. In an ideal case, where all tasks run for an identical time, the utilization of each AME-unit remains constant as the system scales and number of tasks increase.

We define task states as INVALID or READY based on the availability of their input data. All tasks are initiated as INVALID. As input data becomes available, either because it already exists before execution or is produced during the run, the state of the associated task is changed from INVALID to READY, as shown by Fig. 2. A READY task state indicates a task can be executed. This data-driven state transition separate task management logic from data management logic. The DDAP and the DDLLS simply monitor the state of every single piece of data. Task dependencies are resolved based on data state updates. Tasks assigned to a worker are executed in a first READY, first run manner. Task dependencies in AME are implicit; they are extracted from the workflow script. In other tools (e.g., Pegasus), they are explicit; the user has to specify them in the workflow description.

A. AME Execution Model

Fig. 3 shows the AME execution model. ① Initially, a user describes a workflow in the Swift [2], [6] language. Then the Swift script is compiled into ② a file list and ③ a task list. ④ Once AME starts, the file list is loaded by the Distributed Data Location Lookup Service (the DDLLS), and file records are initialized accordingly. ⑤ All task descriptions are dispatched to workers, and each worker maintains a local queue of tasks. ⑥ Workers talk to the DDLLS to find out the location of the files, in this case, the intermediate files. Workers communicate with each other either ⑦ to forward a task to another worker for the purpose of locality or ⑧ to copy a file that was produced remotely. (Note that locality is the subject of ongoing research and is not further addressed in this paper.)

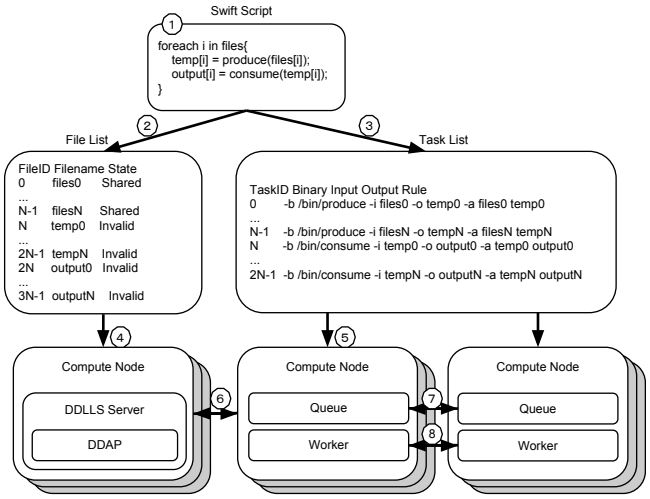

Fig. 3. AME execution model

B. Distributed Memory Coherence Protocol

We introduce the distributed data availability protocol (DDAP) to record file state transition. The DDAP is implemented in two places, the worker and the DHT-based Data Location Lookup Services.

In a worker, state transition logic tracks the state change of every file that is related to tasks on the worker. There are four states in this protocol: INVALID, LOCAL, SHARED, and REMOTE. INVALID indicates that this file is not available anywhere in the system; it is expected to be generated by some task, but the task has not yet run. LOCAL means this file is available on the local disk or the memory of this compute node. SHARED files are in the 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. Upon an update from INVALID to REMOTE, the protocol initiates a broadcast of the file location to all workers that have requested the file. After an intermediate file is copied from the producer to consumer, its state (on the consuming node) is updated from REMOTE to LOCAL. A state transition from LOCAL to SHARED only happens when an output file is written from a local disk to the shared file system. As MTC applications have a single-write multiple-read pattern, once an intermediate file is written, its state is

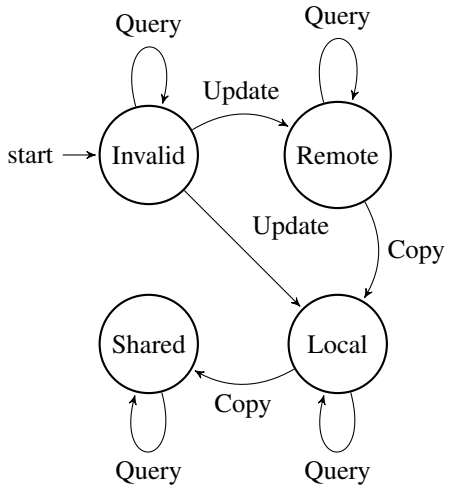

Fig. 4. Local file state transition diagram, as used in the DDAP on the workers

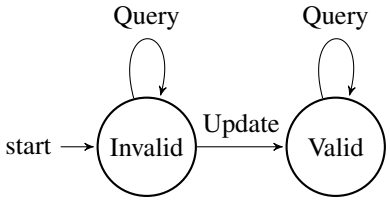

Fig. 5. Global file state transition diagram, as used in the DDAP on the DDLLS servers

LOCAL and there will be no further state updates. Files that are application outputs, on the other hand, will be copied to persistent storage and their final state will be SHARED. Fig. 4 shows these state transitions.

The DDAP on the DDLLS has two states: INVALID and VALID. All intermediate files are initialized as INVALID. Upon a state query for an INVALID file, the protocol returns the "not available" message, and links the address of the worker to this file. Once an INVALID file is produced by a worker, the producer updates the DDAP for that file from INVALID to VALID. The DDAP associates the producer's address with the file, and broadcasts the producer's address to all workers who have queried this file. After this, queries on the VALID file will return the address of the producer. Fig. 5 shows these state transition.

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

C. DHT-based Data Location Lookup Services

Each of the Data Location Lookup Servers 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 a MTC 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.

We use a static approach for the DHT design. (In this work, we assume that no DHT servers leave or join during execution of one workflow. This can be generalized by use of a reliable DHT service, e.g., Chord [16].) The information on all related files is distributed to all Data Lookup Servers by a consistent 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 DDLLS servers. Workers use the same method to find out which server to query for the status of a given file.

To reduce the overhead produced by the MTC applications, we adopt the approach described in our previous paper [3]. A pre-created hashed output directory can significantly reduce the metadata server overhead by avoiding the locking mechanism in GPFS and other shared file systems that don't already have hashed metadata servers.

D. Dispatcher

The AME dispatcher has a three-tier architecture. At the highest level, the submitter is a central point; it provides tasks to the second level dispatchers proportionally to the number of workers associated with each dispatcher. The second level dispatchers send tasks to the workers in its range also in a uniform way. Each dispatcher keeps a record in memory for every task that it owns. Before a task is sent to a worker, the second level dispatcher puts a tag in the task in order to mark the source of the task. (This tag is required for routing of tasks when work stealing is used, but this is related to load-balancing and data-aware scheduling that are not covered in this paper.)

E. Worker

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

The worker has four active data coordination threads. The *fetcher* fetches tasks from the dispatcher, and pushes them into

the task queue. The *committer* pops results from the result queue and send them to dispatcher. The *task mover* checks the availability of input files of the tasks in the task queue. The *receiver* accepts broadcast messages from DHT servers. Upon every received message, it first finds the corresponding task ID in the reverse hash map, then adds one to the available input file count in the task map. In addition, the worker has one thread per core that is used to run the tasks. On the BG/P, there are four such threads.

IV. AME Design Alternatives

In this section, we describe two of the choices we have made in the design on AME, and why we have made them. In §V, we will show experimental results of these choices.

A. Centralized vs. Decentralized Dispatching

In a centralized design, the submitter keeps track of the states of the tasks (i.e., it monitors the state transition of tasks.) It requires some amount of memory to store the return state, queuing time, running time, etc. In a decentralized design, the submitter only keeps track of the number of tasks for each dispatcher, initializes the dispatchers, and waits until all dispatchers return. It is easier for a centralized submitter to find the status of tasks and to rerun tasks that have failed or not returned. Removing the management logic from the submitter has two advantages: it reduces the amount of memory used by the submitter, and it reduces the amount of work done by the submitter. Thus, it enhances scalability, as will be shown in §V-A. In the centralized design, there is a tradeoff between scalability and efficiency, while the decentralized design has better efficiency at large scale, but sacrifices centralized task information.

B. Collocated vs. Isolated Data Processing and Storage

To support intermediate data caching, we could either use an intermediate file system that is aggregated over some dedicated compute nodes on-the-fly, or we could use a distributed data store spread on all compute nodes while they run tasks. We refer to the former case as isolated data processing and storage (isolated), and the latter as collocated data processing and storage (collocated). In both cases, the DDLLS is similarly used, as tasks that need intermediate data files will need to find out where they are. However, in the collocated case, files are only copied once, from their source directly to their destination, while in the isolated case, each file is copied twice, from its source to the data store and from the data store to its destination. Both the collocated and isolated schemes have been implemented in AME, and their performance is discussed in §V-C.

V. Performance Evaluation

We have used the IBM BG/P *Intrepid* at Argonne National Laboratory for performance testing of AME. It has 40,960 quad-core compute nodes, each with 2 GB of memory. Intrepid is composed of 640 psets, each of which is a group of 64 compute nodes and one associated I/O node. Within a pset, the I/O node and compute nodes communicate via a tree network. Compute nodes in different psets communicate via a 3D torus network.

AME on BG/P uses a single submitter on a login node. We divide our allocated resources into AME-units of 64 compute nodes, and on each, we run the dispatcher and the data manager on a single node, with the other 63 nodes running one worker each.. The submitter and dispatchers communicate via the shared file system, and the workers, dispatchers, and data managers communicate over the torus network.

A. Dispatching, without data transfer

We first show the AME dispatching performance with both the centralized and decentralized design. In the centralized design test, we want to find out the appropriate scale, given the task length and required efficiency, while in the decentralized design, we try to verify that the performance remains constant as the scale increases. We first test the AME dispatcher's performance by running a suite of synthetic tasks. We run 16 tasks on each code, each of which runs for the same amount of time, 0, 1, 4, 16, or 64 seconds, in a given test. Figs. 6 and 7 show the dispatching rates for the centralized and decentralized dispatcher. The dispatching rate of the centralized dispatcher increases linearly up to 512 nodes (2,048 cores). From there, the increase slows down significantly due to the login node's limited ability to manage traffic over sockets. In the decentralized dispatching case, the performance keeps increasing linearly up to 4,096 nodes (16,384 cores). The reason for this improvement is that the submitter partitions the task description file and only issues control traffic to the dispatchers, instead of sending tasks to them. The dispatch rate will stop increasing linearly at some point, because the system hits the GPFS read performance limit.

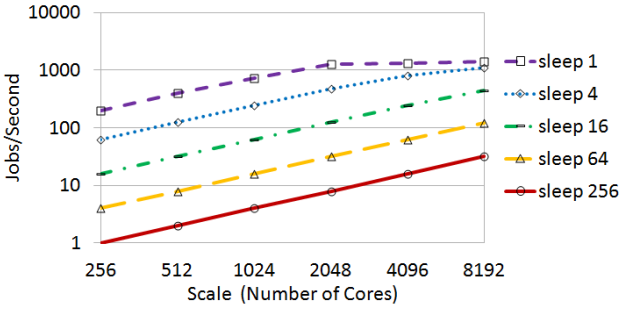

Fig. 6. Dispatching rate of centralized dispatcher

Figs. 8 and 9 shows the workload efficiency in both of the centralized and decentralized dispatchers. The efficiency (E) is computed as

$$E = \frac{task\_length * tasks\_per\_core * num\_cores}{time\_to\_solution * tasks\_per\_core * num\_cores} \quad (1)$$

Figs. 8 and 9 show how long a task should be in order to achieve a certain efficiency, ignoring data transfer at this point. Centralized dispatching performs better at small core counts here, but in order to efficiently scale up, decentralized

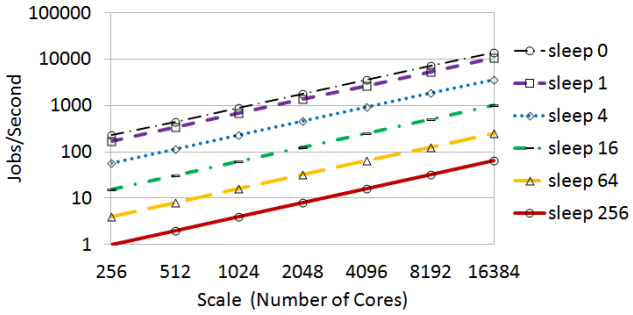

Fig. 7. Dispatching rate of decentralized dispatcher

dispatching is needed. For example, to achieve 90% efficiency at 8,192 cores, tasks run by the centralized dispatcher need to be at least 16 seconds long, but with the decentralized dispatcher, tasks only need to be 4 seconds long. The decentralized dispatcher allows domain scientists more flexibility by permitting them to use shorter tasks in their MTC applications.

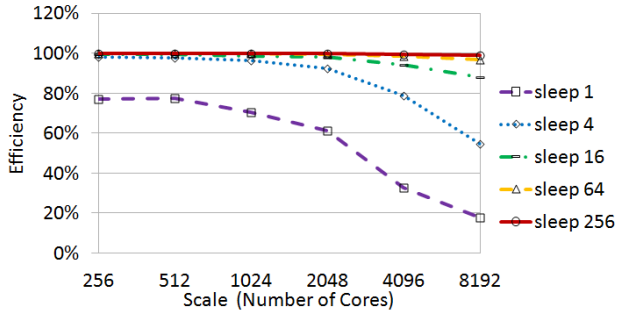

Fig. 8. Efficiency of centralized dispatcher

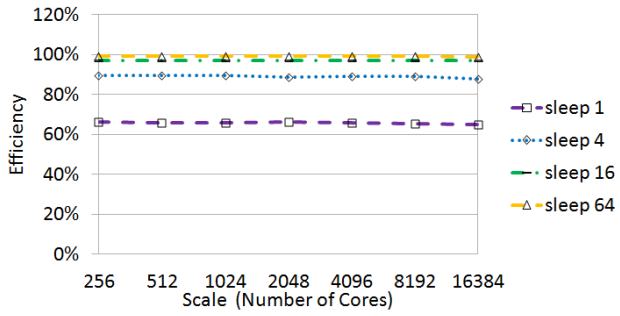

Fig. 9. Efficiency of decentralized dispatcher

B. Dispatching and transferring data

Next, we try to verify the linear scalability of the proposed intermediate data management scheme, evaluate how the file size affect the scheme.

1) Scalability: We use the same settings as the test suite in the above section with only one difference: we introduce task dependencies. Rather than sending 16 independent tasks to each core, we send eight pairs of tasks, each pair containing a task dependency. The first task in each pair runs for the task length, and outputs a 10-byte file. The second task in each pair takes the file from first task as input, then runs for the same task length. We use the decentralized dispatcher with the DDLLS to conduct the tests. The tests are set up so that every pair of tasks runs on two separate nodes, meaning that satisfying the dependencies always requires a file transfer.

Fig. 10 shows the time-to-solution vs. the task length for various numbers of cores. Though some overhead is introduced by the intermediate data handling scheme, it remains almost constant up as the number of cores increases to 8,192 due to the consistent hashing scheme, as shown in Fig. 11. At 16,384 cores, there is a significant increase for the task lengths of 1 and 4 seconds. This is because the peer data transfer on the interconnect network takes a longer time as the scale increases, and shorter tasks cause more temporal hot spots in the DDLLS servers. Overhead is computed as the difference between the time-to-solution of the dispatching without data transfer test (performed in §V-A) and the dispatching with data transfer test (performed here). Note that the intermediate data management overhead decreases as the task length increases because longer running tasks better spread traffic to the DDLLS servers over time, preventing temporal hot spots.

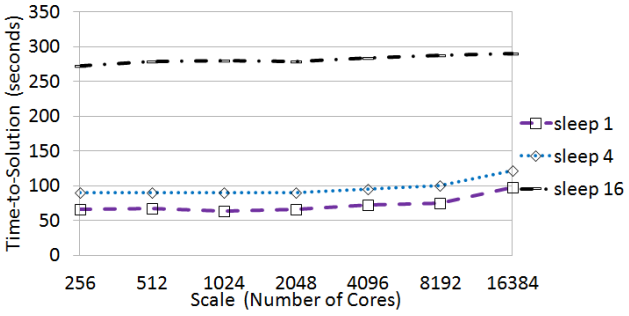

Fig. 10. Time-to-solution with intermediate data

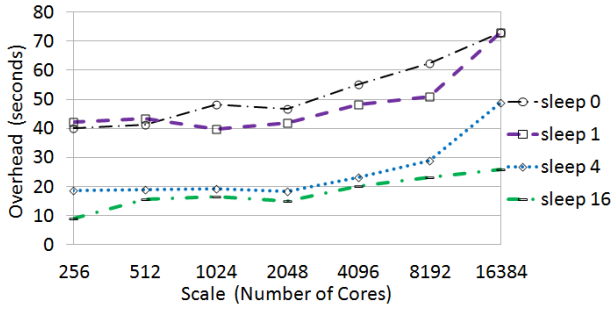

Fig. 11. Overhead introduced by intermediate data handling scheme

2) File Size Impact: We next examine the impact of varying the file size, running four pairs of 16-second tasks with data dependencies between the pairs as in the previous test. We sweep over two parameters: number of cores and file size. In each experiment, these files are either 1 KB, 1 MB, or 10 MB. The tests are again set up so that every pair of tasks runs on two separate nodes, meaning that satisfying the dependencies

always requires a file transfer. In this workload, each compute node has 16 cached files that are produced by its four cores, and another 16 files transferred from other compute nodes. The ideal (no overhead) time-to-solution of this test would be 128 seconds. There are two sources of overhead: task dispatching and intermediate data management. We see two trends in Fig. 12. One is that for a given number of cores, larger file sizes have more overhead, between 0.14% and 0.49% going from 1 KB to 1 MB files, and between 1.1% and 3.1% going from 1 KB to 10 MB files. The other trend is that for the same size files, using more cores has more overhead. The overhead comes from the file transfer over the torus network; more cores mean that file transfers have to take more hops across the network. From 256 cores to 8,192 cores, the increase in data transfer overhead is 5.5%, 5.8%, and 6.4%, for file sizes of 1 KB, 1 MB, and 10 MB respectively.

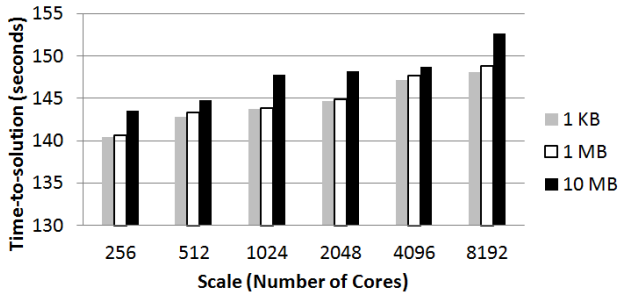

Fig. 12. Impact of file size time-to-solution

C. Collocated vs. Isolated Data Storage and Processing

This test verifies the scalability of both collocated and isolated data storage and processing. Fig. 13 shows the overhead of the two schemes. We used the same settings and workloads as §V-B, and we ran each test 5 times, computing the overhead by subtracting the ideal time-to-solution from the average. Generally, the collocated scheme performs better than the isolated scheme in terms of time-to-solution, as was previously discussed. (One might naïvely assume the overhead of the isolated scheme would be as twice that of the collocated scheme, as it involves two data movements as opposed to one. However, it is not, as the data transfer overlaps the tasks execution on the workers.) It is possible that the isolated scheme would have lower overhead than the collocated scheme if the compute nodes that were used to store the data were saturated by computation. A fuller comparison, which would also have to include any sacrifice of nodes to the intermediate storage system, is left for future work.

D. Data Transfer Overhead Analysis

Finally, we examine the overhead of the intermediate data management scheme. There are four potential sources: network congestion, DDLLS queuing, hash table synchronization in the DDLLS, and CPU-saturation of the OS. Workers access the DDLLS for two reasons: to *query* the state of some intermediate data, and to *update* the state of a piece of

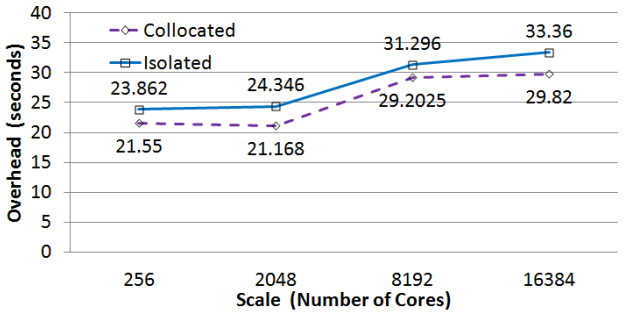

Fig. 13. Performance of collocated vs. isolated data storage and processing

data. Initially, each worker queries the input files for the second task in each pair, which have not yet been generated. The next eight rounds of data operations are updates. When the workers finish the first tasks in each pair, they update the state of the files they produce. Upon these updates, the DDLLS broadcasts the data locations to the workers who queried them. Then, workers copies the files from remote peers. In our test with 64 compute nodes (63 workers and 1 dispatcher and data manager), we run 4,032 tasks with 2,016 intermediate files. There are several periods of traffic congestion: in the first round of query traffic, 2016 queries arrive the DDLLS concurrently; once each round of tasks finishes, 252 updates arrive at the DDLLS concurrently; and upon receiving the location of the files, workers copy them across the network, with potential network congestion. These three potential congestion overlap or partially overlap, and cause overhead in the intermediate data management scheme.

Fig. 14 quantifies this congestion. The average time needed for a query is 148.4 ms (of which 144.3 ms is the queuing time at the DDLLS), while the average update operation takes 3.1 ms (of which 2.5 ms is the queuing time at the DDLLS). Upon receiving the intermediate data location, workers initiate transfers to remote peer to get the data. As Fig. 14 shows, the data transfer takes 0.2 s on average, with standard deviation of 0.24 s. The latency comes from the CPU-saturated OS. Comparing between the rounds of data transfer, later rounds are spread across a longer range of time, with a lower workload on the CPU, and thus lower data transfer latency. There are eight threads running actively in a worker on a quad-core Power CPU. Shorter tasks put a heavier load on thread switching on the CPU, producing a larger overhead.

VI. Application Experiments

Montage is an astronomy application that builds mosaics from a number of small images from telescope. It has been successfully run on supercomputers and grids, with MPI and Pegasus respectively [17]. The Pegasus version of the Montage workflow has nine stages, three of which involve steps that can be executed in parallel. In the AME version of Montage, we divide Montage into eight 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

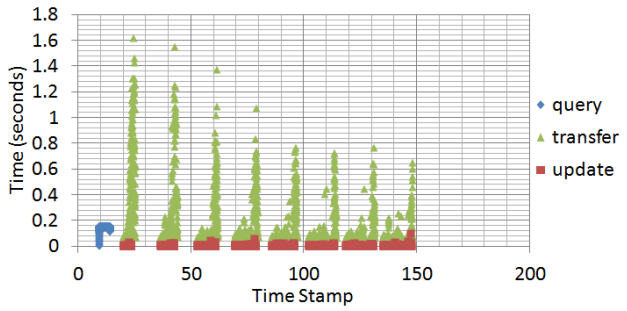

Fig. 14. End-to-end operation time on workers

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 task list of `mDiffFit` tasks (one for each pair of overlapping images). The fourth stage, `mDiffFit`, has tasks that take as input two overlapping output files from Stage 1 and fit 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` task list for the next stage. The 7th stage of the current workflow is `mBackground`, which actually applies the rectification to the reprojected images. 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 last stage, `mAdd`, reads output files from `mBackground`, and writes an aggregated mosaic file, the size of which is close to the sum of the sizes of the input files. Because the combined size of the input and output files in this state exceeds the RAM capacity of a BG/P nodes, we run the Montage-provided version of `mAdd` on the data that AME stored in GPFS.

We ran a test of Montage that produces a 6 x 6 mosaic centered at galaxy M101. It has 1,319 input files, each of which is 2 MB. Stage 1 outputs 1,319 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 tasks. 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 task 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 I. The time is measured in seconds.

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

	# of tasks	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 I
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%
mAdd-input	5200	5200	0%
mAdd-output	3700	3700	0%
total	62500	16901.17	72.96%

TABLE II
COMPARISON OF DATA TRANSFER AMOUNT BETWEEN GPFS AND AME APPROACHES

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 II.

VII. 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 six gaps: resource provisioning, task dispatching, task dependency resolution, load balancing, data management and system resilience.

AME is a MTC Computing engine that is designed for ultrascale supercomputers, with the focus on scalability. Using the principle of avoiding a central point, AME's dispatchers dispatch tasks 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. Dispatching 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 16,384 cores.

AME emphasizes its scalability on ultrascale machines with all of its tasks dispatchers, data managers and load balancers. In the dispatching test on 16,384 cores, AME ran 262,144 tasks with variable task lengths. And in the data management

test on 16,384 cores, the total number of files managed was 131,072. With 10 MB per file, the total file size was 1.3 TB.

AME is successful at running the Montage workflow. The workflow that produces a 6x6 mosaic using 512 cores on BG/P handles 62.5 GB of data in total. AME reduces data movement between compute nodes and GPFS from 62.5 GB to 16.9 GB, and significantly improves the utilization of the allocation during the run. The current implementation of AME can only takes advantage of tasks with I/O that is small enough to be be done in RAM. In addition to using a shared disk storage system, an aggregated shared intermediate file system (e.g., MosaStore) could be used to remedy this issue.

VIII. Future Work

To benefit from data locality, we will determine the advantages and disadvantages of routing tasks to data rather than moving data to tasks by leveraging the existing implementation of the DDLLS. This work is underway.

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

Automatically integrating the engine with existing parallel scripting language such as 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.

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

Finally, we will collaborate with additional domain scientists to run more MTC applications with AME, in order to gain more real-world understanding of AME's performance and utility.

Acknowledgments

This work was partially supported by the U.S. Department of Energy under the ASCR X-Stack program (contract DE-SC0005380) and under contract DE-AC02-06CH11357. Computing resources were provided by the Argonne Leadership Computing Facility. 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 goes to Professor Rick Stevens of the Dept. of Computer Science, U. Chicago for his enlightening class.

References

[1] J. M. Wozniak and M. Wilde, "Case studies in storage access by loosely coupled petascale applications," in *Proc. 4th Annual Workshop on Petascale Data Storage*, 2009, pp. 16–20.

[2] M. Wilde, I. Foster, K. Iskra, P. Beckman, Z. Zhang, A. Espinosa, M. Hategan, B. Clifford, and I. Raicu, "Parallel scripting for applications at the petascale and beyond," *Computer*, vol. 42, pp. 50–60, 2009.

[3] Z. Zhang, A. Espinosa, K. Iskra, I. Raicu, I. Foster, and M. 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*, 2008, pp. 1–10.

[4] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, and M. Wilde, "Falkon: a Fast and Light-weight tasK executiON framework," in *Proc. IEEE/ACM Supercomputing 2007*, 2007, pp. 1–12.

[5] J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke, "Condor-G: A computation management agent for multi-institutional grids," *Cluster Computing*, vol. 5, pp. 237–246, 2002.

[6] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster, "Swift: A language for distributed parallel scripting," *Parallel Computing*, pp. 633–652, September 2011.

[7] K. Iskra, J. W. Romein, K. Yoshii, and P. Beckman, "ZOID: I/O-forwarding infrastructure for petascale architectures," in *Proc. of 13th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, ser. PPoPP '08. New York, NY, USA: ACM, 2008, pp. 153–162.

[8] F. Schmuck and R. Haskin, "GPFS: A shared-disk file system for large computing clusters," in *In Proceedings of the 2002 Conference on File and Storage Technologies FAST*, 2002, pp. 231–244.

[9] S. Donovan, G. Huizenga, A. J. Hutton, A. J. Hutton, C. C. Ross, C. C. Ross, L. Symposium, L. Symposium, L. Symposium, M. K. Petersen, W. O. Source, and P. Schwan, "Lustre: Building a file system for 1,000-node clusters," 2003.

[10] P. H. Carns, W. B. Ligon, III, R. B. Ross, and R. Thakur, "PVFS: a parallel file system for linux clusters," in *Proceedings of the 4th annual Linux Showcase & Conference - Volume 4*. Berkeley, CA, USA: USENIX Association, 2000, pp. 28–28.

[11] B. Allcock, J. Bester, J. Bresnahan, A. L. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel, and S. Tuecke, "Data management and transfer in high-performance computational grid environments," *Parallel Comput.*, vol. 28, pp. 749–771, May 2002.

[12] S. Al-Kiswany, A. Gharaibeh, and M. Ripeanu, "The case for a versatile storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, pp. 10–14, March 2010.

[13] D. Thain, C. Moretti, and J. Hemmes, "Chirp: a practical global filesystem for cluster and grid computing," *Journal of Grid Computing*, vol. 7, no. 1, pp. 51–72, 2009.

[14] R. Thakur, W. Gropp, and E. Lusk, "Data sieving and collective I/O in ROMIO," *Symp. on Frontiers of Massively Par. Proc.*, p. 182, 1999.

[15] D. Borthakur, "HDFS architecture," http://hadoop.apache.org/common/docs/r0.20.0/hdfs_design.pdf.

[16] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proceedings of the ACM SIGCOMM '01 Conference*, August 2001.

[17] D. S. Katz, J. C. Jacob, G. B. Berriman, J. Good, A. C. Laity, E. Deelman, C. Kesselman, and G. Singh, "A comparison of two methods for building astronomical image mosaics on a grid," in *Proc. 2005 Intl. Conf. on Parallel Proc. Workshops*, 2005, pp. 85–94.