A Dataflow System for Unreliable Computing Environments

Chao Jin, Zheng Zhang[*], Lex Stein[*], and Rajkumar Buyya

GRIDS Laboratory, Dept. of CCSE
The University of Melbourne, Australia
{chaojin, raj}@csse.unimelb.edu.au

System Research Group[*]
Microsoft Research Asia, China
{Zheng.Zhang, Lex.Stein}@microsoft.com

Abstract

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

The growing popularity of distributed computing systems, such as P2P [18] and Grid computing [11], amplifies the scale and heterogeneity of network computing model. This is leading to use of distributed computing in e-Science [33] and e-Business [26] applications. However, programming on distributed resources, especially for parallel applications, is more difficult than programming on centralized environment. In particular, a distributed systems programmer must take extra care with data sharing conflicts, deadlock avoidance, and fault tolerance. Programmers lacking experiences face newfound difficulties with abstractions such as processes, threads, and message passing. A major goal of our work is to ease programming by simplifying the abstractions.

There are many research systems that simplify distributed computing. These include BOINC [4], XtremWeb [10], Alchemi [1], SETI@Home [18], Folding@Home [8] and JNGI [17]. These systems divide a job into a number of independent tasks. Applications that can be parallelized in this way are called "embarrassingly parallel". Embarrassingly parallel applications can easily utilize distributed resources, however many algorithms can not be expressed as independent tasks because of internal data dependencies.

The work presented in this paper provides support for more complex applications by exploiting the data dependency relationship during the computing process. Many resource-intensive applications consist of multiple modules, each of which receives input data, performs computations and generates output.. Scientific data-intensive examples include genomics [28], simulation [16], data mining [24] and graph computing [32]. In many cases for these applications, a module's output becomes other modules' input. Generally, we can use *dataflow* [34] to describe such a computing model.

A computing job can be decomposed into a data dependency graph of computing tasks, which can be automatically parallelized and scheduled across distributed computing resources.

This paper presents a dataflow programming model used 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 dataflow graph to schedule tasks across distributed resources and automatically handle the cumbersome problems, such as scalable performance, fault tolerance, load balancing, etc. In particular, the dataflow engine is responsible for maintaining the dataflow graph, updating availability status of data and scheduling tasks onto workers in a fault tolerant manner. 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. The users can create a dataflow graph in a simple manner programmatically, based on which data needed and generated during execution is partitioned into suitable granularity. Each partition is abstracted as a *vertex* in the graph. Each vertex is identified by a unique name, through which the dependency relationship is specified. Also, users need to specify the execution module for each vertex, which is used to generate output vertices from available input vertices. A storage layer is responsible for holding vertices generated during execution.

2) An architecture and runtime machinery that supports scheduling of the dataflow computation in dynamic environments, and handles failures transparently. This system is especially designed for Desktop Grid environments. We use two methods for handling failures: re-scheduling and replication.

3) A detailed analysis of dataflow model using two sample applications over a Desktop Grid. We have investigated scalability, fault tolerance and execution overhead.

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 several examples. Section 4 presents the architecture and the design with a prototype implementation of the dataflow system over .NET platform. Section 5 reports the experimental evaluation of the system. Section 6 concludes the paper with pointer to future work.

2. Related work

Dataflow concept was first presented by Dennis et al. [13] [34] and has led to a lot of research. As the pure dataflow is fine-grained, its practical implementation has been found to be an arduous task[2]. Thus optimized versions of dataflow models have also been presented, including dynamic dataflow model[3] and synchronous dataflow model[20]. However, the dataflow concept still attracts a great interest because it is a natural way to express parallel applications and plays an important role in applications such as digital signal processing for coarse-grained parallel applications [23].

Grid computing platforms such as Condor [15][6] provide mechanisms for workflow scheduling. Condor can manage resources in a single cluster, multiple clusters and even clusters distributed in a Grid-like environment. However, Condor works at the granularity of a single job. Within each job, there may be multiple processes cooperating with message passing middleware. Condor does not focus on the programming difficulties associated with the data communication within one job, but emphasizes on the high level problem of matching the available computing power with the requirements of jobs. Furthermore, workflow research falls into the control flow category, which has a different interface and programming model to that of dataflow work. Most other Grid platforms, for example, Globus [11] and Nimrod [5], share similar interests as to Condor system.

River [25] provides a dataflow programming environment for scientific database like applications [21][29] on clusters of computers through a visual interface. River uses a distributed queue and graduated declustering to provide maximum performance even in the face of non-uniformities in hardware, software and workload. However the dataflow interface in River is coupled with components and the granularity of data is not fine enough for efficient scheduling. Furthermore, River does not focus on the fault tolerance problem in dynamic environment.

MapReduce [14] is a cluster middleware designed to help programmers to transform and aggregate key-value pairs by automating parallelism and failure recovery. Their programming model is specific to the needs of Google. Actually each MapReduce computing can be easily expressed as a dataflow graph.

BAD-FS [12] is a distributed file system designed for batch processing applications. Through exposing explicit policy control, it supports I/O-intensive batch workload through batch-pipeline model. Although BAD-FS shares some similarity with our proposal on how to deal with failures and achieve efficient scheduling, it aims to support batch applications with scheduling on task granularity and does not focus on how to reduce the difficulties for programming in distributed environment.

Kepler [27] is a system for scientific workflows. It provides a graph interface for programming. Its visual programming model is especially suitable for a small number of components. In comparison, language support for composing the dataflow graph and programming model is more flexible and more suitable to partition large scale data and schedule the execution on them over distributed resources.

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 from users;

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

The directed *edge* connects vertices within the dataflow graph, which indicates the dependency relationship between vertices. Generally we expect the dataflow graph to be a Directed Acyclic Graph (DAG).

We call a vertex an *initial vertex* if there are no edges pointing to it and 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. Generally, an initial vertex does not have an associated execution module.

Given a vertex, *x*, its neighbor vertices which have edges pointing to it are its inputs. If all its inputs are ready (i.e. the data of the input vertices are available), its execution module will be triggered automatically to generate its data. Then the generated data may be now available as the input for other vertices. In the beginning, we assume all the initial vertices should be available. A reliable storage system holds all the data for vertices.

Our current programming model focuses on supporting a static dataflow graph, which means the number of vertices and their relationships are known before execution.

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*. In particular, *Category* is a string; the type of *Version* is integer and the type of *Space* is integer array.

As an example, Figure 1 illustrates a one dimensional cellular automata application with 5 cells. Each vertex represents a cell. *Version* 0 means the initial status of cells. Through the execution, each cell updates its status two times, as the result of *Version* 1 and *Version* 2 respectively. Each update is based on its right neighbor and its own status at the last step. For example, <*CA*, 1, 2> denotes the second vertex in *Version* 1. Actually, the data relationship could be specified as:

$<CA, t, s> \leftarrow \{<CA, t\text{-}1, s>, <CA, t\text{-}1, s+1>\}$.

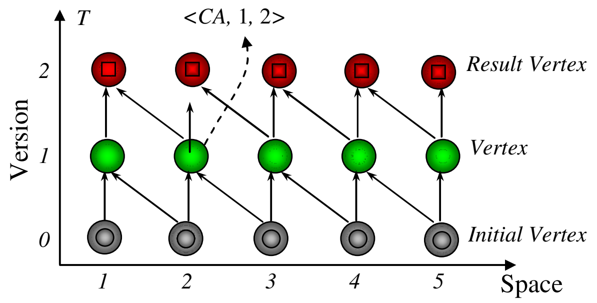


Figure 1 *The dataflow graph for a one dimensional Cellular Automata computation. Each vertex represents a Cell. Each cell updates its status two times and the updating takes its own status and the status of its right neighbor in the last step as inputs.*

Another example is an iterative matrix and vector multiplication, $V^t=M*V^{t-1}$. To parallelize the execution, we partition the matrix and vector into rows of *m* pieces with each piece being denoted as a vertex. To name them, *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:

$<V, t, i> \leftarrow \{<M, 0, i>, <V, t\text{-}1, j>\}$ ($j=1\ldots m$).

3.2 Dataflow library API

3.2.1 Specifying Execution Module

Besides the data dependency relationship, users also need to specify instructions/code to be executed, which we refer to as the execution module, to generate the output for each vertex. Users can inherit the *Module* class in dataflow library to write execution code for each vertex. To do that, users need to implement 3 virtual functions:

- *ModuleName* **SetName**()
- void **Compute**(*Vertex*[] *inputs*)
- byte[] **SetResult**()

SetName() is used to specify a name for the execution module, which will be used as an identifier during editing the data dependency graph. Each different module should have a unique name.

Compute() is implemented by users for generating output data taking input data from other vertices. The input data is denoted by the input parameter *inputs*. Each element of *inputs* consists of two parts: a *name* and a data buffer.

SetResult() is 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*)
- **Dependency**(*vertex*, *InputVertex*).

CreateVertex() is used to specify the name and corresponding execution module for each vertex, denoted by *vertex* and *ModuleName* respectively. The dataflow library will maintain the internal data structure for created vertex, such as its dependent vertices list.

Dependency(*x*, *y*) is used to add *y* as a dependent vertex of vertex *x*. The dataflow library will add *x* to *y*'s dependent vertices list, which is created when calling CreateVertex() for *x*.

Given two vertices, x and y, to specify their dependency relationship, users should first call CreatVertex() for x and y respectively and then call Dependency() to specify their relationship.

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

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

Generally the initial vertices are some input files from users and the result vertices are the final execution result.

3.3 Example

Given the matrix vector iterative multiplication example, $V^t=M*V^{t-1}$. We partition the matrix and vector by rows into m pieces respectively, as $V_i^t = \sum_{j=1}^{m} M_i * V_j^{t-1} (i = 1...m)$. The corresponding dataflow graph is illustrated by Figure 2.

For this computing, users may use two basic execution modules: multiplication of matrix and vector pieces and sum of m multiplication results.

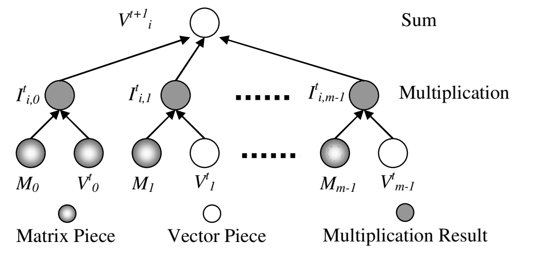


Figure 2 Dataflow graph for the *i*-th vector piece

```
class Multiple : Module {
    byte[] result;
    override string SetName() {
            return "multiple";
    }
    override void Compute(Vertex[] inputs) {
        /*unpack matrix & vector piece
           from inputs*/
          /*compute multiplication*/
          /*put result into result*/
    }
    override byte[] SetResult() {
           return result;
    }
}
```

Figure 3 ***Multiplication Module***

Figure 3 and Figure 4 show the two basic modules. In Multiplication module, *inputs* for Compute() should be M_i and V_i^{t-1}, and *result* should be their multiplication. In Sum module, *inputs* for Compute() should be m multiplications from Multiplication module, and *result* should be V_i^t.

It depends on the user to combine these 2 modules into one execution module.

```
class Sum : Module {
    byte[] result;
    override string SetName() {
            return "sum";
    }
    override void Compute(Vertex[] inputs) {
        /*unpack m multiplication piece
           from inputs*/
          /*compute sum*/
          /*put result into result*/
    }
    override byte[] SetResult() {
           return result;
    }
}
```

Figure 4 ***Sum Module.***

Given m partitions and T iterations, Figure 5 illustrates how to edit the data dependency graph for this example.

```
for (int i = 0; i < m; i++)
    //m pieces matrix vertices
    CreateVertex(name("M",0,i), null);
for (int i = 0; i < m; i++)
    //m pieces vector vertices
    CreateVertex(name ("V",0,i), null);
for (int t = 0; t < T; t++) { //T iteration
    for (int i = 0; i < m; i++) {
        matriV = name ("M", 0, i);
        for (int j = 0; j < m; j++) {
           /*multiplication    result*/
           interV = name ("I", t, i, j);
           CreateVertex(interV, "Multiplication");
           vecV = Vertex("V", t-1, j);
           Dependency(interV, matriV);
           Dependency(interV, vecV);
        }
        sumV = Vertex("V", t, i); //sum result
        CreateVertex(sumV, "Sum");
        for (int j = 0; j < m; j++) {
           interV = Vertex("I", t-1, i, j);
           Dependency(sumV, interV);
        }
    }
}
```

Figure 5 ***Composition of the dataflow graph.***

Finally users set the input files for the matrix and vector pieces through SetInitialVertex(). Also users need to specify to collect the final version of vector as the result through SetResultVertex().

4. Architecture and Design

The design of a dataflow system should take into account the features of its target execution environment. For example, in a Grid environment, which generally consists of multiple supercomputing nodes and high bandwidth network across different geographic locations, we need to handle large latency across wide area networks and make the granularity of tasks so that computation to communication ratio is high; in desktop computing or volunteer computing environments, that generally consist of large number of dynamic commodity PCs across WAN, it is better to consider the high failure rate of contributing PCs and the granularity of tasks should be small enough for commodity PCs.

This section describes an 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 distributed system as soon as interactive programs are started by the users on them. Such nodes can rejoin the system when they are idle again. So it is important to handle the frequent machine entries and exits. In our design, we take the exit of a node as a failure.

4.1 System Overview

The dataflow system consists of a single *master* and multiple *workers* as illustrated in Figure 6. 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 dataflow system and waits for executing requests from the master. The shared disk space among the workers is organized by the master as a distributed dataflow storage system, which holds the vertex data during execution.

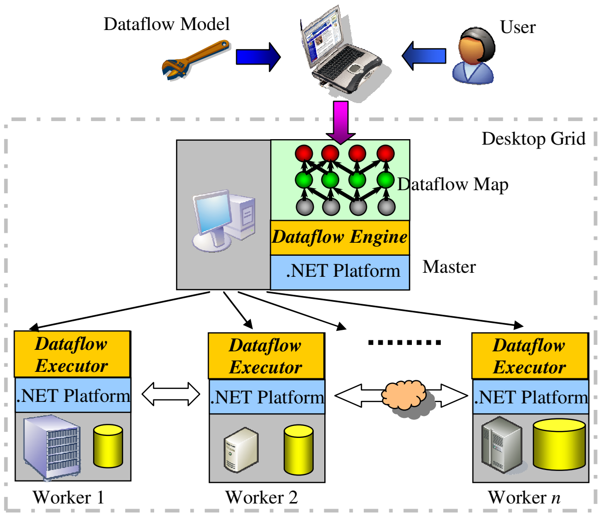


Figure 6 Architecture of Dataflow system

After the user submits a dataflow task, including initial files, data dependency graph and execution modules to the master, the master will send the initial files as initial vertices to distributed storage across workers. When the initial vertices are available from the storage, the execution of the dataflow graph is started and the master explores the data dependency graph to check ready tasks. A vertex is read, if all of its input vertices are available, and it will be dispatched to a worker for executing. Its output data will be kept in the dataflow storage and the master will schedule new ready tasks as and when new vertex data becomes available.

After the execution finishes, the result vertex data is stored in result files specified by the user. These can be collected by the user from the master.

Granularity of vertex data is important for the scheduling efficiency. The partitioning of data depends on many parameters, such as application properties and hardware environment. According to our physical settings, we suggest each vertex not be larger than 100M bytes. A vertex size that is too small will also impact the performance.

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*, *registry*, *dataflow graph* and *scheduling*.

- *Membership component*: maintains the list of available worker nodes. When some nodes join or leave the system, the list is correspondingly. It provides the list of available workers to other components on querying.
- *Registry component*: maintains the location information for available vertex data. In addition, it maintains a list of indices for available vertex data. Each vertex has an index, which lists workers that hold its data. Each time a new vertex is generated, it will be kept in the dataflow storage on some worker, and then the worker will notify the registry component the availability of the new vertex. When registry finds new available vertex, it will notify the Dataflow Graph component to explore the ready tasks. Also the registry is responsible for replicating the vertex data to improve the reliability of execution.
- *Dataflow Graph component*: maintains the data dependency graph for each job, keeps track of the availability of each vertex and explores ready tasks according to available vertices. When it finds ready tasks, it will send them to 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. While dispatching a task, the scheduler gives more weight to locality of data to reduce remote data transfers as much as possible.

4.3 The Structure of a Worker

Each worker works in a peer to peer fashion. To cooperate with the master and achieve dataflow computing, each worker involved in the dataflow computing has two functions: executing upon requests from master and storing the vertex data. Correspondingly there are 2 important components on each worker: *executor* and *storage*.

- *Executor component*: receives executing requests from the master, fetches input data from the storage component, generates output data to the storage component and notifies the 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. The storage component checks if the request could be met from the local repository. If the data requested by the executor is not locally available, it transparently fetches it from the remote worker hosting one copy through its storage component. Actually the storage components across workers run as a distributed storage service. To handles failures, upon request from master, the storage component can replicate some vertices to improve the reliability and availability.

4.4 System Interaction

Each dataflow cluster has a single master, which maintains a list of the available workers. When the worker node joins the dataflow system, it first needs to register itself to the master node. Then the Membership component on the master node monitors the availability of each worker through a heartbeat signal every 5 seconds. The heartbeat signal carries the status information about the worker, such as CPU, memory and disk usage. If master can not receive the heart beat from some worker for 3 times, it will take that worker as unavailable; if the heartbeat indicates the worker is dominated by other users, that worker is also taken as unavailable by the master. Besides heartbeats, the worker status information is also carried in the control flow data from workers to the master, such as publishing index.

After the user specifies the dataflow graph and the execution module for their computation, he can submit them with the initial vertex files as a whole task to the master node.

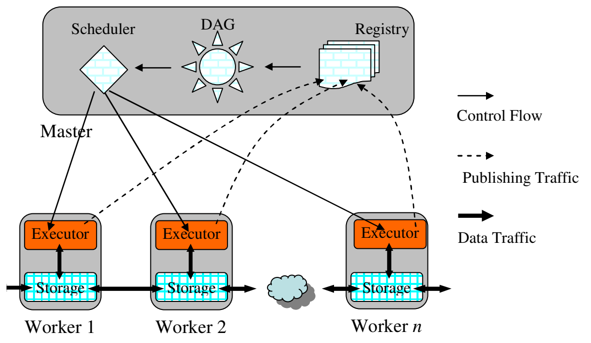


Figure 7 Components Interaction

Upon receiving submission from users, the master node will create an instance as a thread for each execution module. Based on .NET platform, the master node first load the execution module, then serialized it as an object, and finally sends to the object to workers when dispatching vertices executing as tasks.

To begin the execution of each dataflow job, master node first sends the initial vertex to workers. When a worker receives the vertex, it will first keep it in local instance of the dataflow storage, and then notifies the registry component that it has received the vertex through an *index publishing* message which carries the size of the vertex data.

After the registry component receives a publishing message for vertex *x*, it first adds an entry into *x*'s index and then checks with the dataflow graph component to check if there is a vertex execution waiting for the availability of the data specified in the publishing message. If so, the ready vertex will be scheduled as an executing task. The scheduling component sends the request of executing task to candidate workers. To choose candidate workers, the scheduling component mainly considers the location of input data to reduce bandwidth traffic and the CPU status of 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. After the execution is finished, the master collects the final results and stores them as files according to users' request.

In the current implementation, the master pushes the initial vertices to workers. After the execution starts, the worker pulls available input data from other peers (workers). That means the worker node that needs some input data will actively to fetch them. To improve the scalability, the request does not contain the input data. Upon receiving an executing request, if the input data is not kept locally, the worker need to fetch them from other workers according to the location specified in the request. By conducting data transfer in this P2P manner, we aim to increase the scalability of the system.

The storage component on each worker is responsible for maintaining the vertex data generated during execution. Whenever the executor component receives an executing request from master node, it sends a *fetch* request to the local storage component. The storage component first checks if the request can be served by local cache. If there is a local copy of the requested data, this is returned to the executor component; if not, it will contact remote storage component to fetch data remotely 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 calls the *SetResult*() to save the result vertex into local storage component and then publish an index message to notify the *registry component* on the master.

Dataflow storage maintains a cache in memory. After remotely fetching or after a SetResult(), all the vertices first will be kept in the cache and dumped to disk asynchronously when there is a need to reduce memory space. Keeping hot data in memory could improve the performance. Worker schedules the executing and network traffic of multiple tasks as a pipeline to optimize the performance.

4.5 Fault Tolerance

Since the dataflow system is deployed in Desktop Grid environments, we need to handle node failures to ensure the availability of computation. In the shared environment, we face two kinds of failure: physical failure and soft failure. Physical failure means some node cannot work for some time due to due to software or hardware problems. Soft failure occurs when a higher priority users demands node resources and the dataflow system yields. Under soft failure, the node still works, but for the time being cannot contribute its resources to the dataflow system. During soft failure, a node's CPU and network interface are no longer available to the dataflow system, but the local vertex storage is not reclaimed. When the node leaves soft failure and rejoins the system, those vertices are once again available. However, in our current design, we use same mechanisms handling soft failures as handling physical failures.

4.5.1 Worker Failure

The master node 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 one worker makes tasks which it is processing to be lost and the master needs to re-schedule the lost tasks. 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 these two kinds of failures, the master 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 and Granularity

There is a lot of ongoing research on scheduling complex DAG tasks effectively. Generally, we could borrow their results and choose suitable algorithms for our applications. In the current implementation, the scheduling of tasks is performed by the master giving priority to locality of data [22] and performance history of workers [31]. Exploring the dataflow graph, the scheduler component on the master takes each vertex as the basic scheduling unit.

To begin execution, the scheduler component distributes the initial vertices across available workers. For an efficient distribution, the size of each initial vertex data and computing power, i.e. CPU frequency, are taken as the measure for load balancing. Furthermore, the dataflow library provides ways for user to combine some vertices as a unit for distribution.

During the computation, 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. This prediction is important to achieve an efficient scheduling in heterogeneous environment.

Granularity is important for the efficiency of scheduling. Our philosophy is to use homogeneous granularity of vertices to manipulate the power in heterogeneous environment. So it is better for users to partition the initial data into homogeneous vertices with similar data size.

5. Performance Evaluation

In this section, we evaluate the performance of the dataflow system through two experiments running in a Windows Desktop Grid, which is deployed in Melbourne University and shared by students and researchers. One experiment consists of a matrix multiplication: one square matrix multiplied with another square matrix. The other experiment involves an iterative matrix vector multiplication.

These two programs are example applications for the dataflow system. Generally this kind of parallel program is performed through MPI applications. However, with dataflow system, users need not concern about specifying the explicit communications involved in MPI-based applications. Similar coarse-grained programs could also be easily handled by dataflow system.

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 storage), 1 Gbps Ethernet network and ran Windows XP.

5.2 Testing Programs

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

5.2.1 Matrix Multiplication

Each matrix consists of 4000 by 4000 randomly generated integers . Each matrix needs about 64M bytes. Each matrix is partitioned into square blocks with different granularity.

In the testing, we choose two granularities: 250 by 250 square block (16*16 blocks with 255KB per block) and 125 by 125 square block (32*32 blocks with 63KB per block).

For 16*16 blocks partition, there are 512 initial vertices for the two matrix and 256 result vertices for the result matrix. For 32*32 blocks partition, there are 2,048 initial vertices for the two matrix and 1024 result vertices as the result matrix.

5.2.2 Matrix Vector Iterative Multiplication

The matrix consists of 16000 by 16000 integers, and the vector consists of 16000 integers. All the integers are generated randomly. The matrix uses about 1GB and the vector uses 64KB. The benchmark iterates 50. The matrix and vector are partitioned by rows. Two granularities for partition are adopted in the evaluation: 24 stripes and 32 stripes.

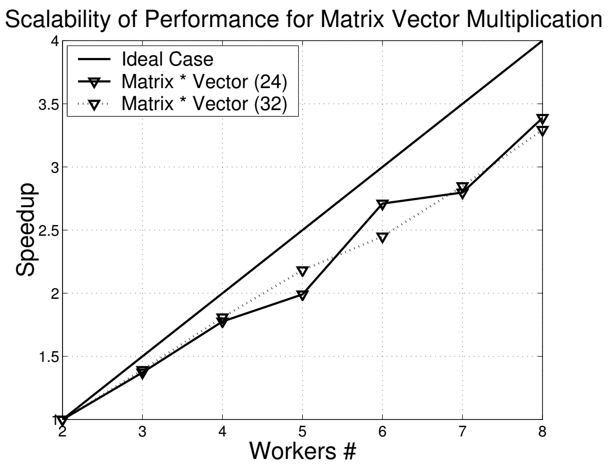
For 24 stripes, the matrix and the vector are respectively partitioned by rows into 24 pieces. Each matrix one is about 41 MB and each vector one is about 2.6 KB. There are 48 initial vertices. During the computation, 1200 vertices are generated. Finally 48 result vertices are collected as the result vector.

For 32 stripes, the matrix and the vector are respectively partitioned into 32 pieces. Each matrix one is about 31 MB and each vector one is about 2 KB. There are 64 initial vertices. During the computation, there are 1600 vertices are generated. Finally, 32 result vertices are collected as the result vector.

As example programs, the multiplication code is not specially optimized for performance purpose.

5.3 Scalability of Performance

The performance scalability evaluation does not include the time consumed for sending initial vertex data and collecting result vertex data as these two actions need to transfer data across single master which is sequential behavior.



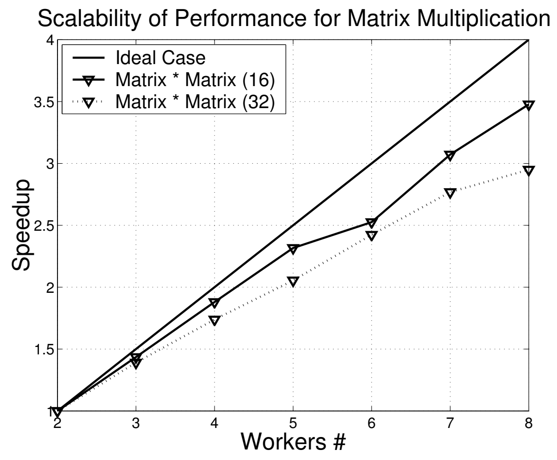


Figure 8 *Scalability of Performance*

Figure 8 illustrates the speedup of performance with an increasing number of workers. The execution time on two workers is used as the speedup baseline, because the matrix vector example cannot be run on a single worker as there is not enough memory to hold the input and intermediate vertices. 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. In most cases, the efficiency of speedup is over 80%. Table 1 shows the speedup ratio of matrix vector multiplication with 24 partitions and matrix multiplication with 256 partitions. The speedup ratio for matrix multiplication is a little higher. The reason is that the ratio of computation to communication for matrix multiplication is a little higher than vector matrix multiplication, as illustrated by Figure 9.

Worker #	3	4	5	6	7	8
M*V(24)	1.37	1.78	1.99	2.71	2.80	3.39
M*M(256)	1.44	1.88	2.32	2.53	3.07	3.48

Table 1 *Speedup Ratio*

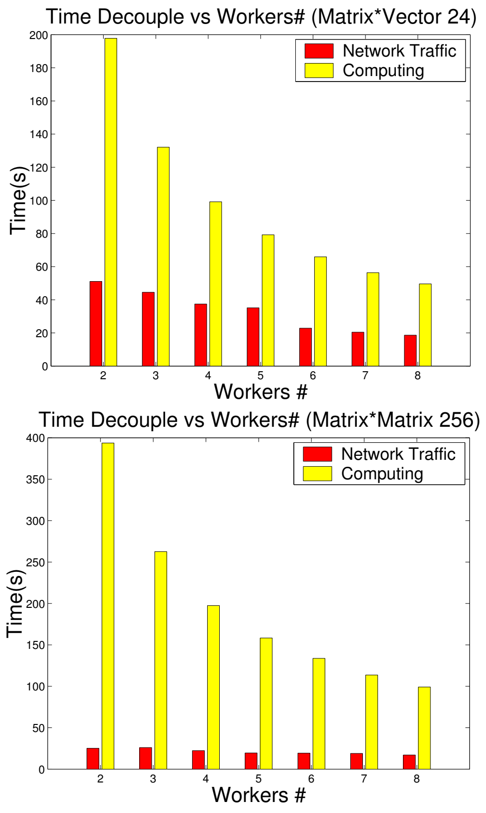


Figure 9 *Computation and Network load decouple. For each workers setting, average load decouple is shown. Matrix multiplication has bigger ratio of computation vs. network traffic.*

One expectation of partition granularity is that more partitions will introduce additional overhead during execution. Figure 8 is consistent with this, especially for the matrix multiplication application as the number of vertices for 32*32 partitions is 4 times bigger than that for 16*16 partitions for matrix multiplication, while the vertex number for matrix vector multiplication is nearly same under 24 and 32 partitions.

5.4 Impact of Replication

This section discusses the performance impacts of vertex replication. As vertex replication consumes additional network bandwidth, it will increase the time for whole computation as expected. This however depends on the number of vertices and data size for replication. We use iterative matrix vector multiplication with 24 partitions and 50 iterations. There are 1200 vertices generated with 3.1 MB as the total size.

As Figure 10 illustrates the comparative performance under 1 and 2 step replication.

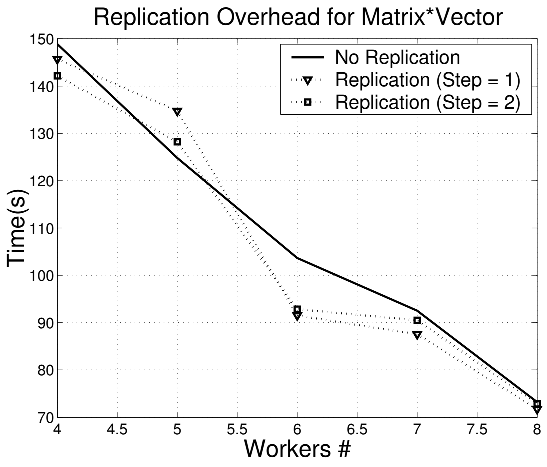


Figure 10 *Replication overhead.*

Generally the replication overhead is not big and the performance under replication is not impacted heavily. This is because the replicated data is not too big, only 3.1MB and we set the replication as low priority for contending the network bandwidth. An interesting phenomenon is sometimes the performance under replication is even better than the case without replication. This is because actively replication decreases the competition during peak traffic.

5.5 Handling Worker Failure

This section evaluates the mechanisms dealing with worker failure in the dataflow system. We use iterative matrix vector multiplication with 24 partitions and 100 iterations. In total, 2400 vertices are generated during the testing. Vertices created by the computation are nearly same size. So Figure 11 measures the number of vertices created during execution. This number is collected on the *master* node by the *registry* component. There are 8 workers and 1 master involved in the testing.

The testing compares how rescheduling and replication deal with worker failures. The testing first checks the dynamic number of vertices created by the computation without worker failures and without vertex replication as the first line illustrated in Figure 11.

The whole computation lasts about 4 minutes, depending on the dataflow configuration and physical setting. The initial phase where the line is a little inclined, is when the master node sends initial vertices to multiple workers. Because it is actually a sequential process, so the line is nearly flat. After all initial vertices are available in the dataflow storage, the execution begins and the slope of vertices number line correspondingly increases. After all of the vertices are created, the line changes to flat.

Next we add one worker failure in the testing. At first we have 1 master node and 8 worker nodes involved in the execution. During the computation, we unplug the network cable of one worker to simulate the worker failure at around the 4th minute. First we do not take any replication and then one worker failure causes some vertices to be lost, illustrated by the 2nd line in Figure 11. After the *master* node finds the failure, it will first dispatch live workers to regenerate lost vertices and then continue the execution. So in the 2nd line of Figure 11, there is a big drop at the 230th second. After a while, however the vertices number increases back again due to the re-execution to generate lost vertices. Because only 7 workers left for execution, so the slope is smaller than before the drop point.

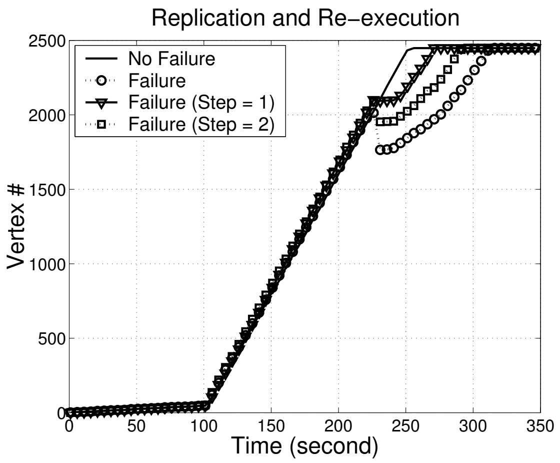


Figure 11 *Handling worker failures with replication and re-execution.*

Next we add replication mechanism to handle the failure. We test for two settings: 1 step and 2 step replication. With 1 step replication, after one worker failure, the line changes to flat, because the master needs to resend some initial vertices to continue the computation. For 2 step replication, after one worker failure, there is a small drop first and then the line changes to flat, because one worker failure makes some non-replicated vertices lost. Eventually we find replication mechanism effectively reduces the time consumed for regenerating lost vertices.

6. Conclusion and Future Work

This paper presents 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 environment.

The mechanisms adopted in our dataflow system support scalable performance and transparent fault tolerance based on the evaluation of example applications.

Next we plan to incorporate the dataflow programming model and the dataflow engine into Alchemi[1], and then extend this computing platform into larger scale distributed environment, such as Grids and P2P networks.

Acknowledgement

We would like to thank Yu Chen, Krishna Nadiminti, Srikumar Venugopal, Hussein Gibbins, Marcos Dias de Assuncao, and Marco A. S. Netto for their support during the implementation and improving the quality of this paper.

References

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

[2] Arvind and Culler, D. E.. *The tagged token dataflow architecture (preliminary version).* Tech. Rep. Laboratory for Computer Science, MIT, Cambridge, MA., 1983.

[3] Arvind and Nikhil, R. S., *Executing a program on the MIT tagged-token dataflow architecture.* IEEE Trans. Compute. 39, 3, 300–318, 1990.

[4] Berkeley open infrastructure for network computing. *http://boinc.berkeley.edu*

[5] D. Abranson, J. Giddy, and L. Kotler, *High Performance Parametric Modeling with Nimod/G: Killer Application for the Global Grid?*, IPDPS'2000, 2000.

[6] D. Thain, T.Tannenbaum, and M. Livny. *Distributed computing in practice: The Condor experience.* Concurrency and Computation: Practice and Experience, 2004.

[7] Fagg, G., Angskun, T., Bosilca, G., et al *Scalable Fault Tolerant MPI: Extending the Recovery Algorithm***,** 12th Euro PVM/MPI, 2005.

[8] Stanford Folding at Home distributed computing, *http://folding.stanford.edu/download.html*

[9] G. Bosilca, A. Bouteiller, F. Cappello, et al *MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes*. IEEE/ACM SC'02, November 2002.

[10] G. Fedak, C. Germain, V. N'eri, F. Cappello, *Xtremweb: A generic global computing system,* First IEEE/ACM International Symposium on Cluster Computing and the Grid, 2001.

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

[12] J. Bent, D. Thain, A.C. Arpaci-Dusseau et al. *Explicit control in a batch-aware distributed file system.* 1st USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2004.

[13] J. B. Dennis and D. P. Misunas, *A Preliminary Architecture for a Basic Data-Flow Processor*, The Second IEEE Symposium on Computer Architecture, 1975

[14] J. Dean and S. Ghemawat, *MapReduce: Simplified Data Processing on Large Clusters*, OSDI'04, San Francisco, CA, 2004.

[15] J. Frey, T. Tannenbaum, Ian Foster, et al. *Condor-G: A Computation Management Agent for Multi-Institutional Grids*, Proceedings of 10th IEEE Symposium on High Performance Distributed Computing, 2001.

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

[17] J. Verbeke, N. Nadgir, G. Ruetsch, I. Sharapov, *Framework for peer-topeer distributed computing in a heterogeneous, decentralized environment,* Third International Workshop on Grid Computing, 2002.

[18] Korpela, E. et al, 2001. *SETI@home-massively distributed computing for SETI.* Computing in Science & Engineering, Vol. 3, No. 1, pp. 78.

[19] L.G.Valiant. *A bridging model for parallel computation*. Communications of the ACM, 33(8):103-111, 1997.

[20] LEE, E. AND MESSERSCHMITT, D., *Static scheduling of synchronous dataflow programs for digital signal processing*. IEEE Trans. Comput. C-36, 1, 24–35, 1987.

[21] M. Stonebraker, J.Chen, N.Nathan, et al. *Tioga: providing data management support for scientific visualization applications.* In International Conference on Very Large Data Bases (VLDB), 1993.

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

[23] Ptolemy II, http://ptolemy.eecs.berkeley.edu/ptolemyII/

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

[25] R.H.Arpaci-Dusseau, Eric Anderson, Noah Treuhaft, et al. *Cluster I/O with River: Making the fast case common.* In Proceedings of the Sixth Workshop o Input/Output I Parallel ad Distributed Systems (IOPADS'99), pages 10-22, 1999.

[26] R. Kalakota and M. Robison, *E-business: road-map for success*, Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1999.

[27] S. Bowers, B. Ludaescher, A. H.H. Ngu, T. Critchlow, *Enabling Scientific Workflow Reuse through Structured Composition of Dataflow and Control-Flow*, IEEE Workshop on Workflow and Data Flow for Scientific Applications (SciFlow), 2006.

[28] S.F. Altschul, T.L. Madden, A.A. Schaffer et al. *Gapped BLAST and PSI-BLAST: a new generation of protein database search programs.* In Nucleic Acids Research, pages 3389-3402, 1997.

[29] S.Kubica, T.Robey, and C.Moorman. *Data parallel programming with the Khoros Data Services Library.* Lecture odes in Computer Science, 1388:963-973, 1998.

[30] S. Louca, N. Neophytou, A. Lachanas, P. Evripidou, *MPI-ft: Portable fault tolerance scheme for MPI*, In Parallel Processing Letters, 10(4), pp 371-382, World Scientific Company, 2000.

[31] Smith, W., Taylor, V.E. & Foster, I.T. *Using Run-Time Predictions to Estimate Queue Wait Times and Improve Scheduler Performance*, Proceedings of the Job Scheduling Strategies for Parallel Processing (IPPS/SPDP '99/JSSPP '99), Springer-Verlag, 1999, 202-219.

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

[33] T. Hey, and A. E. Trefethen, The UK e-Science Core Programme and the Grid, Journal of Future Generation Computer Systems, 18(8): 1017-1031, Elsevier, Oct 2002.

[34] W. M. Johnston, J. R. P. Hanna, and R. J. Millar. *Advances in Dataflow Programming Languages.* ACM Computing Surveys, 36(1):1–34, March 2004