Pipelining/Overlapping Data Transfer for Distributed Data-Intensive Job Execution

Eun-Sung Jung, Ketan Maheshwari, Rajkumar Kettimuthu
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439
Email: {esjung, ketan, kettimut}@mcs.anl.gov

***Abstract*—Scientific workflows are increasingly gaining attention as both data and compute resources are getting bigger, heterogeneous, and distributed. Many scientific workflows are both compute intensive and data intensive and use distributed resources. This situation poses significant challenges in terms of real-time remote analysis and dissemination of massive datasets to scientists across the community. These challenges will be exacerbated in the exascale era.**

Parallel jobs in scientific workflows are common, and such parallelism can be exploited by scheduling parallel jobs among multiple execution sites for enhanced performance. Previous scheduling algorithms such as heterogeneous earliest finish time (HEFT) did not focus on scheduling thousands of jobs often seen in contemporary applications. Some techniques, such as task clustering, have been proposed to reduce the overhead of scheduling a large number of jobs. However, scheduling massively parallel jobs in distributed environments poses new challenges as data movement becomes a nontrivial factor.

We propose efficient parallel execution models through pipelined execution of data transfer, incorporating network bandwidth and reserved resources at an execution site. We formally analyze those models and suggest the best model with the optimal degree of parallelism. We implement our model in the Swift parallel scripting paradigm using GridFTP. Experiments on real distributed computing resources show that our model with optimal degrees of parallelism outperform the current parallel execution model by as much as 50% reduction of total execution time.

I. Introduction

Scientific workflows have gained prominence as tools of choice for running multistaged computations on large, heterogeneous, and distributed resources. Many scientific workflows [1], [2] are both compute intensive and data intensive and use distributed resources. This situation poses significant challenges in terms of real-time remote analysis and dissemination of massive datasets to scientists across the community. These challenges will be exacerbated in the exascale era.

Parallelism is a common occurence in various scientific workflow patterns [3]. This parallelism can be classified into two broad classes:

1) **Workflow parallelism**, which occurs because of the presence of independent branches in a workflow.
2) **Data parallelism**, which occurs because of a need to execute a workflow repeatedly for multiple datasets.

Such parallelism can be exploited by distributing parallel jobs among multiple execution sites for enhanced performance. Previous scheduling algorithms such as heterogeneous earliest finish time (HEFT) [4] did not focus on scheduling parallel jobs that can amount to a few thousand. Some techniques, such as task clustering [5], have been proposed to reduce the overhead of scheduling a large number of jobs.

Despite these opportunities of parallelism, scheduling massively parallel jobs in distributed environments poses new challenges as data movement becomes a nontrivial factor. Distributing parallel jobs among execution sites depending on their computing capacities is not enough for efficient job execution. A job can be naturally divided into three steps: stage-in, execution, and stage-out. Stage-in and stage-out steps refers to the input and output of data at an execution site. This involves using data transfer mechanisms such as sockets over TCP, specialized tool like GridFTP [6], etc. A simple policy of blocking job execution until data for all jobs in an execution step are available can lead to performance degradation.

In this paper, we propose efficient parallel execution models through pipelined execution of data transfer via dedicated channels overlapped with execution, incorporating better utilization of network bandwidths and reserved resources at an execution site. We formally analyze these models and suggest the best model with an improved exploitation of parallelism. Consequently, the paper makes the following contributions:

1) Review and theoretical analysis of various pipelined execution models for wide-area distributed computing. They were originally designed for tightly coupled architectures such as vector and superscalar machines.
2) Decompose traditional staging and execution cycles of application tasks into distinct jobs to exploit parallelism by overlapping those cycles in a pipeline.
3) Implement and test the above method via Swift parallel scritping framework [7] on wide-area distributed resources enabled with a third-party GridFTP data transfers.

The remainder of the paper is structured as follows. In Section II, we briefly present background on distributed parallel computing. In Section III, we propose several parallel job execution models and present mathematical analysis of those models and corresponding optimal degrees of parallelism. In Section IV, we evaluate our analytical models with actual implementation in the Swift parallel scripting paradigm with

third-party GridFTP transfers for data staging. In Section V, we briefly summarize our work.

II. Background

In this section, we briefly present parallelizing techniques in the literature.

A. Parallel Computing in Local Machines

Parallel computing on single, large machines has been an active research area since the 1990s [8]. computer architectures can be categorized into four areas: (1) temporal parallelism: pipelining, (2) data parallelism: SIMD, (3) functional parallelism: MIMD, and (4) instruction level parallelism: superscalar, VLIW. In case of applications, several parallel programming models, such as the message-passing model implemented and popularized by MPI and OpenMP standards, have become standard [9].

In order to apply these techniques for distributed applications, additional constraints such as wide-area network communications among distributed processes should be considered.

B. Distributed Parallel Computing

Distributed computing platforms are deployed to run and manage scientific workflows efficiently. Popular workflow management systems include Condor DAGMan [10], Pegasus [11], Taverna [12], and makeflow [13]. Regarding massively parallel job scheduling, Pegasus [11] provides a task clustering technique that groups parallel jobs based on fixed parameters such as number of groups. This static approach has limitations such as ignorance of job properties (e.g., compute-/data-intensive job). Some approaches have attempted to address the limitations through dynamic task clustering. One example is the Falkon project [14], which dynamically generates workflows and clusters tasks for compute resource sites.

As connection-oriented networks such as ESnet [15] and Internet2 [16] started to provide in-advance network path reservation through OSCARS [17] and as the amount of data is expected to grow rapidly in the near future, data movement and placement become important factors in workflow scheduling algorithms. Accordingly, recent workflow scheduling algorithms focus not only on optimal compute resource allocations but also on efficient data movement [18].

To the best of our knowledge, however, distributed parallel job scheduling has not been tackled in terms of optimization of data transfer. Especially when multiple parallel jobs, each of which needs to receive/send data to/from remote sites, are mapped to a single site, overlapping and pipelined data transfer can significantly improve overall performance. In the following sections, we propose and analyze parallel execution models enabling data transfer to be overlapped with computation.

III. Distributed Massively Parallel Job Scheduling

In this section, we first describe challenges in distributed massively parallel job scheduling. We then formally analyze the parallel execution models and propose optimal co-scheduling algorithms, taking into account both compute and network resources.

A. Problem Statement

Scheduling massively parallel jobs in wide-area, federated distributed high-performance computing (HPC) systems such as XSEDE [19] is not trivial for several reasons. First, distributed HPC systems have diverse capability, and users have different resource allocations on the systems depending on subscribed projects. Moreover, each HPC system is managed by different local resource managers, such as MAUI [20] and SLURM [21], which makes job scheduled on the system go through different internal scheduling policies. With data-intensive workflows, the time required for data movement for jobs becomes more important compared with the CPU time required for running the jobs. Data movement could involve network topologies in a single or multiple domains, and concurrent data movements should be carefully coordinated because network bottlenecks are largely influenced by network topologies. The network resource scheduling efforts may be quite different depending on network properties, packet-oriented vs. circuit-oriented networks.

In the following sections, we first formalize the scheduling problem for a single HPC site where a user has a limited amount of allocation and multiple parallel jobs are given. We show that the solution to the problem is the optimal degree of parallelism, which means how many jobs should be executed in parallel considering compute and network resources. In future, we will extend the scheduling solution to a workflow consisting of many jobs targetted to multiple compute resource sites.

B. Theoretical Analysis of Execution Time

In this section, we theoretically analyze the performance of distributed parallel tasks. We start by describing basic well-known performance equations at the computer instruction level [8]. We then develop analytical performance equations for distributed parallel tasks.

In Equation 1, $T_{inst}(m, p)$ is the number of base cycles to complete tasks in the instruction level when all instructions take one base cycle, k is the length of one pipeline iteration, m is the number of simultaneous instruction issues, p is the degree of parallelism in each pipeline stage, and n is the number of iterations. The parameter p means that each instruction can be further pipelined into p subpipeline stages. A machine is called a *superscalar machine* if $m > 1$; a machine is called a *superpipelined machine* when $p > 1$.

$$T_{inst}(m, p) = k + \frac{n - m}{m \times p} \tag{1}$$

k : Lengh of one pipeline iteration in base cycles

m : Number of simultaneous instruction issues

p : Degree of parallelism in each pipeline stage

n : Number of iterations

In Equation 2, S_{inst} is speedup compared with sequential execution that requires nk base cycles.

$$S_{inst} = \frac{nk}{T_{inst}(m,p)} \tag{2}$$

At the task level, when instructions are substituted by tasks of variable lengths and $p = 1$, the previous equations can be extended as in Equation 3. If the execution times of pipeline tasks vary, the execution time of one pipeline stage is dominated by the maximum execution time of pipeline tasks. Since $T_{task}(m,1)$ becomes close to $\frac{n}{m} \times t_{max}$ as n increases, the speedup for task-level pipelining can be expressed as in Equation 4 if the times of SI, EX, SO are similar.

$$T_{task}(m,1) = (k + \frac{n-m}{m}) \times t_{max} \tag{3}$$

k : Lengh of one pipeline iteration in number of tasks

m : Number of simultaneous task issues

n : Number of pipeline iterations

t_{max} : Maximum execution time of pipeline tasks.

$$S_{task} = \frac{n \times t_{total}}{T_{task}(m,1)} \simeq mk \tag{4}$$

t_{total} : Total execution time of tasks without pipelining

Based on Equations 3 and 4, we can formally analyze the performance of jobs dispatched to one compute resource site when data movement and computation are overlapped. For example, if four jobs are scheduled on a cluster *A* consisting of 2 nodes and each job is required to occupy the whole node, the data for the third and fourth jobs can be transferred while the first and second jobs are running. This approach is implementable by using GridFTP and associated data transfer nodes dedicated for data transfers. The dedicated data transfer node model is used by many compute facilities including the national cyberinfrastructure such as XSEDE and this is also the basic model of Science DMZ [22]. We categorize overlapping of data movement and computation into three classes: simple pipelined execution, multiple control-flow based execution, and superscalar-style execution. In the following, we describe each class in more details.

1) Simple pipelined execution (SPE): Without loss of generality, we can assume that a job is composed of three tasks or stages, stage-in (SI), execution (EX), and stage-out (SO). In the SI stage, the input data for the job is transferred from the precedent job sites. In the EX stage, the job runs with the input data and produces output data for descendent jobs. In the SO stage, the output data are transferred to the descendent job sites or to the submit site if this is the last job in the workflow. The simple pipelined execution is the simplest of parallel execution cases, in which the SI, EX, and SO stages are executed in a pipeline as in Figure 1. The execution time(SPE) and speedup(SPE) for this model are analyzed in Equations 5 and 6, respectively.

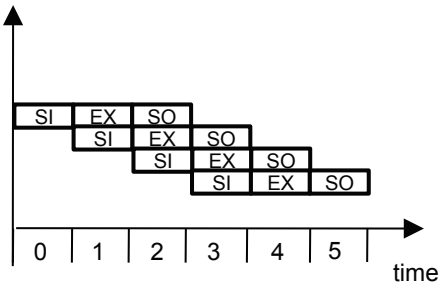

Fig. 1. Simple pipelined execution.

$$T_{SPE}(1,1) = (k + (n-1)) \times t_{max} \tag{5}$$

$$S_{SPE} = \frac{n \times t_{total}}{T_{SPE}(1,1)} \simeq k \tag{6}$$

One implementation of this model is that, in the EX stage, the job calls a nonblocking data receive function for input data required by the next job and a nonblocking data send function for output data produced by itself at the start and end of the EX stage, respectively.

2) Multiple control-flow-based execution (MCE): Multiple control-flow-based execution (MCE) is a model of a fixed number of independent control flows/threads executing SI, EX, and SO repeatedly. For instance, Figure 2 shows that four threads are pooled for multiple job execution, and their progress is synchronized at best. They may not be synchronized, however, depending on the operating system's scheduling policy. The typical example of this model is the Swift parallel scripting paradigm. The job throttle parameter in Swift takes control of the number of concurrent job executions. In the best case, the execution time(MCE) and speedup(MCE) for this model are analyzed in Equations 7 and 8, respectively.

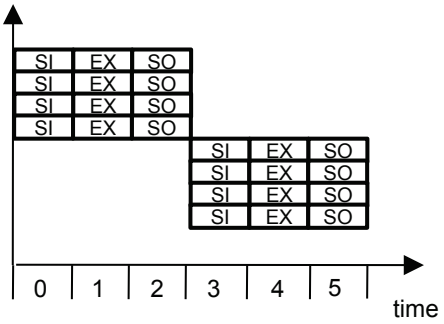

Fig. 2. Multiple control-flow based execution.

$$T_{MCE} = (k \cdot \lceil \frac{n}{m} \rceil) \times t_{max} \tag{7}$$

$$S_{MCE} = \frac{n \times t_{total}}{T_{MCE}} \simeq m \tag{8}$$

Then how do we determine optimal m for the system? There should be enough compute resources to run m EX stages. If not, the time required for m EX stages may be more than t_{max}, which leads to prolonged T_{MCE}. For instance, if two CPU cores are available and three jobs are in the EX stage, two jobs will first run simultaneously and the third job will wait until those jobs are completed. Consequently, the total required time would be two times of the one job execution time.

3) Superscalar-style execution (SSE): The superscalar-type execution (SSE) model is a hybrid of the SPE model and the MCE model, in which m jobs are in the same stage and each stage is overlapped with other stages. Figure 3 shows an example of SSE, and equations 9 and 10 are the required time(SSE) and speedup(SSE), respectively, for the SSE model. In this model, m is determined in the same way as in MCE model.

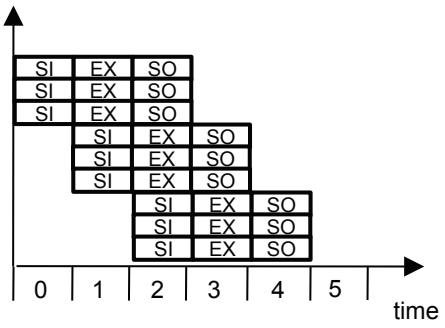

Fig. 3. Superscalar-style execution.

$$T_{SSE}(m,1) = (k + \frac{n-m}{m}) \times t_{max} \tag{9}$$

$$S_{SSE} = \frac{n \times t_{total}}{T(m,1)} \simeq m \times \frac{t_{total}}{t_{max}} \simeq mk \tag{10}$$

C. Coscheduling with networks

When one compute resource has r computing nodes and unlimited I/O capacity is assumed, the model for the best performance is a SSE model with $m = r$ whose speedup are $3r$ and 3 compared with the sequential execution model and the MCE model, respectively. However, the network bandwidth is not unlimited, and the times for the SI and SO stages vary according to the network status and m. Therefore, we define the time required for pipeline stages as functions of m, where $m = 0$ means a sequential execution. Table I shows a list of time functions for pipeline execution.

With the newly defined functions, Equation 10 can be rewritten as Equation 11.

$$S_{SSE} \simeq m \times \frac{t_{total}(0)}{t_{max}(m)} = m \times \frac{t_{SI}(0) + t_{EX}(0) + t_{SO}(0)}{\max\{t_{SI}(m), t_{EX}(m), t_{SO}(m)\}} \tag{11}$$

TABLE I
TIME FUNCTION LIST

Function	Description
$t_{SI}(m)$	Time required for the SI stage
$t_{EX}(m)$	Time required for the EX stage
$t_{SO}(m)$	Time required for the SO stage
$t_{total}(m)$	Time required for all sequential pipeline stages; $t_{SI}(m) + t_{EX}(m) + t_{SO}(m)$
$t_{max}(m)$	Maximum of $t_{SI}(m)$, $t_{EX}(m)$, and $t_{SO}(m)$

For simplicity, we assume that the network bandwidth, B_{shared}, is shared by both the SI and SO stages and that the EX stage is not interfered with the SI/SO stages. In reality, the problem may get more complicated when a complex network topology is involved and network paths used for SI and SO may or may not overlap at our discretion.

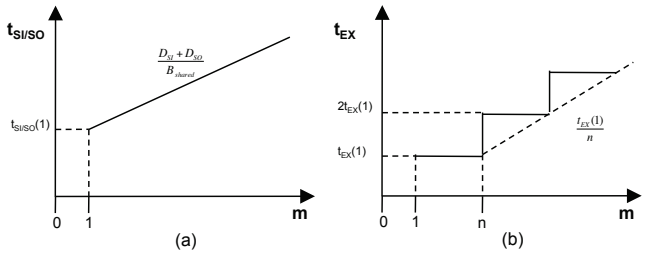

Fig. 4. (a) $t_{SI/SO}(m)$ plot, (b) $t_{EX}(m)$ plot

With Equation 11, our goal is to find the optimal m that maximizes the speedup S_{SSE}. Since the SI and SO stages are overlapped and the B_{shared} bandwidth is also shared by both data transfers, Equation 12 holds true.

$$t_{SI}(m) = t_{SO}(m) = m \times \frac{(D_{SI} + D_{SO})}{B_{shared}} \tag{12}$$

D_{SI} : Amount of data transferred in SI stages

D_{SO} : Amount of data transferred in SO stages

Therefore, we can substitute $\max\{t_{SI}(m), t_{EX}(m), t_{SO}(m)\}$ by $\max\{m \times \frac{(D_{SI}+D_{SO})}{B_{shared}}, t_{EX}(m)\}$. The resulting equation is Equation 13.

$$S_{SSE} = m \times \frac{t_{total}(0)}{\max\{m \times \frac{(D_{SI}+D_{SO})}{B_{shared}}, t_{EX}(m)\}} \tag{13}$$

Equation 13 can be further analyzed through the denominator max function. Figure 4 shows the plots of two terms, $m \times \frac{(D_{SI}+D_{SO})}{B_{shared}} (= t_{SI/SO}(m))$ and $t_{EX}(m)$, in the max function.

Figure 5 shows the possible four cases of the max function depending on the relative positions of two functions, $t_{SI/SO}(m)$ and $t_{EX}(m)$. We can find optimal m as follows.

a) **Data-intensive case** $(\frac{D_{SI}+D_{SO}}{B_{shared}} > \frac{t_{EX}(1)}{r}) \wedge (t_{SI/SO}(1) > t_{EX}(1)) \equiv (\frac{D_{SI}+D_{SO}}{B_{shared}} > t_{EX}(1))$*:* The speedup of this case is constant regardless of m; we can pick

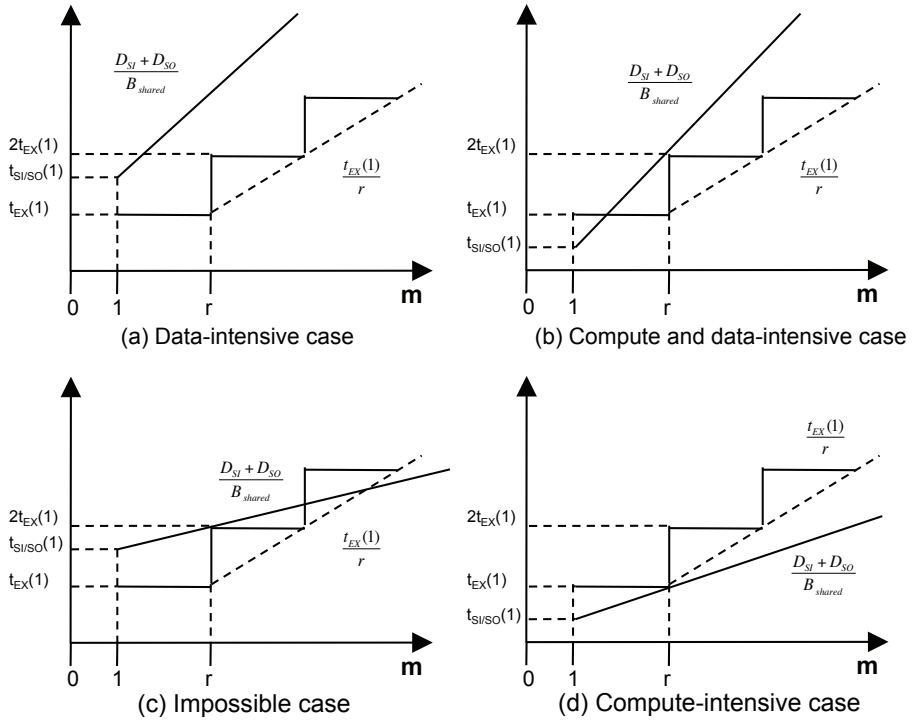

Fig. 5. (a) Data-intensive case (b) Compute and data-intensive case (c) Impossible case (d) Compute-intensive case.

$m = 1$, which requires the simplest implementation.

$$max\{\cdot\} = m \times \frac{D_{SI} + D_{SO}}{B_{shared}}, m > 0$$
$$S_{SSE} = \frac{t_{total}(0) \times B_{shared}}{D_{SI} + D_{SO}}$$
$$Optimal\ m = 1 \quad (14)$$

b) **Compute and data-intensive case** $(\frac{D_{SI}+D_{SO}}{B_{shared}} > \frac{t_{EX}(1)}{r}) \wedge (t_{SI/SO}(1) < t_{EX}(1)) \equiv (\frac{t_{EX}(1)}{r} < \frac{D_{SI}+D_{SO}}{B_{shared}} < t_{EX}(1))$*:* This case is further divided into two subcases: prior and posterior to the intersection of $t_{SI/SO}(m)$ and $t_{EX}(m)$. Optimal $m^{'}$s of both cases are same.

$$max\{\cdot\} = \begin{cases} t_{EX}(1), & \text{if } 1 < m \leq \frac{t_{EX}(1) \times B_{shared}}{D_{SI}+D_{SO}} \\ \frac{D_{SI}+D_{SO}}{B_{shared}} \times m, & \text{if } \frac{t_{EX}(1) \times B_{shared}}{D_{SI}+D_{SO}} \leq m < r \end{cases}$$
$$S_{SSE} = \begin{cases} \frac{m \times t_{total}(0)}{t_{EX}(1)}, & \text{if } 1 < m \leq \frac{t_{EX}(1) \times B_{shared}}{D_{SI}+D_{SO}} \\ \frac{B_{shared} \times t_{total}(0)}{D_{SI}+D_{SO}}, & \text{if } \frac{t_{EX}(1) \times B_{shared}}{D_{SI}+D_{SO}} \leq m < r \end{cases}$$
$$Optimal\ m = \frac{t_{EX}(1) \times B_{shared}}{D_{SI} + D_{SO}} \quad (15)$$

c) **Impossible case** $(\frac{D_{SI}+D_{SO}}{B_{shared}} < \frac{t_{EX}(1)}{r}) \wedge (t_{SI/SO}(1) > t_{EX}(1)) \equiv (t_{EX}(1) < \frac{D_{SI}+D_{SO}}{B_{shared}} < \frac{t_{EX}(1)}{r})$*:* This is an impossible case.

d) **Compute-intensive case** $(\frac{D_{SI}+D_{SO}}{B_{shared}} < \frac{t_{EX}(1)}{r}) \wedge (t_{SI/SO}(1) < t_{EX}(1)) \equiv (\frac{D_{SI}+D_{SO}}{B_{shared}} < \frac{t_{EX}(1)}{r})$*:* The speedup of this case increases as m grows. However, m is limited by $\lceil n/3 \rceil$, which guarantees pipelined execution at least.

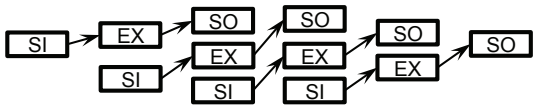

Fig. 6. A data-flow view of the pipeline; the jobs are shown in boxes and data dependencies as arrows, the jobs in non-connected boxes are independent and can be executed in parallel.

$$max\{\cdot\} \simeq \frac{(m + n - 1)t_{EX}(1)}{r}, m > 0$$
$$S_{SSE} = \frac{mn \times t_{total}(0)}{(m + n - 1)t_{EX}(1)}$$
$$Optimal\ m = \lceil n/3 \rceil, \text{ where } n \text{ is the number of jobs.} \quad (16)$$

IV. Experimental Evaluations

An alternative, data-flow view of the pipeline shown in figure 1 can be as shown in figure 6. Jobs are shown as boxes and arrows between boxes show a data dependency between these jobs. Each triplet of SI-EX-SO forms one logical task. This view of decomposed tasks into three jobs gives a maximum oppurtunity for parallel and overlapped execution assuming each task in this pipeline are independent, either by the virtue of data parallelism or workflow parallelism discussed in Section I.

The data-flow view depicted in Figure 6 above is implemented by coding it using the Swift parallel scripting frame-

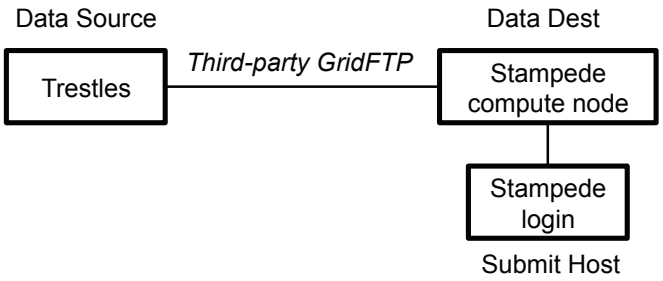

Fig. 7. Experimental setup on XSEDE resources; Data is hosted at SDSC Trestles and consumed at TACC Stampede; application is submitted via Swift from Stampede login nodes; jobs run on Stampede compute nodes.

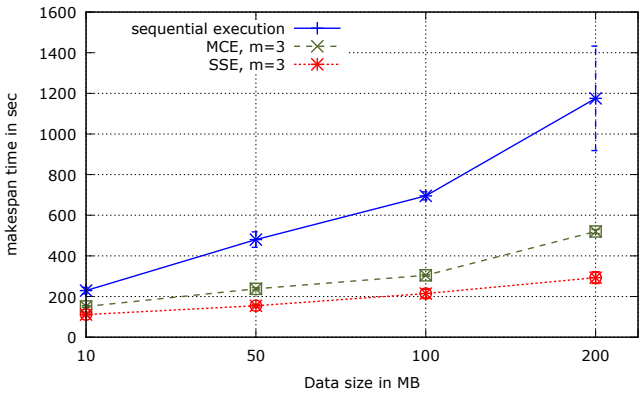

Fig. 8. Performance results: A 10-task application running in pipeline with different staging loads under different models of pipelined execution

work. The programming model of Swift allows for implicit parallelism governed and steered via data-flow constraints. The underlying Karajan [23] engine and the Coasters [24] framework of Swift provides a built-in data staging mechanism using socket-based TCP transfers. However, in the current implementation, we bypass the builtin staging for third-party data transfer via GridFTP motivated by the following factors:

1) GridFTP servers used for third-party data staging are performant and dedicated for data movement resulting in faster data transfers compared to regular TCP channels.
2) Detaching the data staging steps from computation allows for overlapped data transfer and execution which in turn improves the overall execution time of application.

```
1  import "header";
2
3  int n=4; //number of tasks
4
5  #first stage
6  token[0] = stage_in (dstore[0], local[0]);
7
8  #second stage
9  res[0] = exec (local[0], token[0]);
10 token[1]=stage_in (dstore[1], local[1]);
11
12 foreach i in [2:n-1]{
13  dstore_out[i-2] = stage_out (res[i-2]);
14  res[i-1] = exec (local[i-1], token[i-1]);
15  token[i] = stage_in (dstore[i], local[i]);
16 }
17
18 #second from last stage
19 dstore_out[n-2] = stage_out (res[n-2]);
20 res[n-1] = exec (local[n-1], token[n-1]);
21
22 #last stage
23 dstore_out[n-1] = stage_out (res[n-1]);
```

Listing 1. Swift script of our model for parallel execution of pipelined workflow tasks

A Swift script implementation for the pipeline is shown in listing 1. Swift variables and function definitions are defined in the 'imported' `header` (line 1) file. The first (lines 6, 9-10) and last two (lines 19-20, 23) stages of the pipeline are represented by the lines before and after the `foreach` loop. The `foreach` (lines 12-16) loop represents the jobs for each of the task. A 'token' variable is used for synchronization between `stage_in`, `exec` and `stage_out` jobs. Array indices makes sure that the same tasks are chained by the dependencies while still running in fully asynchronous execution mode.

The script shown in the listing 1 was executed from the login node of the Stampede supercomputer located at Texas. The data was hosted on the data storage node of remotely located SanDiego Supercomputer Center's (SDSC) Trestles system. A complete setup is shown in figure 7. Figure 8 shows performance results for our approach with different degrees of parallelism (controlled via Swift throttle parameter) and an increasing number of data sizes. The execution time of the jobs were adjusted in proportion to the average data movement times from the staging jobs. The plot shows resulting execution times averaged over 5 runs with standard deviation recorded. A basic run with sequential pipeline execution shows a baseline result as shown by the blue (solid) plot. As expected, the performance is poor since each task is run in three sequential stages. A clearly improved performance is seen when running the jobs in parallel pipelined fashion as shown by the green (dashed) plot corresponding to the MCE model with $m = 3$, a parallelism of 3 jobs at a time. Finally, as seen by the red (dotted) plot corresponding the SSE model with $m = 3$, an improvement by 50% is observed when the parallelism is increased to 9 mimicking a superscalar pipeline execution. In our experiments, the results follows the models in Section III-B where the data transfer performance is linearly scalable as the number of data streams increases. The MCE model with with $m = 3$ is about 2.4 times faster than the sequential model, which is comparable to the ideal speedup with linearly scalable data transfer rates is 3. We suspect that this is either because a single data transfer is not saturating the network link or multiple alternate network paths exist between two hosts. In former case, we will tune data transfers to evaluate models taking into account network contention in Section III-C. In latter case, we will investigate more sophisticated parallel execution models to incorporated multiple network paths.

V. Conclusion

We propose efficient parallel execution models through pipelined execution of data transfer, incorporating network

bandwidth and reserved resources at an execution site. We formally analyze those models and suggest the best model with the optimal degree of parallelism. We also implement our model in the Swift parallel scripting paradigm using GridFTP. Experiments on real distributed computing resources show that our model with optimal degrees of parallelism outperform the current parallel execution model by as much as 50% reduction of total execution time.

ACKNOWLEDGMENT

This work was supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, under Contract DE-AC02-06CH11357.

REFERENCES

[1] H. B. Newman, M. H. Ellisman, and J. A. Orcutt, "Data-intensive e-science frontier research," *Commun. ACM*, vol. 46, no. 11, pp. 68–77, 2003. [Online]. Available: http://portal.acm.org/ft_gateway.cfm?id=948411&type=html&coll=Portal&dl=ACM&CFID=46158761&CFTOKEN=74773945

[2] E. Deelman, D. Gannon, M. Shields, and I. Taylor, "Workflows and e-science: An overview of workflow system features and capabilities," *Future Generation Computer Systems*, vol. 25, no. 5, pp. 528–540, May 2009. [Online]. Available: http://www.sciencedirect.com/science/article/B6V06-4SYCPKX-2/2/bb631979e3dd7071ddede90bbff65a91

[3] C. Pautasso and G. Alonso, "Parallel computing patterns for grid workflows," in *Workshop on Workflows in Support of Large-Scale Science, 2006. WORKS '06*, 2006, pp. 1–10.

[4] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, pp. 260–274, 2002.

[5] G. Singh, M.-H. Su, K. Vahi, E. Deelman, B. Berriman, J. Good, D. S. Katz, and G. Mehta, "Workflow task clustering for best effort systems with pegasus," in *Proceedings of the 15th ACM Mardi Gras conference*, ser. MG '08. New York: ACM, 2008, pp. 9:1–9:8. [Online]. Available: http://doi.acm.org/10.1145/1341811.1341822

[6] B. Allcock, J. Bresnahan, R. Kettimuthu, M. Link, C. Dumitrescu, I. Raicu, and I. Foster, "The Globus striped GridFTP framework and server," in *SC'2005*, 2005.

[7] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster, "Swift: A language for distributed parallel scripting," *Parallel Computing*, vol. 37, pp. 633–652, 2011.

[8] K. Hwang, *Advanced Computer Architecture - Parallelism, Scalability, Programmability*. McGraw-Hill, 1993.

[9] J. Diaz, C. Munoz-Caro, and A. Nino, "A survey of parallel programming models and tools in the multi and many-core era," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 8, pp. 1369–1386, 2012.

[10] P. Couvares, T. Kosar, A. Roy, J. Weber, and K. Wenger, "Workflow management in condor," in *Workflows for e-Science*, I. J. Taylor, E. Deelman, D. B. Gannon, and M. Shields, Eds. London: Springer London, 2007, pp. 357–375. [Online]. Available: http://www.springerlink.com/content/r6un6312103m47t5/

[11] J. Blythe, S. Jain, E. Deelman, Y. Gil, K. Vahi, A. Mandal, and K. Kennedy, "Task scheduling strategies for workflow-based applications in grids," in *IEEE International Symposium on Cluster Computing and the Grid, 2005. CCGrid 2005*, vol. 2, 2005, pp. 759–767 Vol. 2.

[12] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, and P. Li, "Taverna: a tool for the composition and enactment of bioinformatics workflows," *Bioinformatics*, vol. 20, no. 17, pp. 3045–3054, Nov. 2004. [Online]. Available: http://dx.doi.org/10.1093/bioinformatics/bth361

[13] M. Albrecht, P. Donnelly, P. Bui, and D. Thain, "Makeflow: a portable abstraction for data intensive computing on clusters, clouds, and grids," in *Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*, ser. SWEET '12. New York, NY, USA: ACM, 2012, p. 1:13. [Online]. Available: http://doi.acm.org/10.1145/2443416.2443417

[14] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, and M. Wilde, "Falkon: a fast and light-weight tasK executiON framework," in *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing, 2007. SC '07*, 2007, pp. 1–12.

[15] "ESnet," http://www.es.net/.

[16] "Internet2," http://www.internet2.edu/.

[17] C. Guok, D. Robertson, M. Thompson, J. Lee, B. Tierney, and W. Johnston, "Intra and interdomain circuit provisioning using the OSCARS reservation system," in *3rd International Conference on Broadband Communications, Networks, and Systems*, 2006.

[18] E.-S. Jung, S. Ranka, and S. Sahni, "Workflow scheduling in e-science networks," in *Computers and Communications (ISCC), 2011 IEEE Symposium on*, 2011, pp. 432–437.

[19] "Extreme science and engineering discovery environment (xsede)," `http://www.xsede.org/`.

[20] D. B. Jackson, Q. Snell, and M. J. Clement, "Core algorithms of the maui scheduler," in *Revised Papers from the 7th International Workshop on Job Scheduling Strategies for Parallel Processing*, ser. JSSPP '01. London: Springer-Verlag, 2001, p. 87?102. [Online]. Available: http://dl.acm.org/citation.cfm?id=646382.689682

[21] A. B. Yoo, M. A. Jette, and M. Grondona, "SLURM: Simple Linux Utility for Resource Management," in *Job Scheduling Strategies for Parallel Processing*, ser. Lecture Notes in Computer Science, D. Feitelson, L. Rudolph, and U. Schwiegelshohn, Eds. Springer Berlin Heidelberg, Jan. 2003, no. 2862, pp. 44–60.

[22] "Science DMZ," http://fasterdata.es.net/science-dmz/.

[23] Y. Zhao, I. Raicu, I. T. Foster, M. Hategan, V. Nefedova, and M. Wilde, "Realizing fast, scalable and reliable scientific computations in grid environments," *CoRR*, vol. abs/0808.3548, 2008.

[24] M. Hategan, J. Wozniak, and K. Maheshwari, "Coasters: uniform resource provisioning and access for scientific computing on clouds and grids," in *Proc. Utility and Cloud Computing*, 2011.