A DATAFLOW MODEL FOR .NET-BASED GRID COMPUTING SYSTEMS

CHAO JIN, RAJKUMAR BUYYA

Grid Computing and Distributed Systems Laboratory
Department of Computer Science and Software Engineering
the University of Melbourne, Melbourne, VIC, Australia

LEX STEIN, ZHENG ZHANG

System Research Group
Microsoft Research Asia, Beijing, China

This paper presents the design, implementation and evaluation of a dataflow system, including a dataflow programming model and a dataflow engine, for coarse-grained distributed data intensive applications. The dataflow programming model provides users with a transparent interface for application programming and execution management in a parallel and distributed computing environment. The dataflow engine dispatches the tasks onto candidate distributed computing resources in the system, and manages failures and load balancing problems in a transparent manner. The system has been implemented over .NET platform and deployed in a Windows Desktop Grid. This paper uses two benchmarks to demonstrate the scalability and fault tolerance properties of our system.

1. Introduction

Due to the growing popularity of networked computing environments and the emergence of multi-core processors, parallel and distributed computing is now required at all levels of application development, from desktops to Internet-scale computing environments, such as Grid [6] and P2P. However, programming on distributed resources, especially for parallel applications, is more difficult than programming on centralized environment. There are many research systems that simplify distributed computing. These include BOINC [3], XtremWeb [5], Alchemi [1], and JNGI [9]. These systems divide a job into a number of independent tasks. Applications that can be parallelized in this way are called "embarrassingly parallel". However many algorithms can not be expressed as independent tasks because of internal data dependencies.

The work presented in this paper aims towards supporting advanced applications containing multiple tasks with data dependency relationships. Many resource-intensive applications consist of multiple modules, each of which receives input data, performs computations and generates output. Scientific applications for this nature include genomics [16], simulation [8], data mining

[12] and graph computing [18]. In many cases for these applications, a module's output becomes an input other modules. A coarse grained dataflow model [19] can be used to describe such applications.

We use a dataflow programming model to compose a dataflow graph for specifying the data dependency relationship within a distributed application. Under the dataflow interface, we use a dataflow engine to explore the graph to schedule tasks across distributed resources and automatically handle the cumbersome problems, such as scalable performance, fault tolerance, load balancing, etc. Within this process, users do not need to worry about the details of processes, threads and explicit communication.

The main contributions of this work are: 1) A simple and powerful dataflow programming model, which supports the composition of parallel applications for deployment in a distributed environment; 2) An architecture and runtime machinery that supports scheduling of the dataflow computation in dynamic environments, and handles failures transparently; 3) A detailed analysis of dataflow model using two sample applications over a Desktop Grid.

The remainder of this paper is organized as follows. Section 2 provides a discussion on related work. Section 3 describes the dataflow programming model with examples. Section 4 presents the architecture and design for a dataflow system over .NET platform. Section 5 presents experimental evaluation results. Section 6 concludes the paper with pointer to future work.

2. Related Work

Dataflow concept was first presented by Dennis et al. [19]. Since then several researchers have investigated various aspects of dataflow models for parallel and distributed systems and applications. As the pure dataflow is fine-grained, its practical implementation has been found to be an arduous task. Thus optimized versions of dataflow models have also been presented, including dynamic dataflow model and synchronous dataflow model [10]. Its usage continued to investigate for coarse-grained parallel applications.

River [13] provides a dataflow programming environment for scientific database like applications on clusters through a visual interface. River focuses on solving transient heterogeneity problems rather than fault tolerance problem in dynamic environment.

Grid systems such as Condor [4], Gridbus Workflow Engine [21], and Pegasus [20] provide mechanisms for workflow scheduling. Workflow systems are trying to seek opportunities for concurrency at the level of tasks. However, how to easily achieve the concurrency within each task has been ignored. Kepler [15] provides a graph based interface for scientific workflow scheduling, and Grid superscalar [14] allows users to write their applications in a sequential way. However they do not focus on handling failures.

MapReduce [7] is a cluster middleware designed to help programmers to transform and aggregate key-value pairs by automating parallelism and failure

recovery. Their programming model can also be taken as a fixed static dataflow graph.

3. Programming Model

Dataflow programming model abstracts the process of computation as a *dataflow graph* consisting of *vertices* and directed *edges*.

The *vertex* embodies two entities:

a) The data created during the computation or the initial input data;

b) The execution module to generate the corresponding vertex data.

The directed *edge* connects vertices, which indicates the dependency relationship between vertices. A vertex, takes its dependent vertices as its inputs.

A vertex is an *initial vertex* if there are no edges pointing to it but it has edges pointing to other vertices; correspondingly, a vertex is called a *result vertex* if it has no edges pointing to other vertices and there are some edges pointing to it. An initial vertex does not have an associated execution module.

Our current programming model focuses on supporting a static dataflow graph for SPMD (Single Program Multiple Data) applications, which means the number of vertices and their relationships are known before execution. We expect the graph to be a Directed Acyclic Graph (DAG).

3.1. *Namespace for vertices*

Each vertex has a unique name in the dataflow graph. The name consists of 3 parts: *Category*, *Version* and *Space*. Thus, the name is denoted as **<*C*, *T*, *S*>**. *Category* denotes different kinds of vertices; *Version* denotes the index for the vertex along the time axis during the computing process; *Space* denotes the vertex's index along the space axis during execution. In the following text, we call vertex name as *name*.

3.2. *Dataflow library API*

3.2.1. *Specifying Execution Module*

To specify instructions/code to be executed, which is called execution module and used to generate the output for each vertex, users need to inherit the *Module* class in dataflow library for writing each vertex's execution code. In particular, users need to implement 3 virtual functions:

- *ModuleName* **SetName**() : specify a name for the execution module, which is used as an identifier during editing the data dependency graph.
- void **Compute**(*Vertex*[] *inputs*) : implemented by users for generating output taking input vertex data. The input data is denoted by *inputs*. Each element of *inputs* consists of a *name* and a data buffer.
- byte[] **SetResult**():called by the system to get the output data after Compute() is finished.

3.2.2. *Composing Dataflow Graph*

The *dataflow* API provides two functions for composing the static data dependency graph:

- **CreateVertex**(*vertex*, *ModuleName*) : is used to specify the name and corresponding execution module for each vertex.
- **Dependency**(*vertex*, *InputVertex*): is used to add y as x's dependent vertex.

Two functions are provided to set the initial and result vertices as follows:

- **SetInitialVertex**(vertex, file)
- **SetResultVertex**(vertex, file)

3.3. *Example*

Given the matrix vector iterative multiplication, $V^t=M*V^{t-1}$. We partition the matrix and vector by rows into m pieces respectively, as $V_i^t = \sum_{j=1}^{m} M_i.*V_j^{t-1} (i = 1...m)$.

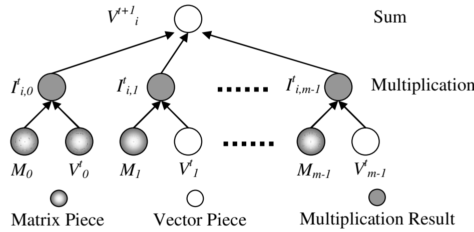

Figure 1: Dataflow graph for the i-th vector piece.

The corresponding dataflow graph is illustrated by **Figure 1**. To name vertices, *Category* = *M* denotes the matrix vertices and *Category* = *V* denotes the vector vertices. For *i*-th vector vertex, the data relationship should be specified as: <*V*, *t*, *i*>←{<*M*, *0*, *i*>, <*V*, *t*-1, *j*>} (j=1…*m*).

4. Architecture and Design

This section describes a dataflow architecture designed for a Windows Desktop Grid consisting of commodity PCs based on .NET platform. The environment consists of idle desktops that are used for computing but drop out of the system as soon as interactive programs are started by the users on them. Such nodes can rejoin the system when they are idle again.

4.1. *System Overview*

The dataflow system consists of a single master and multiple workers as illustrated in **Figure 2**. The master is responsible for accepting jobs from users, organizing multiple workers to work cooperatively, sending executing requests to workers and handling failures of workers. Each worker contributes CPU and disk resources to the system and waits for executing requests from the master.

4.2. *The Structure of the Master*

The master is responsible for monitoring the status of each worker, dispatching ready tasks to suitable workers and tracking the progress of each job according to the data dependency graph. On the master, there are 4 key components:

- *Membership component*: maintains the list of available worker nodes. When some nodes join or leave the system, the list is updated correspondingly. The membership is maintained through heartbeat signal between master and workers. The heartbeat signal also carries the status information about the worker, such as CPU, memory and disk usage.
- *Registry component*: maintains the location information for available vertex data. In particular, it maintains a list of indices for each available vertex data. Each vertex has an index, which lists workers that hold its data.
- *Dataflow Graph component*: maintains the data dependency graph for each job, keeps track of the availability of vertices and explores ready tasks. When it finds ready tasks, it will notify the scheduler component.
- *Scheduler component*: dispatches ready tasks to suitable workers for executing. For each task, the master notifies workers of inputs & initiates the associated execution module to generate the output data.

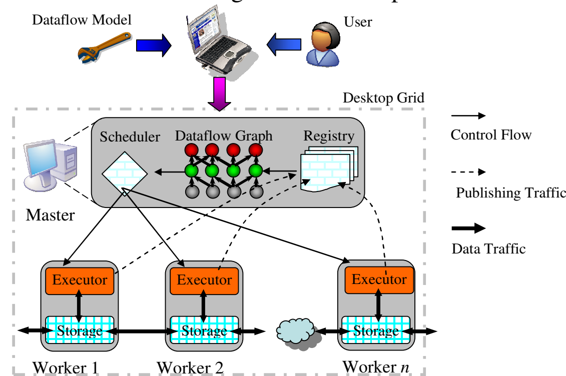

Figure 2: Architecture of dataflow-based desktop Grid computing system.

4.3. *The Structure of a Worker*

Workers work in a peer to peer fashion. To cooperate with the master, each worker has two functions: executing upon requests from master and storing the vertex data. Correspondingly there are 2 important components on each worker:

- *Executor component*: receives executing requests from the master, fetches input from the storage component, generates output to the storage component and notifies *master* about the available vertex of the output data.
- *Storage component*: is responsible for managing and holding vertex data generated by executors and providing it upon requests. Actually the storage components across workers run as a distributed storage service. To handles failures, upon request from master, the storage component can keep data

persistently locally or replicate some vertices on remote side to improve the reliability and availability.

4.4. *System Interaction*

Upon receiving submission from users, including the dataflow graph and execution modules, the master node will create an instance as a thread for each module. Based on .NET platform, the master serializes the execution module as an object, and then sends it to workers when dispatching vertices executing.

To begin the execution, master node first sends the initial vertex to workers. When a worker receives the vertex, its storage component will keep it, and then notifies the registry component through an index publishing message.

Every time the registry component receives an index publishing message for vertex x, it updates x's index and then notifies the dataflow graph component to check if there is a vertex execution waiting x. If so, the ready vertex will be scheduled as an executing task. The scheduling component sends the executing request to candidate workers. The execution request carries the serialized object of corresponding execution module, and the location information of the input vertex data. After receiving it, the worker first fetches the input data, and then un-serializes the execution object and executes it.

To improve the scalability of the system, workers transfer vertex data in a P2P manner. Whenever the executor component receives an executing request from master node, it sends a fetch request to the local storage component. If there is non local copy for the requested data, the storage component will fetch the data from remote worker according to the location specified in the executing request. After all the input data is available on the worker node, the executor component creates a thread instance for the execution module based on the serialized object from the master, feeds it with the input vertices and starts the thread. After the computation finishes, the executor component saves the result vertex into local storage component and notify the registry component.

The storage component keeps hot vertex data in memory while holding cold data on disk. The vertex data will be dumped to disk asynchronously when there is a need to reduce memory space. Worker schedules the executing and network traffic of multiple tasks as a pipeline to optimize the performance.

4.5. *Fault Tolerance*

In our Desktop Grid environment, besides physical failure (node cannot work due to software or hardware problems), we frequently face soft failure. Soft failure occurs when higher priority users demand node resources and the dataflow system yields. We use same mechanisms handling both failures.

4.5.1. *Worker Failure*

The *master* monitors status for each task dispatched to workers. Each vertex task has 4 statuses: *unavailable*, *executing*, *available* and *lost*. *Unavailable* and *lost* means no any copy exists in the dataflow storage for the vertex. the difference between these two statuses is *unavailable* is specified to the vertex which is

never generated before, while *lost* means the vertex has been generated before but now lost due to worker failures. *Available* means that at least one copy for the vertex is held by some storage component in the dataflow system. *Executing* the vertex has been scheduled to some worker but still not finished.

The failure of a worker/node leads to termination of the task it is processing and the master needs to re-schedule such tasks elsewhere. Furthermore, since the vertex data on the failure worker will not be accessible again, the master node will need to regenerate them if there are some *unavailable* tasks are eventually dependent on them.

When the master detects that a worker has failed, it notifies the registry component to remove the failed worker from indices. During the removing process, status of some of the vertices will change from *available* to *lost*. For the *lost* vertices, if they are directly dependent by some *executing* or *unavailable* vertex tasks, we need to regenerate them to continue the execution. The rescheduled tasks may be dependent on other *lost* vertices, and eventually cause domino effects. For some extreme cases, the master node may need to re-send the initial vertices to continue the execution.

Generally, rescheduling due to the domino effect will takes considerable time. The system replicates vertices between workers to reduce rescheduling. This is a feature triggered by the configuration of the master. If replication feature is set, the registry component will choose candidate workers to replicate the vertex after it receives the first publishing message for that vertex. Replication algorithm needs to take load balancing into consideration.

Replication causes additional overhead. If we take vertices under same version as a checkpoint for the execution, it is not necessary for us to replicate every checkpoint. It is better for users to specify a replication step. It is called as *n* step replication if users want to replicate the vertices every *n* versions. Under failure cases, there is a tradeoff between replication steps and executing time.

4.5.2. *Master Failure*

Generally *master* is running over a dedicated node, it may experience physical failures, but seldom has soft failures. To handle this, it frequently writes its internal status, including data structure in registry component, scheduler component and graph component to disk and then replicate the internal status to other node. After the master node fails, we could use the backup version to start a new master and continue the computation.

4.6. ***Scheduling***

In the current design, the scheduling is performed by the master giving priority to locality of data [11] and performance history of workers [17]. For an efficient scheduling, the size of each input vertex data and computing power, i.e. CPU frequency, are taken as the measure for load balancing. The scheduler collects the related performance information for each execution module, such as the input data size and time consumed. Based on this history information, we can predict the execution time for the execution module which has been scheduled.

5. Performance Evaluation

In this section, we evaluate the performance of the dataflow system through two experiments running in a Windows Desktop Grid deployed in Melbourne University and shared by students and researchers.

5.1. *Environment Configuration*

The evaluation is executed in a Desktop Grid with 9 nodes. During testing, one machine works as master and the other 8 machines work as workers. Each machine has a single Pentium 4 processor, 500MB of memory, 160GB IDE disk (10GB is contributed for dataflow), 1 Gbps Ethernet and ran Windows XP.

5.2. *Testing Benchmarks*

We use two examples as testing programs for the evaluation. These examples are built using the dataflow API.

(a) Matrix Multiplication

In this benchmark one matrix is multiplied with another one. Each matrix consists of 4000 by 4000 randomly generated integers. We partition matrix into square blocks with two granularities: 250 by 250 block and 125 by 125 block.

The firs granularity of partition generates 16*16 blocks (255KB per block), and, the second one generates 32*32 blocks (63KB per block).

(b) Matrix Vector Iterative Multiplication

In this benchmark, one matrix is multiplied with one vector in an iterative manner. The matrix consists of 16000 by 16000 random integers, and the vector consists of 16000 random integers. The matrix is about 1GB and the vector is 64KB. The matrix and vector are partitioned by rows. Two granularities for partition are adopted in the evaluation: 24 stripes and 32 stripes.

For 24 stripes, the matrix and the vector are respectively partitioned by rows into 24 pieces. So there are 1200 vertices are generated. For 32 stripes, there are 1600 vertices are generated.

5.3. *Scalability of Performance*

Figure 3 illustrates the speedup of performance with an increasing number of workers. We can see that under same vertex partition settings as more workers are involved in the computation, better performance is obtained. On the other hand, overheads such as connections with the master also increase with the number of workers. So the speedup line is not ideal.

The matrix vector multiplication benchmark illustrates a super linear speedup phenomenon for one worker. The reason is one computer has only 500MB memory and is not enough to hold 1GB matrix. The swapping overhead causes reduction in performance, where this is not the case with parallelism as data is distributed across multiple workers.

One expectation of partition granularity is that more partitions will introduce additional overhead during execution.

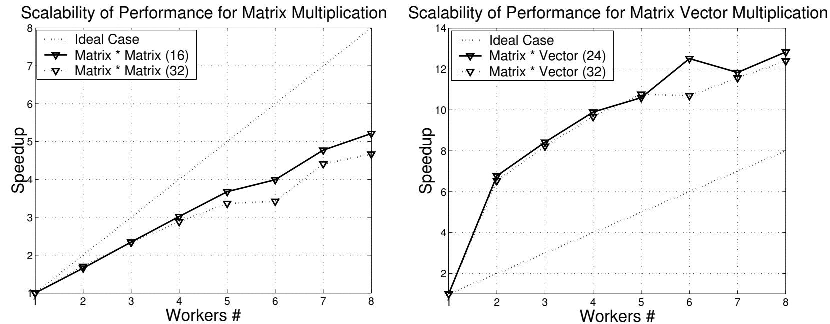

Figure 3: Scalability of performance.

5.4. *Handling Worker Failure*

This section evaluates the mechanisms dealing with worker failure, including replication and rescheduling. We use iterative matrix vector multiplication with 24 partitions and 100 iterations. In total, 2400 vertices are generated during the testing. As vertices created have same size, we measure the vertices number.

8 workers and 1 master involve in the testing. We first collect the number of vertices without worker failures and replication, as the first line in Figure 4. The whole testing lasts for about 4 minutes. Within the initial phase, where the line is nearly flat, the master node sends initial vertices to workers. It is a sequential process. After that, the execution begins and the slope of vertices number line increases heavily. After all vertices are created, the line changes to flat.

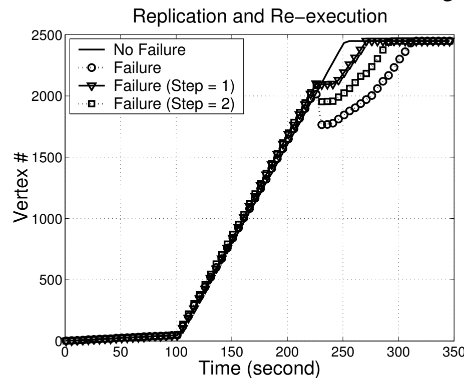

Figure 4: Handling worker failures with replication and re-execution.

Next we add one worker failure in the testing. We unplug one worker's network cable to simulate its failure at around the 4th minute. When we do not take replication, one worker failure causes some vertices to be lost, illustrated by the 2nd line. Once the master detects the failure, it will dispatch live workers to regenerate lost vertices and then continue the execution. So there is a big drop at

the 230th second in the 2nd line. After generation of lost vertices, only 7 workers are available, so the slope is smaller than the one before the drop point.

Then we add replication mechanism to handle the failure. We test two settings: 1-step and 2-step replication. Compared with no replication, 2-step replication has only a small drop during the failure while 1 step replication has no drop. Eventually we can see replication mechanism effectively reduces the time consumed for regenerating lost vertices.

6. Conclusion

We presented a dataflow computing platform within shared cluster environment. Through a static dataflow interface, users can freely express their data parallel applications and easily deploy applications in distributed environments. The mechanisms adopted in our system support scalable performance and transparent fault tolerance. We plan to incorporate dataflow model into Alchemi.

Acknowledgments

We would like to thank Yu Chen, Krishna Nadiminti, Srikumar Venugopal, Hussein Gibbins, Marcos Dias de Assuncao, Xingchen Chu and Marco A. S. Netto for their support. This work is partially supported by grants from the eWater CRC and DEST International Science Linkage Program.

References

1 A. Luther, R. Buyya, R. Ranjan, and S. Venugopal. *Alchemi: A .NET-Based Enterprise Grid Computing System.* 6th International Conference on Internet Computing, 2005, Las Vegas, USA.

2 Arvind and R. Nikhil. *Executing a program on the MIT tagged-token dataflow architecture.* IEEE Transaction Computers. 39, 3, 300–318, 1990.

3 D. P. Anderson. *BOINC: A System for Public-Resource Computing and Storage*. Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing, R. Buyya (ed.), IEEE CS Press, USA, 2004.

4 D. Thain, T.Tannenbaum, et al. *Distributed computing in practice: The Condor experience.* Concurrency and Computation: Practice and Experience, 17(2-4), February/April 2005.

5 G. Fedak, C. Germain, et al. *Xtremweb: A generic global computing system.* Proceedings of the 1st International Symposium on Cluster Computing and the Grid (CCGrid 2001), Brisbane, Australia, 2001.

6 I.Foster and C.Kesselman. *The Grid Blueprint for a Future Computing Infrastructure*. Morgan Kaumann Publishers, USA, 1999.

7 J. Dean and S. Ghemawat. *MapReduce: Simplified Data Processing on Large Clusters*. Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI'04), San Francisco, CA, 2004.

8 J.F.Cantin and M.D.Hill. *Cache Performance for Selected SPEC CPU2000 Benchmarks*. Computer Architecture news (CAN), 2001.

9 J. Verbeke, N. Nadgir, et al. *Framework for peer-to-peer distributed computing in a heterogeneous, decentralized environment.* Proceedings of the 3rd International Workshop on Grid Computing, 2002.

10 E. Lee and D. Messerschmmitt. *Static scheduling of synchronous dataflow programs for digital signal processing.* IEEE Transactions Computers. C-36, 1, 24–35, 1987.

11 C. Polychronopoulos and D. Kuck. *Guided self-scheduling: A practical scheduling scheme for parallel supercomputers*. IEEE Transactions on Computers, 36 (12), 1425–1439, 1987.

12 R.Agrawal, T.Imielinski, et al. *Database mining: A Performance Perspective.* IEEE Transactions on Knowledge and Data Engineering, 5(6):914-925, 1993.

13 R.H.Arpaci-Dusseau, Eric Anderson. Noah Treuhaft, et al. *Cluster I/O with River: Making the fast case common.* Sixth Workshop on I/O in Parallel and Distributed Systems, May 5, 1999, Atlanta, GA, USA. ACM, 1999.

14 Rosa M. Badia, J. Labarta, et al. *Programming Grid Applications with GRID superscalar*, Journal of Grid Computing, Volume 1, Issue 2, 2003.

15 S. Bowers, B. Ludaescher,et al*.* *Enabling Scientific Workflow Reuse through Structured Composition of Dataflow and Control-Flow*. IEEE SciFlow: The IEEE International Workshop on Workflow and Data Flow for Scientific Application, 2006.

16 S. Altschul, T. Madden, et al. *Gapped BLAST and PSI-BLAST: a new generation of protein database search programs.* In Nucleic Acids Research, pages 3389-3402, 1997.

17 W. Smith, V. Taylor, and I. Foster. *Using Run-Time Predictions to Estimate Queue Wait Times and Improve Scheduler Performance*. Proceedings of the 5th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP '99), Springer-Verlag, 1999.

18 T.L.Lancaster. *The Renderman Web site.* http://www.renderman.org, 2002.

19 W. M. Johnston, J. R. Hanna, et al. *Advances in Dataflow Programming Languages.* ACM Computing Surveys, 36(1):1–34, March 2004.

20 E. Deelman et. al. *Pegasus: a Framework for Mapping Complex Scientific Workflows onto Distributed Systems*. Scientific Programming Journal, Vol 13(3), 2005, Pages 219-237.

21 J. Yu and R. Buyya. *Scheduling Scientific Workflow Applications with Deadline and Budget Constraints using Genetic Algorithms*. Scientific Programming Journal, 14(3-4), 2006, pp. 217 – 230.