Performance Model for Parallel Matrix Multiplication with Dryad: Dataflow Graph Runtime

Hui Li, Geoffrey Fox, Judy Qiu
School of Informatics and Computing, Pervasive Technology Institute
Indiana University Bloomington
lihui@indiana.edu
gcf@indiana.edu
xqiu@indiana.edu

Abstract—**In order to meet the big data challenge of today's society, several parallel execution models on distributed memory architectures have been proposed: MapReduce, Iterative MapReduce, graph processing, and dataflow graph processing. Dryad is a distributed data-parallel execution engine that model program as dataflow graphs. In this paper, we evaluated the runtime and communication overhead of Dryad in realistic settings. We proposed a performance model for Dryad implementation of parallel matrix multiplication (PMM) and extend the model to MPI implementations. We conducted experimental analyses in order to verify the correctness of our analytic model on a Windows cluster with up to 400 cores, Azure with up to 100 instances, and Linux cluster with up to 100 nodes. The final results show that our analytic model produces accurate predictions within 5% of the measured results. We proved some cases that using average communication overhead to model performance of parallel matrix multiplication jobs on common HPC clusters is the practical approach.**

Keyword: Dryad, Dataflow Graph, Performance Modeling, MPI, Matrix Multiplication

I. MOTIVATION AND BACKGROUND

A data deluge exists in today's society. The rapid growth of information requires domain technologies and runtime tools to process huge amounts of data. In order to meet this big data challenge, several parallel execution models on distributed memory architectures have been proposed: MapReduce[1], Iterative MapReduce[2], graph processing[3], and dataflow graph processing [4, 5]. The MapReduce programming model has been applied to a wide range of applications and attracted enthusiasm from distributed computing communities due to its ease of use and efficiency in processing large scale distributed data. However, MapReduce has the limitations. For example, it is not efficient to process multiple, related heterogeneous datasets, and iterative applications. Paper [2] implemented a parallel runtime for iterative MapReduce applications that outperform Hadoop in performance by several orders of magnitude. Paper [6] found that Hadoop is not efficient when processing an RDF graph pattern match that requires the joining of multiple input data streams. Dryad is a data flow runtime that models application as data flow among processes or DAGs. Dryad supports relational algebra and can process relational un-structure and semi-structure data more efficiently than Hadoop.

Performance modeling of applications has been well studied in the HPC domain for decades. It is not only used to predicate the job running time for specific applications, but can also be used for profiling runtime environments. Among various runtime environments, the performance modeling of MPI on distributed memory architecture is well understood. The performance modeling approaches include: analytical modeling, empirical modeling, and semi-empirical modeling. Papers [7] and [8] investigated the analytical model of parallel programs implemented with MPI executed in a cluster of workstations. Paper [9] proposed a semi-empirical model for bioinformatics applications that was implemented using the hybrid parallel approach on a Windows HPC cluster. Paper [10] introduced a semi-experimental performance model for Hadoop. However, the performance analysis of parallel programs using data flow runtimes is relatively understudied. In fact, the performance impact of dataflow graph runtimes at massive scale is increasingly of concern. In addition, a growing concern exists about the power issue and cost effectiveness of the "pay-as-you-go" environment. We argue that the data flow graph runtime should also deliver efficiency for parallel programs. Thus, this study of the performance modeling of data flow graph runtimes is of practical value.

Several sources of runtime performance degradations exist, including [11]: latency, overhead, and communication. Latency is the time delay used to access remote data or services such as memory buffers or remote file pipes. Overhead is the critical path work required to manage parallel physical resources and concurrent abstract tasks. It can determine the scalability of a system and the minimum granularity of program tasks that can be effectively exploited. Communication is the process of exchanging data and information between processes. Previous studies [12] of application usage have shown that the performance of collective communications is critical to high performance computing (HPC). The difficulty of building analytical models of parallel programs of data flow runtimes is to identify the communication pattern and model communication overhead.

In this paper, we proposed an analytical timing model for Dryad implementation of PMM in different realistic settings which is more general than the settings used in the empirical and semi-empirical model. We extended the proposed analytical model to MS.MPI, and made comprehensive comparisons between Dryad and MPI implementations of the PMM applications. We conducted experimental analyses in order to verify the correctness of our analytic model on a Windows cluster with up to 400 cores, Azure with up to 100 instances, and Linux cluster with up to 100 nodes. The final results show that our analytic model produces accurate predictions within 5% of the measured results.

The remainder of this paper is organized as follows. An introduction to Dryad is briefly discussed in section 2, followed by an evaluation of the overhead of the Dryad runtime in section 3. In section 4, we present the analytical model for the Dryad and MPI implementations of the PMM application. Section 5 contains the experiments results of the proposed analytical model. Finally, we provided remarks and conclusions in section 6.

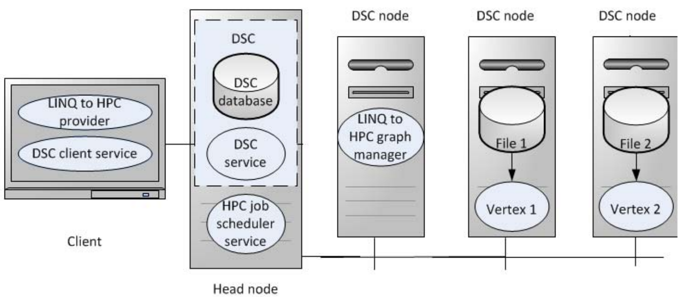


Figure 1: Dryad Architecture

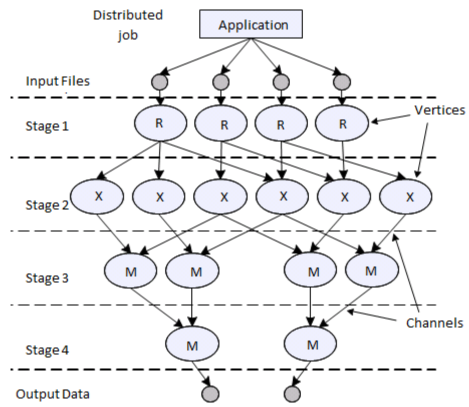


Figure 2: Dryad Job Graph

II. DRYAD OVERVIEW

Architecture of Dryad

Dryad, DryadLINQ and DSC [13] are a set of technologies that support the processing of data intensive applications in the Windows platform. Dryad is a general purpose runtime and a Dryad job is represented as a directed acyclic graph (DAG), which is called the Dryad graph. One Dryad graph consists of vertices and channels. A graph vertex is an independent instance of the data processing program in a certain step. The graph edges are the channels transferring data between the vertices. DryadLINQ is the high-level programming language and compiler for Dryad. The DryadLINQ compiler can automatically translate the Language-Integrated Query (LINQ) programs written by .NET language into distributed, optimized computation steps that run on top of the Dryad cluster. The Distributed Storage Catalog (DSC) is the component that works with the NTFS in order to provide data management functionalities, such as data sets storage, replication and load balancing within HPC cluster. Figure 1 is architecture of Dryad software stack.

Parallel Execution Model

Dryad uses directed acyclic graph to describe the control flow and dataflow dependencies among Dryad tasks that are spawn by DryadLINQ programs. The Dryad graph manager, which is a centralized job manager, reads the execution plan that was initially created by the DryadLINQ provider. Each node of the graph represents a unit of work called a vertex. The vertices are scheduled onto DSC nodes for the execution according to data locality information. If there are more vertices than DSC nodes, then the Dryad graph manager queues the execution of some of the vertices. Dryad utilizes the generalization of the UNIX piping mechanism in order to connect the vertices that comprise a job. Dryad extends the standard pipe model to handle distributed applications that run on a cluster. Figure 2 illustrates the Dryad job graph for a typical Dryad job.

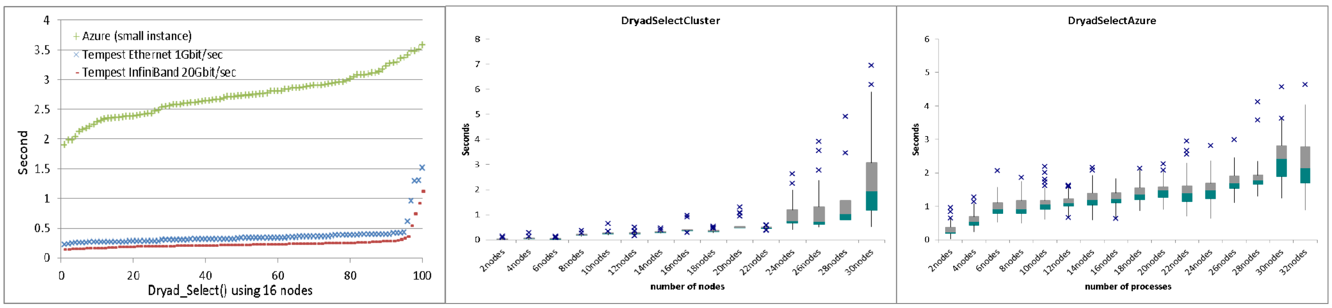
III. EVALUATING AND MEASURING THE DRYAD OVERHEAD

A. *Overhead of the Dryad Primitives*

Dryad utilizes the centralized job manager to schedule Dryad tasks to Dryad vertices in order to run LINQ queries. The centralized job scheduler can create an optimized execution plan based on global information, such as resource availability, tasks status, and workload distribution. However, the downside of centralized scheduling is that the scheduling overhead will be the performance bottleneck of many fine grain tasks [14, 19]. In order to investigate the runtime overhead of Dryad, we measured the overhead of the Select and Aggregate operations of Dryad with zero workload within the user defined function of each Dryad task. We put a timer within the user defined function, and calculated the maximum time span of all of the Dryad tasks to get an overhead of calling Select and Aggregate in Dryad.

Table 1: System Parameters of Experiment Settings

Infrastructure	Tempest (32 nodes)	Azure (100 instances)	Quarry (230 nodes)	Odin (128 nodes)
CPU (Intel E7450)	2.4 GHz	2.1 GHz	2.0 GHz	2.7 GHz
Cores per node	24	1	8	8
Memory	24 GB	1.75GB	8GB	8GB
Network	InfiniBand 20 Gbps, Ethernet 1Gbps	100Mbps (reserved)	10Gbps	10Gbps
Ping-Pong latency	116.3 ms with 1Gbps, 42.5 ms with 20 Gbps	285.8 ms	75.3 ms	94.1 ms
OS Version	Windows HPC R2 SP3	Windows Server R2 SP1	Red Hat 3.4	Red Hat 4.1
Runtime	LINQ to HPC, MS.MPI	LINQ to HPC, MS.MPI	IntelMPI	OpenMPI



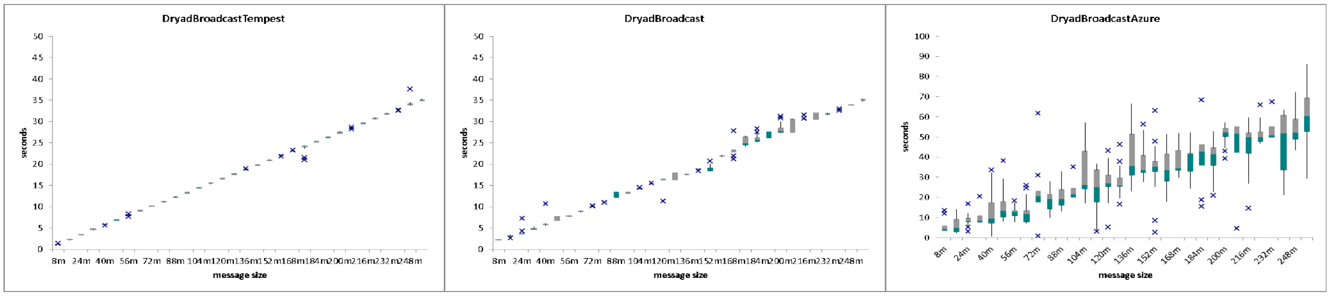
(a)Dryad_Select Tempest/Azure (b)Dryad_Select Tempest (c) Dryad_Select Azure

Figure 3. Overhead of Calling Dryad Select Operation: (a) 100 runs of Dryad_Select on 16 instances/nodes on Azure and Tempest using 1Gbps and 20Gbps network, respectively; (b) Dryad_Select using up to 30 nodes on Tempest; (c) Dryad_Select using up to 30 nodes on Azure.

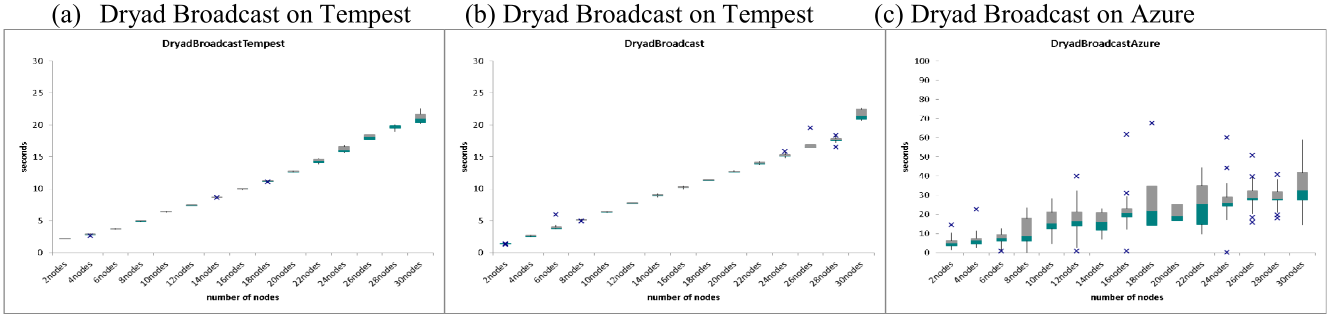
Figure 3 (a) shows the sorted overhead of invoking Dryad_Select 100 times on 16 nodes and 16 small instances on Tempest and Azure, respectively. The system parameters of all experiment settings discussed in this paper is illustrated in Table 1. The average overhead of calling Select with a zero workload on 16 nodes on different runtime environments was 0.24 (with InfiniBand on Tempest), 0.36 (with Ethernet on Tempest) and 2.73 seconds (with 100Mbps virtual network on Azure). We also conducted the same set of experiments for the Dryad Aggregate operation, and found similar overhead patterns as depicts in Figure 3(a). As a comparison, we measured the overhead of MPI_Bcast in MS.MPI with a zero payload using the same hardware resources. The average overhead of calling MPI_Bcast using 20Gbps InfiniBand, 1Gbps Ethernet on Tempest and 100Mbps virtual network on Azure were 0.458, 0.605 and 1.95 milliseconds, respectively. The results indicated Dryad prefers to deal with coarse-grain tasks due to the relative high overhead of calling Dryad primitives. We also investigated the scalability of Dryad Select on Tempest and Azure. Figures 3(b) and (c) depict the overhead of Dryad Select with a zero workload using up to 30nodes on Tempest and up to 30 instances on Azure. Figure 3 (b) showed few high random detours when using more than 26 nodes due to a more aggregated random system interruption, runtime fluctuation, and network jitter. Figure 3(c) show more random detour than figure 3(b) due to the fluctuations in the cloud environment. In sum, the average overhead of Dryad Select on Tempest and Azure were both linear with the number of nodes and varied between 0.1 to 7 seconds depending upon the number of nodes involved. Given that Dryad is designed for coarse grain data parallel applications, the overhead of calling Dryad primitives will not be the bottleneck of application performance.

B. *Overhead of Dryad Communication*

Previous studies of application usage have shown that the performance of collective communications is critical to the performance of the parallel program. In MPI, broadcast operations broadcast a message from the root process to all of the processes of the group. In Dryad, we can use the Select or Join functions to implement the broadcast operation among the Dryad vertices. We made the parameters sweep for the Dryad Broadcast operation using different numbers of nodes and message sizes. Figures 4 (a)(b) and (c) plot the time of the Dryad broadcasting message from 8MB to 256 MB on 16 nodes/instances on Tempest and Azure. The results indicated that the broadcasting overhead was linear with the size of the message. As expected, Azure had a much higher performance fluctuation due to the network jitter in the cloud. We also found that using Infiniband did not improve the broadcast performance of Dryad when compared with the results using Ethernet. The average speed of the Dryad Broadcast using 1Gbps and 20Gbps network was 108.3MB/sec and 114.7MB/sec, respectively. This minor difference in broadcast performance is due to the bottleneck of transferring data via the file pipe in Dryad. Figures 4 (d)(e) and (f) plots the overhead of Dryad broadcasting using 2~30 nodes on Tempest and Azure. While we strove to find the turning points in performance study of MPI collective communication caused by the network contention, we did not find this pattern for Dryad broadcasting because Dryad use flat tree to broadcast messages to all of its vertices and does not explore the parallelism in collective communication as MPI does. Thus, the results showed that the overhead for the Dryad broadcasting was linear with the number of computer nodes, which is not scalable behavior for message intensive applications.



(a) Dryad Broadcast on Tempest (b) Dryad Broadcast on Tempest (c) Dryad Broadcast on Azure



(d) Dryad Broadcast on Tempest (e) Dryad Broadcast on Tempest (f) Dryad Broadcast on Azure

Figure 4: Dryad broadcast overhead using different message sizes, number of nodes, and network environments. (a) message size between 8MB and 256MB on 16 nodes on Tempest with 20Gbps network; (b) (8MB~256MB) on 16 nodes on Tempest with 1Gbps network; (c) (8MB~256MB) on 16 small instances on Azure. (d) 72 MB message broadcast to 2 to 30 nodes on Tempest with 20Gbps network; (e) 72 MB on 2-30 nodes on Tempest with 1Gbps network; (f) 72 MB on 2-30 small instances on Azure.

IV. PERFORMANCE MODEL FOR DRYAD AND MPI IMPLEMENTATIONS OF PARALLEL MATRIX MULTIPLICATION

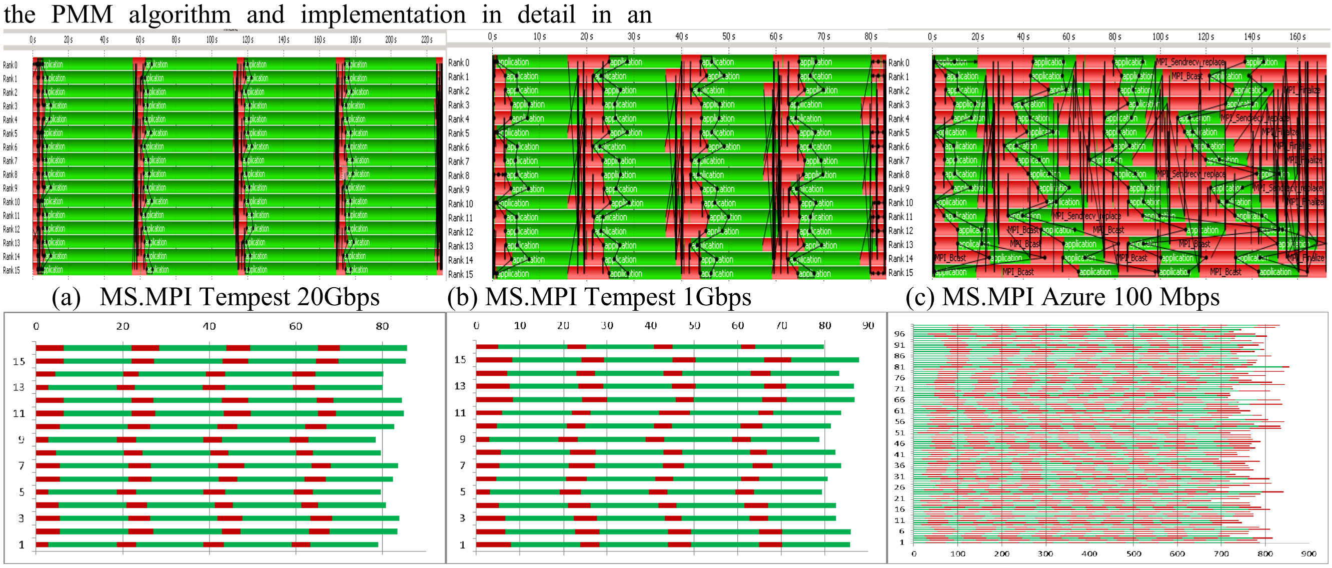
We chose the BMR algorithm of parallel matrix multiplication (PMM) application [15] to evaluate performance because BMR PMM has well established communication and computation patterns. Table 2 shows BMR PMM algorithm briefly. Figures 5 (a) (b) and (c) show the communication and computation patterns of PMM using MS.MPI on Tempest and Azure. The horizontal lines are application run times and the red bar represents the collective communication operations while the green bar represents the computation operations. The MS.MPI PMM implementation has regular communication and computation patterns on Tempest, while the Dryad implementation of PMM has a less regular pattern. The results showed that the broadcast overhead of Dryad is less sensitive than that of MPI in different network environments. As shown in Figures 5(c) and (f), both the Dryad and MPI implementations of PMM have irregular communication and computation patterns on Azure due to the network jitter in the cloud. As compared to the communication overhead, the computation overhead is relatively consistent in Tempest and Azure. Thus, we need to carefully consider the communication behavior in Dryad and MPI in order to model application performance accurately.

Table 2: BMR Algorithm for Parallel Matrix Multiplication:

Partitioned matrix A, B to blocks
For each iteration i:
1) broadcast matrix A block (j, i) to row j
2) compute matrix C blocks, and add the partial results to the results calculated in last iteration
3) roll-up matrix B block

The matrix-matrix multiplication is a fundamental kernel whose problem model has been well studied for decades. The computation overhead increases in terms of N cubic, while the memory overhead increases in terms of N square. The workload of the dense general matrix multiplication

(DGEMM) job can be partitioned into homogeneous subtasks with an even workload, and run in parallel. The regular computation and communication pattern of the homogeneous PMM tasks makes it an ideal application for our performance study of Dryad. We have already illustrated the PMM algorithm and implementation in detail in an earlier publication and technical report [16, 17, 18]. In this paper, we proposed the analytical timing model of PMM using Dryad and MPI on common HPC clusters, which is different from the analytical model proposed by Geoffrey Fox in 1987 [15] on hypercube machine.



(a) MS.MPI Tempest 20Gbps (b) MS.MPI Tempest 1Gbps (c) MS.MPI Azure 100 Mbps

(d) Dryad, Tempest, 20Gbps (e) Dryad, Tempest , 1Gbps (f) Dryad, Azure, 100Mbps,

Figure 5: communication and computation pattern of 16000x 16000 PMM jobs using different runtime environments. (a) MS.MPI on 16nodes on Tempest with 20Gbps network. (b) MS.MPI on 16nodes on Tempest with 1Gbps network. (c) MS.MPI on 16 small instances on Azure with 100Mbps network. (d) Dryad on 16nodes on Tempest with 20Gbps network. (e) Dryad on 16nodes on Tempest with 1Gbps network. (f) Dryad on 100 small instances on Azure with reserved 100Mbps network.

In order to simplify the analytical model, we assume that the input sub-matrices A and B already reside in the compute nodes that have been decomposed in the two-dimensional fashion. The elements of input matrices are double values generated randomly. We also assume that the output sub-matrix C will end up decomposed in the same way. Thus, our timings did not consider the overhead of loading sub-matrices A, B and C. We assumed that the M*M matrix multiplication jobs are partitioned and run on a mesh of ($\sqrt{N}*\sqrt{N}$) compute nodes. The size of the sub-matrices in each node is m*m, where $m=M/(\sqrt{N})$. In Dryad for the implementation of PMM, we used the Select operator to spawn ($\sqrt{N}*\sqrt{N}$) Dryad tasks, each of which runs the " broadcast-multiply-rollup" cycle on every iteration of the PMM algorithm. The overhead of Dryad Select, which equals the time taken to schedule ($\sqrt{N}*\sqrt{N}$) Dryad tasks on each iteration is:$N*T_{scheduling}$

$T_{scheduling}$ is the average overhead of scheduling one Dryad task at a time. It includes the overhead that the Dryad job manager interacts with the Windows HPC cluster scheduler via COM, and with the Dryad vertices via the file pipe. After the ($\sqrt{N}*\sqrt{N}$) Dryad tasks start running, they will run the "broadcast-multiply-roll" cycles of the algorithm. In the broadcast stage, the ($\sqrt{N}*\sqrt{N}$) tasks are split into $\sqrt{N}$ row broadcast subgroups each of which consist of $\sqrt{N}$ tasks. As Dryad uses a flat tree algorithm for broadcasting, it takes ($\sqrt{N}$-1) sequential steps to broadcast m*m data from one task to the other ($\sqrt{N}$-1) tasks within the same row broadcast subgroup. Based on the latency and bandwidth (Hockney) model, the time taken to broadcast one sub-matrix A for $\sqrt{N}$ Dryad tasks within one cycle is:

$$(\sqrt{N} - 1)*T_{startup} + (\sqrt{N} - 1)*m^2*(T_{io} + T_{comm})$$

$T_{startup}$ is the start-up time for the communication. ($T_{io}+T_{comm}$) is the time cost to transfer one matrix element between two Dryad vertices via the file pipe. We take T_{io} into account because Dryad usually uses the file pipe (NTFS temporary file) to transfer the intermediate data over the HPC cluster. Our experiment results show that the IO overhead makes up 40% of the overall overhead of point to point communication operation within Dryad.

In order to build accurate analytical model, we need further determine the overlap between communication and computation of the PMM application. In the multiplication stage, the MKL BLAS program within the user-defined function can be invoked immediately after getting the input data, and there is no need to wait for the whole broadcasting process to be finished. As shown in Figure 5, some communication of one process is overlapped with the computation of other processes. In the idea execution flow, due to the symmetry of the PMM algorithm, the communication overhead of one process over $\sqrt{N}$ iterations are successively: 0, $(m^2)*(T_{io} + T_{comm})$, $2*(m^2)*(T_{io} + T_{comm})$, $(\sqrt{N} - 1)*(m^2)*(T_{io} + T_{comm})$. Given these factors, we defined the average long term overhead of broadcasting one sub-matrix A of one process as in equation (1).The process to "roll" sub-matrix B can be done in parallel with Dryad tasks as long as the aggregated requirement of the network bandwidth is satisfied by the switch. The overhead of this step is defined in equation (2).The time taken to compute the sub-matrix product (including multiplication and addition) is defined in equation (3). Before summing up the overhead list above to calculate the overall job turnaround time, we noticed that the average scheduling overhead, $((N+1)/2)*T_{scheduling}$, was much larger than the communication start-up overhead, $((\sqrt{N}+1)/2)*T_{startup}$, which can be eliminated in the model. Finally, we defined the analytical timing model of the Dryad implementation of the PMM as the formulas (4) and (5). In addition, we defined parallel efficiency 'e' and parallel overhead 'f' as in Equations 6) and 7). The deduction of Equation 6) is based on the hypothesis: $\{(T_{comm}+T_{io})/(2M) + T_{flops} \approx T_{flops}\}$ for the large matrices. Equation 7) shows that the parallel overhead f is linear in $(\sqrt{N}*(\sqrt{N}+1))/(4*M)$, as $(T_{io}+T_{comm})/T_{flops}$ can be considered consistent for different N and M.

We also implemented PMM with MS.MPI and proposed the corresponding analytical model. The main difference between the MS.MPI and Dryad implementations lies in the step to broadcast sub-matrix A among the √N tasks. MS.MPI utilizes the binomial tree algorithm in order to implement the broadcast operation, the total number of messages that the root sends is $(\log_2 N)$, where N is the number of processes. According to the Hockney model, the communication overhead of the MS.MPI broadcasting is $T_{start\text{-}up} + (\log_2 N)*(m^2)*T_{comm}$. The average long-term overhead of broadcasting one sub-matrix A of the MS.MPI implementation of the PMM is defined as the Equation (8). The job turnaround time of the MS.MPI implementation of the PMM and corresponding parallel overhead f is defined as Equation (9) and (10). The deduction process is similar with that of the Dryad implementation of PMM. Table 3 summarizes the performance equations of the broadcast algorithms of the three different implementations. In order to make a comprehensive comparison, we also included an analysis for Geoffrey Fox's implementation in 1987.

$$\frac{(\sqrt{N}-1)}{2} * \{T_{startup} + m^2 * (T_{io} + T_{comm})\} \quad (1)$$

$$T_{startup} + m^2 * (T_{io} + T_{comm}) \quad (2)$$

$$2 * m^3 * T_{flops} \quad (3)$$

$$T(N) = \sqrt{N} * \{\frac{(N+1)}{2} * T_{scheduling} + \frac{(\sqrt{N}+1)}{2} * m^2 * (T_{io} + T_{comm}) + 2 * m^3 * T_{flops}\} \quad (4)$$

$$T(N) = \frac{\sqrt{N}*(N+1)}{2} * T_{scheduling} + \frac{(\sqrt{N}+1)*M^2}{2*\sqrt{N}} * (T_{io} + T_{comm}) + 2 * \frac{M^3}{N} * T_{flops} \quad (5)$$

$$\varepsilon = \frac{1}{N} * \frac{T(1)}{T(N)} \approx \frac{1}{1+\frac{\sqrt{N}(\sqrt{N}+1)}{4*M} * \frac{T_{comm}+T_{io}}{T_{flops}}} \quad (6)$$

$$f = \frac{1}{\varepsilon} - 1 = \frac{\sqrt{N}*(1+\sqrt{N})}{4*M} * \frac{T_{comm}+T_{io}}{T_{flops}} \quad (7)$$

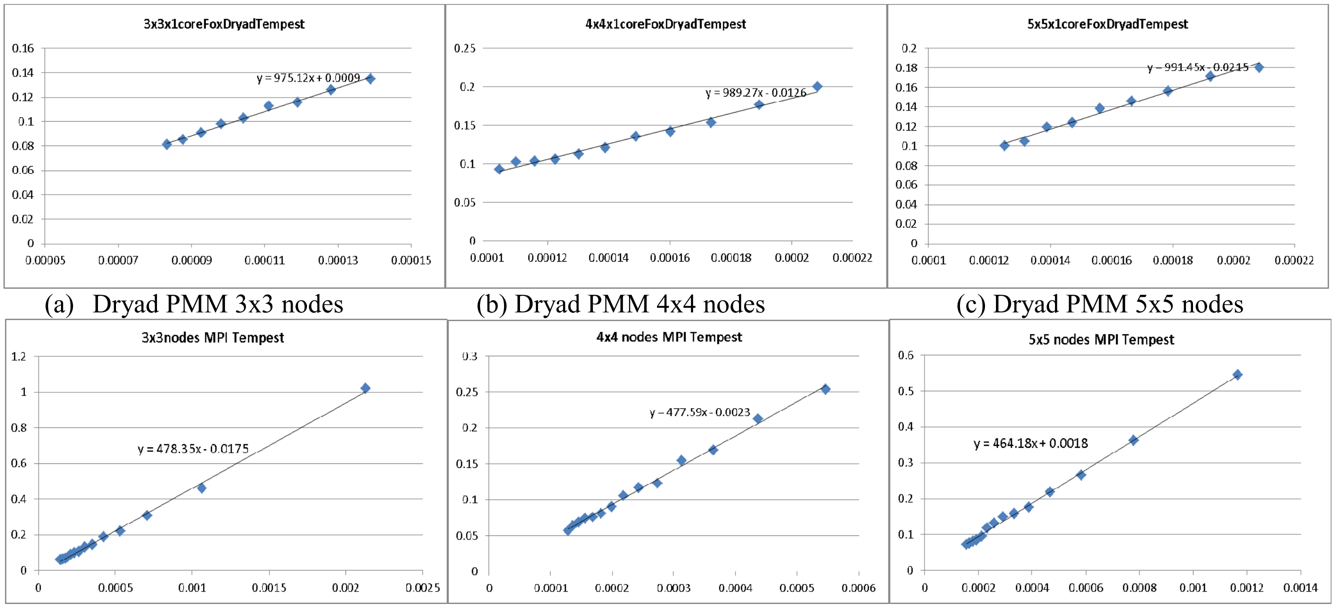
$$\frac{(\sqrt{N}-1)}{2} * T_{startup} + \frac{\log_2\sqrt{N}}{2} * m^2 * T_{comm} \quad (8)$$

$$T(N) = \frac{\sqrt{N}*(N+1)}{2} * T_{scheduling} + M^2 * \frac{(1+\log_2\sqrt{N})}{2*\sqrt{N}} * T_{comm} + 2 * \left(\frac{M^3}{N}\right) * T_{flops} \quad (9)$$

$$f = \frac{1}{\varepsilon} - 1 = \frac{\sqrt{N}*(1+\log_2\sqrt{N})}{4*M} * \frac{T_{comm}}{T_{flops}} \quad (10)$$

Table 3: Analysis of Broadcast algorithms of different implementations

Implementation	Broadcast algorithm	Broadcast overhead of N processes	Converge rate of parallel overhead
Fox	Pipeline Tree	$(M^2)*T_{comm}$	$(\sqrt{N})/M$
MS.MPI	Binomial Tree	$(\log_2 N)*(M^2)*T_{comm}$	$(\sqrt{N}*(1 + (\log_2\sqrt{N})))/(4*M)$
Dryad	Flat Tree	$N*(M^2)*(T_{comm} + T_{io})$	$(\sqrt{N}*(1 + \sqrt{N}))/(4*M)$



(a) Dryad PMM 3x3 nodes (b) Dryad PMM 4x4 nodes (c) Dryad PMM 5x5 nodes

(d) MS.MPI PMM 3x3 nodes (e) MS.MPI PMM 4x4 nodes (f) MS.MPI PMM 5x5 nodes

Figure 6: (a)(b)(c) plot parallel overhead vs. $(\sqrt{N}*(\sqrt{N}+1))/(4*M)$ of Dryad PMM using different number of nodes on Tempest. (d)(e)(f) plot parallel overhead vs $(\sqrt{N}*(1+\log_2\sqrt{N}))/(4*M)$ of MS.MPI PMM on Tempest.

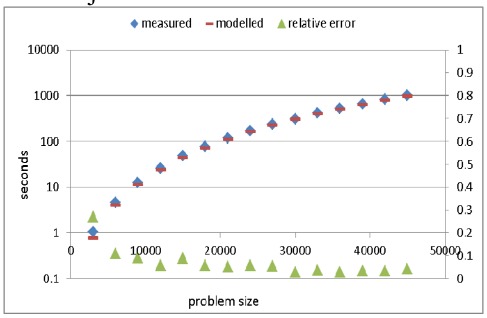
I. Experimental Analysis of Performance Model

In order to verify the soundness of the proposed analytical model (Equations 7 and 9), we investigated the consistency of T_{comm}/T_{flops} using different numbers of nodes and problem sizes. The T_{comm}/T_{flops} was linear rising term of fitting function of parallel overhead. We measured the turnaround time of the parallel matrix multiplication jobs to calculate parallel efficiency and parallel overhead as defined in Equations 5) and 6). The sequential time of the matrix multiplication jobs is the turnaround time of sequential Intel MKL BLAS jobs.

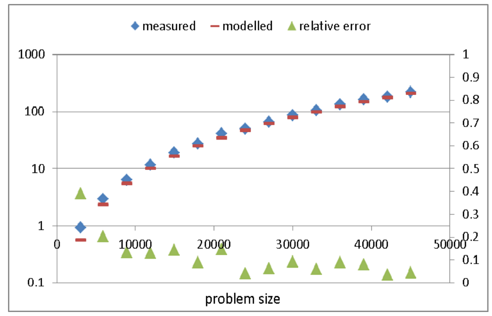
Figures 6(a)(b) and (c) depicts the parallel overhead vs. $(\sqrt{N}*(\sqrt{N}+1))/(4*M)$ of the Dryad PMM using different numbers of nodes and problem sizes on Tempest. The results show that parallel overhead 'f' is linear in small $(\sqrt{N}*(\sqrt{N}+1))/(4*M)$ (large matrices) which proves the correctness of Equation 7). The error between the linear rising term plotted in Figure 6(a)(b) and (c) and the direct measurement of $(T_{io} + T_{comm})/T_{flops}$ is 1.4%, 2.8%, and 3.08%, respectively. One should note that the linear rising term plotted in Figure 6(a)(b) and (c) include other overhead, such as synchronization, runtime fluctuation, and software latency. Overall, the communication costs will dominate those overhead as the matrices sizes increasing. Figures 6(d)(e) and (f) depicts the parallel overhead vs. $(\sqrt{N}*(1+\log_2\sqrt{N}))/(4*M)$ of the MS.MPI implementation of PMM using different number of nodes and problem sizes. The error between linear rising term of fitting function plotted in (d)(e)(f) and their corresponding measurement of T_{comm}/T_{flops} are also small than 5%. The fitting functions in Figures (d)(e)(f) indicated parallel overhead 'f' is linear in $(\sqrt{N}*(1+\log_2\sqrt{N}))/(4*M)$ which proves that the function form of Equation 7) is correct.

We further verify the proposed analytical timing model by comparing the measured and modeled job running times. Frist, we measured the overhead parameters, which included $T_{startup}$, $T_{communication}$, T_{io}, $T_{scheduling}$, and T_{flops} as discussed in the proposed analytical model. Then we calculated the modeled job running using Equation (5) and (9) with the measured parameters. Figures 7 (a)(b)(c)(d)(e) and (f) depict the comparison between the measured and modeled results of the PMM jobs of various problem sizes using different runtime environments.

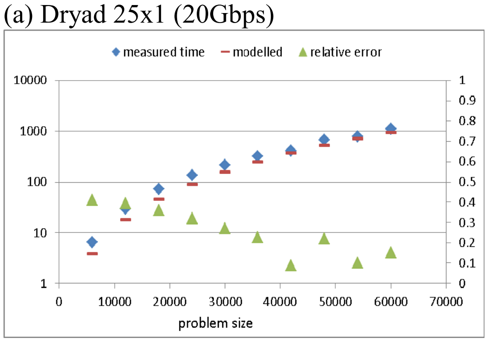
Figures 7 (d)(e) and (f) plot measured and modeled job running time of MS.MPI, IntelMPI, and OpenMPI implementations of PMM jobs using 25, 100, and 100 nodes on Tempest, Quarry, and Odin clusters, respectively. Figure 7 (d)(e) showed that the relative errors between the modeled and measured values are less than that of Dryad implementation for most larger problem sizes which further verifies the correctness of Equation (9). The higher accuracy of modeled job running time of MS.MPI and IntelMPI implementations is because the runtime latency and overhead of MPI is smaller than that of Dryad. Figure 7(f) showed the same experiments using 100 nodes on the Odin cluster. The relative error for problem size between 12000 and 48000 indicated that the proposed analytical model of OpenMPI implementation of PMM still hold when using 100 nodes as long as the aggregated network bandwidth requirement are satisfied. However, the relative error increase dramatically when problem size larger than 48000 because of the serious network contention of the OpenMPI implementation of PMM application. In fact, when using 100 nodes to run PMM jobs, there are 10 subgroups to conduct MPI_Bcast operations in parallel. Our current proposed analytical model cannot model this network contention and just consider the scenarios that network contention does not exist when running PMM jobs. Table 4 summarizes the parameters of analytical model of different runtime environments and equations of analytical model of PMM jobs using those runtime environments.



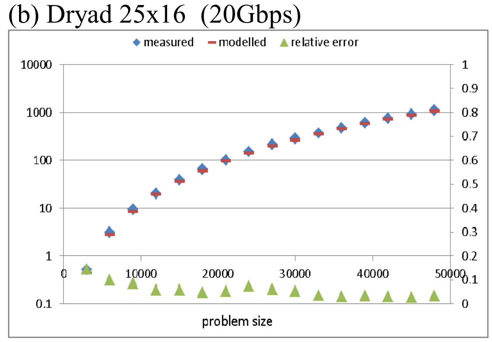
(a) Dryad 25x1 (20Gbps)



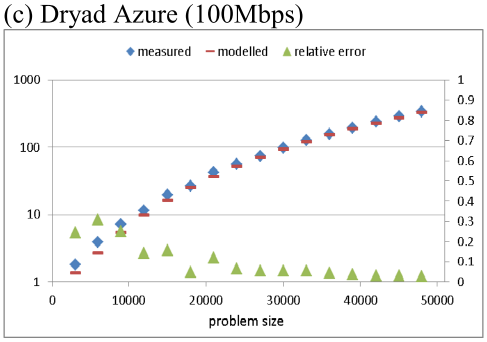
(b) Dryad 25x16 (20Gbps)



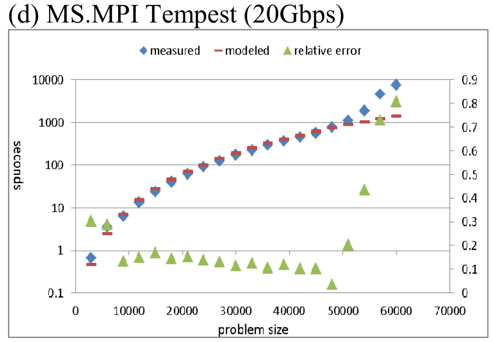
(c) Dryad Azure (100Mbps)



(d) MS.MPI Tempest (20Gbps)



(e) IntelMPI Quarry (10Gbps)



(f) OpenMPI Odin (10Gbps)

Figure 7: comparisons of measured and modeled job running time using different runtime environments. (a) Dryad PMM on 25 nodes on Tempest with 20Gbps network. (b) Dryad PMM with 25nodes with 16 cores per node on Tempest with 20Gbps network. (c) Dryad PMM on 100 small instances on Azure with 100Mbps network. (d) MS.MPI PMM on 25 nodes on Tempest with 20Gbps network. (e) IntelMPI PMM on 100 nodes on Quarry with 10Gbps network. (f) OpenMPI PMM on 100 nodes with 10Gbps network.

Table 4: Analytic model parameters of different runtime environment

Runtime environments	#nodes #cores	T_{flops}	Network	$T_{io+comm}$ (Dryad) T_{comm} (MPI)	Equation of analytic model of PMM jobs
Dryad Tempest	25x1	$1.16*10^{-10}$	20Gbps	$1.13*10^{-7}$	$6.764*10^{-8}*M^2 + 9.259*10^{-12}*M^3$
Dryad Tempest	25x16	$1.22*10^{-11}$	20Gbps	$9.73*10^{-8}$	$6.764*10^{-8}*M^2 + 9.192*10^{-13}*M^3$
Dryad Azure	100x1	$1.43*10^{-10}$	100Mbps	$1.62*10^{-7}$	$8.913*10^{-8}*M^2 + 2.865*10^{-12}*M^3$
MS.MPI Tempest	25x1	$1.16*10^{-10}$	1Gbps	$9.32*10^{-8}$	$3.727*10^{-8}*M^2 + 9.259*10^{-12}*M^3$
MS.MPI Tempest	25x1	$1.16*10^{-10}$	20Gbps	$5.51*10^{-8}$	$2.205*10^{-8}*M^2 + 9.259*10^{-12}*M^3$
IntelMPI Quarry	100x1	$1.08*10^{-10}$	10Gbps	$6.41*10^{-8}$	$3.37*10^{-8}*M^2 + 2.06*10^{-12}*M^3$
OpenMPI Odin	100x1	$2.93*10^{-10}$	10Gbps	$5.98*10^{-8}$	$3.293*10^{-8}*M^2 + 5.82*10^{-12}*M^3$

I. Summary and Conclusion

In this paper, we discussed how to analyze the influence of the runtime and communication overhead on making the analytical model for parallel program in different runtime environments. We showed the algorithm of collective communication operations and overlap between communication and computation are two important factors when modeling communication overhead of parallel programs run on data flow graph runtime.

We proposed the analytic timing model of Dryad implementations of PMM in realistic settings which is more general than empirical and semi-empirical models. We extend the proposed analytical model to MPI implementations, and make comprehensive comparison between the Dryad and MPI implementations of PMM in different runtime environments. We conducted experimental analyses in order to verify the correctness of our analytical model on a Windows cluster with up to 400 cores, Azure with up to 100 instances, and Linux cluster with up to 100 nodes. The final results show that our analytic model produces accurate predictions within 5% of the measured results. We proved some cases that using average communication overhead to model performance of parallel matrix multiplication jobs on common HPC clusters is the practical approach. Another key result we found is Dryad and MPI implementations of nontrivial parallel programs, such as PMM, may not scale well due to the behavior of their collective communication implementation or the limitation of network bandwidth. Our current analytical timing model does not consider the network contention case which requires some further study in future work. Besides, our analysis did not study other PMM algorithms or compare with other performance models, which also should be considered in the future work.

References

[1] Dean, J. and S. Ghemawat (2004). "MapReduce: Simplified Data Processing on Large Clusters." Sixth Symposium on Operating Systems Design and Implementation: 137-150.

[2] J.Ekanayake, H.Li, et al. (2010). Twister: A Runtime for iterative MapReduce. Proceedings of the First International Workshop on MapReduce and its Applications of ACM HPDC 2010 conference June 20-25, 2010. Chicago, Illinois, ACM.

[3] Malewicz, G., M. H. Austern, et al. (2010). Pregel: A System for Large-Scale Graph Processing. Proceedings of the 2010 international conference on Management of data, Indianapolis, Indiana.

[4] Isard, M., M. Budiu, et al. (2007). Dryad: distributed data-parallel programs from sequential building blocks. Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007. Lisbon, Portugal, ACM: 59-72.

[5] Yu, Y., M. Isard, et al. (2008). DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. Symposium on Operating System Design and Implementation (OSDI). San Diego, CA.

[6] Padmashree Ravindra, Seokyong Hong, etc. Efficient Processing of RDF Graph Pattern Matching on MapReduce Platforms, DataCloud 2011

[7] Torsten Hoefler, Timo Scheider, Andrew Lumsdaine, Characterizing the Influence of System Noise on Large-Scale Applications by Simulation, SC10

[8] K.C. Li. Performance analysis and prediction of parallel programs on network of workstations. Ph.D. thesis, Department of Computer Engineering and Digital Systems, University of São Paulo, 2001.

[9] Judy Qiu, Scott Beason, et al. (2010). Performance of Windows Multicore Systems on Threading and MPI. Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, IEEE Computer Society: 814-819.

[10] Apache (2010). "Hadoop MapReduce." Retrieved November 6, 2010, from http://hadoop.apache.org/mapreduce/docs/current/index.html.

[11] Guang R. Gao, Thomas Sterling, Rick Stevens, et al. ParalleX: A Study of A New Parallel Computation Model. Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International

[12] Jelena Thara Angskun, George Bosilca, Grapham E. Fagg Performance Analysis of MPI Collective Operations. Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International

[13] Introduction to Dryad, DSC and DryadLINQ. (2010). http://connect.micorosft.com/HPC

[14] Li, H., Y. Huashan, et al. (2008). A lightweight execution framework for massive independent tasks. Many-Task Computing on Grids and Supercomputers, 2008. MTAGS 2008. Austin, Texas.

[15] G. Fox, A. Hey, and Otto, S (1987). Matrix Algorithms on the Hypercube I: Matrix Multiplication, Parallel Computing, 4:17-31

[16] Jaliya Ekanayake, Thilina Gunarathne, et al. (2010). Applicability of DryadLINQ to Scientific Applications, Community Grids Laboratory, Indiana University.

[17] Hui Li, Yang Ruan, Yuduo Zhou, Judy Qiu and Geoffrey Fox, Design Patterns for Scientific Applications in DryadLINQ CTP, to appear in Proceedings of The Second International Workshop on Data Intensive Computing in the Clouds (DataCloud-2) 2011, The International Conference for High Performance Computing, Networking, Storage and Analysis (SC11), Seattle, WA, November 12-18, 2011

[18] Ekanayake, J., A. S. Balkir, et al. (2009). DryadLINQ for Scientific Analyses. Fifth IEEE International Conference on eScience: 2009. Oxford, IEEE.

[19] Zhenhua Guo, Geoffrey Fox, Mo Zhou, Yang Ruan. Improving Resource Utilization in MapReduce IEEE International Conference on Cluster Computing 2012.