A Dataflow System with a Local Optima-based Scheduling for Enterprise Grids

Chao Jin and Rajkumar Buyya

Grid Computing and Distributed System Laboratory
Department of Computer Science and Software Engineering
The University of Melbourne, Australia
{chaojin, raj}@csse.unimelb.edu.au

Abstract

This paper presents the design, implementation and evaluation of a dataflow system, including a macro-dataflow programming model, runtime system and an online scheduling algorithm, to simplify the development and deployment of distributed applications. The model provides users a simple interface for programming applications with complex parallel patterns. The associated runtime system dispatches tasks onto distributed resources through an online algorithm, called L-HEFT (Localized Heterogeneous Earliest-Finish-Time) proposed in this paper, and manages failures and load balancing in a transparent manner. The system has been implemented over a .NET-based enterprise Grid software platform, called Aneka. This paper evaluates the scalability and fault tolerance properties of the system. The results demonstrate that our L-HEFT scheduling algorithm is efficient compared to existing techniques as it introduces low overhead while making mapping decisions.

1. Introduction

Recent years, parallel and distributed computing techniques have been applied to execute e-Science [30] and e-Business [25] applications over P2P [8] and Grid computing [13] platforms. The complex nature of these distributed applications has lead to research into simplifying development and deployment over large scale distributed environments. Large scale distributed system within an organization, also called Enterprise Grids or Desktop Grids, have been pioneered by systems, such as Condor [19], XtremWeb [10], and SETI@Home [8] etc. However, the focus of these systems has been on executing embarrassingly parallel applications. With the increasing deployment of such systems, there is a need for simplifying and enabling the execution of complex parallel applications on enterprise Grids. In this context, the well-known dataflow programming model [31] shows a significant promise. We have proposed a macro-dataflow programming model [4] that (a) exploits the coarse-grained dataflow relationship in (enterprise Grid) computing processes and converts the dataflow graph into a DAG (Directed Acyclic Graph) for executing and (b) supports namespace for data generated during the dataflow execution.

To efficiently execute the macro-dataflow computation in distributed environments, we need an efficient mapping algorithm that assigns tasks in the DAG graph of dataflow to distributed

resources. Furthermore, it should be robust enough to handle the heterogeneity and frequent failures common in the target execution environment (shared enterprise Grids) containing autonomous resources/contributors. Meeting these requirements is a challenge [34] and has been extensively studied as DAG-based scheduling models/techniques, which are either static or dynamic in nature. Popular static scheduling algorithms, such as heuristic-based HEFT (Heterogeneous Earliest-Finish-Time) [11] and genetic search [1], map DAG tasks to distributed resources prior to the execution. Such static methods do not work effectively for dynamic distributed environments where the availability of resources and their capability varies dynamically at runtime.

On the other hand, dynamic scheduling methods can make the mapping decision for ready tasks during the execution. One simple solution applied in Condor-G [17], called *just-in-time* scheduling, normally puts little weight on the overall optimized mapping and therefore it is not good at efficiently utilizing global resources. Another method is to start with a static scheduling plan, and then iterative rescheduling is performed for adaptation to resource changes [33]. However, they need to have the detailed knowledge of the whole graph and their scheduling cost could be high for large scale graphs with thousands of tasks [24]. Furthermore, most heterogeneous scheduling algorithms give little weights on efficient failure handling.

We propose a macro-dataflow system with a *Localized Heterogeneous Earliest-Finish-Time* (L-HEFT) scheduling algorithm especially for heterogeneous environment with frequent failures. In contrast with previous methods, our adaptive scheduling algorithm focuses on optimizing the scheduling efficiency based on the available partial part of the graph which is gradually generated during the execution and works in an online manner. Compared with iterative static mapping-based rescheduling methods, our algorithm introduces low overhead in managing schedules and execution in distributed environment with dynamic resources. Furthermore, it delivers nearly the same performance result as the static mapping-based rescheduling methods and even outperforms rescheduling methods for dataflow applications of a large size of symmetric graph with balanced tasks distribution. In addition, our macro-dataflow system naturally supports replication-based fault tolerance mechanism for intermediate data generated during dataflow execution, which simplifies the failure handling of our adaptive scheduling algorithm.

The main contributions of this work are: (1) an architecture and runtime machinery of a dataflow system with a simple and powerful macro-dataflow programming model, which supports the composition of parallel applications for transparent deployment in a heterogeneous distributed environment; (2) a L-HEFT heuristic online scheduling algorithm; (3) evaluating the scalability of our system and the performance of the scheduling policy with real applications in an enterprise Grid environment. As our experiments illustrate, our system is effective in addressing parallelism of data dependency in real applications and our scheduling policy can achieve same performance target as existing dynamic policy with scheduling cost that is 30% to 50% of existing static mapping-based rescheduling policy.

The remainder of this paper is organized as follows. Section 2 provides a discussion on the related work. Section 3 presents the architecture and design with an implementation of the macro-dataflow system. Section 4 presents L-HEFT scheduling algorithm for heterogeneous and un-reliable distributed environment. Section 5 discusses the experimental evaluation of the system. Section 6 concludes the paper with pointers to future work.

2. Related work

The dataflow concept was first presented by Dennis et al. [16] [31] and has led to a lot of research, such as the dynamic dataflow model [3] and the synchronous dataflow model [9]. Currently, 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 [22].

Grid computing platforms such as Condor [17][6], Entropia [2], Pegasus [7], ASKALON [29], provide mechanisms for workflow scheduling. Condor works at the granularity of a single job. Existing tools, such as DAGMan, can schedule jobs with data dependencies and address the parallelism between tasks. Condor does not focus on the programming difficulties associated with the data communication between tasks, but emphasizes on the high level problem of matching the available computing power with the requirements of jobs. For example, within each job, users mainly depend on message passing interface for programming, such as MPI. Therefore users must take extra care with data sharing conflicts, deadlock avoidance, and fault tolerance. Pegasus [7] works on a higher level than DAGMan, and deploys heuristic scheduling policy for scheduling the DAG graph of jobs rather than the *just-in-time* scheduling policy in DAGMan.

Grid Superscalar [26] aims to simplify the development of Grid applications with a different method, wherein users can write sequential program within small tasks and parallelism between tasks is discovered through analyzing the dependency of input and output files for tasks. However scheduling and fault tolerance are not the focus of Superscalar. Kepler [27] is a scientific workflow system that allows composition of both data and control flows. It also provides a graph interface for programming. Compared with Kepler, our macro-dataflow model (with a language interface) is more suitable for programming applications with a large number of repetitive tasks. Furthermore, we adopt a different fault tolerance mechanism, and the scheduling algorithm in heterogeneous distributed environment is not emphasized by Kepler.

To effectively schedule tasks of DAG graph to Grid environments with dynamic features, many dynamic scheduling strategies have been proposed to map tasks to resources during execution. One simple choice is called just-in-time scheduling, for example, Condor-G [17] and Virtual Grid [24] deploy a greedy matching algorithm to decide the mapping of each task during the execution. This category of policies is difficult to achieve overall optimized mapping and efficiently utilizes global resources.

To improve the efficiency, the adaptive rescheduling method starts with a static scheduling plan, and then depends on iterative rescheduling for adaptation to resource changes [33]. Although these policies can potentially achieve optimized overall efficiency, they need to know the detail knowledge of whole graph and their scheduling cost could be high for large scale graphs with thousands of tasks [24]. The plan switching method [12] can construct a family of activity graphs beforehand and investigate the means of switching from one member to anther when the execution of one activity fails. However, all of the plans are limited within the most updated information of resources, which does not take the future changes into consideration.

We propose a localized heuristic policy, which does not requires the knowledge of global graph, and therefore, has no extra cost of rescheduling. At the same time, we can achieve nearly the same performance target and even outperform the rescheduling methods for large scale dataflow applications with a balanced symmetric DAG graph.

3. Architecture and Design

This section describes the dataflow system which supports the execution of our macro-dataflow graph. Please refer to the Appendix on our macro-dataflow APIs for composing a macro-dataflow graph. The target environment of our dataflow system is a shared enterprise Grid consisting of commodity PCs, where PCs can drop out of the system as soon as being dominated, turned off or restarted by interactive users. Such nodes can rejoin the system when they are idle again. The design goal aims to make our system adapt to the resource heterogeneity, including transient or static, easily incorporate new resources and handle failures. For adaptation to heterogeneity, we deploy online heuristic scheduling algorithm with assistance of a performance prediction algorithm [Section 4.2] based on historical data. To handle failures, we organize the large scale of free disks over PCs as a virtual storage pool and hold intermediate data generated during dataflow execution as the resuming point in handling failures. This section first presents the architecture of our dataflow system, and next section describes our scheduling algorithms in detail.

3.1 System Overview

Components of dataflow system include *coordinator* and *contributor*, as illustrated in Figure 1. The coordinator is responsible for accepting jobs from users, organizing contributors to work cooperatively. For example, it monitors availability of resources, send executing requests to contributors, and handle failures of contributors, etc. Each contributor joins the dataflow system through contributing CPU, memory and disk resources, and then passively waits for requests from coordinator. Both coordinator and contributor are implemented as a pluggable service component in Aneka [32], which is a .NET-based enterprise Grid software platform and can support the creation of enterprise Grid environment. We also utilize existing Grid services in Aneka to simply our implementation, such as Resource Monitor Service.

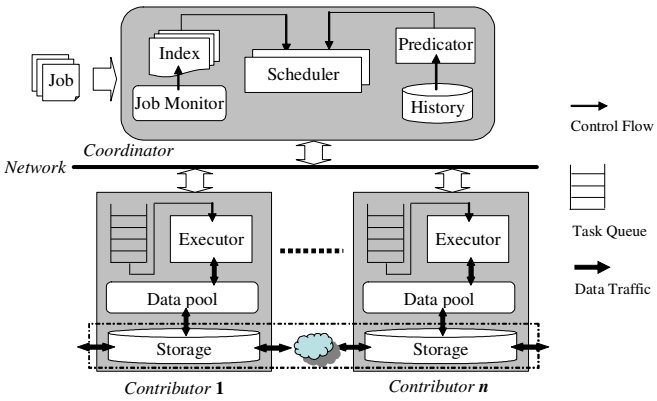

Figure 1 Architecture of Dataflow System

3.2 Structure of Coordinator

Coordinator consists of a set of key sub-components, including *job monitor*, and *scheduler*, database of *performance history*, *performance predicator*, and *index* of intermediate data. A *scheduler* is instantiated for each job and adopts an online scheduling policy to map ready tasks to suitable contributors for executing. The historical information of execution is recorded in the database of *performance history* component, which can be used by a *predicator* component to predict the performance of tasks. The *job monitor* maintains the dataflow graph for each job, keeps track of the intermediate data generated during the execution, and explores ready tasks for scheduling. The *index* component maintains the location for available intermediate data.

Normally each intermediate data stays in memory on the contributor where it is generated. In order to improve the reliability of execution, however, the *index* can choose when and where to make the intermediate data persistent on disk or replicated to other contributors.

3.3 Structure of Contributor

Each contributor contributes local resources for dataflow computing with a task queue to buffer commands of coordinator. Due to the large disk drive in current popular desktops, contributors in a dataflow system actually have a significant amount of free disk space. The free disk space available at the contributors is organized by the coordinator as a virtual storage pool, which can hold the intermediate data generated during dataflow execution to improve the availability of computation and handle transient or permanent departure of contributors as well. Furthermore, there are the following important sub-components on each contributor: *executor*, *data pool* and *storage*.

- *Executor*: fetch executing commands from the task queue, execute the tasks and put the output data into local *data pool*. *Executor* requests input data for tasks from *data pool*.
- *Data pool*: maintains the intermediate data generated by dataflow in memory, and meet the request of input data from the *executor*. If the request is missed locally, the data pool will notify the *storage* component to fetch the requested data from other contributors according to the location in command. When the *data pool* finds that allocated memory is nearly full, it can swap data in memory to disks through the *storage* component. Another matter is, in order to efficiently handle failures, the *data pool* may also swap those data not needed by the remainder dataflow execution to the *storage* component for persistent maintenance until the whole job is finished, rather than simply removing them.
- *Storage component*: works as a backup cache for data pool, and at the same time is responsible for managing persistent intermediate data, which may be generated for reliability purposes. The local storage component can communicate with the storage component on remote contributors to transfer data, which is transparent from the point view of the executor. Actually storage components across contributors constitute a virtual storage that is especially designed for holding persistent intermediate data for dataflow with a flat name space. To handle failures, upon request from coordinator, the storage component can replicate requested intermediate data on the remote side to improve reliability and availability.

3.4 Replication Support

With the cooperation between the *index* component on coordinator and the *storage* component on contributors, our dataflow system can replicate intermediate data generated during the dataflow execution to multiple contributors. The replication works in a lazy manner, which just replicates the copy of intermediate data if there are contributors found not to be busy. Rather than replicating intermediate data for tasks in every level [4.3], we replicate data associated with tasks in every n levels. The replication step size, n, can be specified by users during job submission. In order to achieve execution in the face of failures, some of the intermediate data may have to be re-generated. This requires identifying the finished tasks to be re-executed to regain the lost data. Therefore, we need to explore tasks which should be re-executed to generate the intermediate data necessary to resume the execution. This exploration stops until we find replicated copies of lost intermediate data or we reach the initial tasks.

4. Scheduling Policy

This section describes in detail our dynamic scheduling algorithms on mapping ready tasks of the dataflow graph to heterogeneous resources in a shared enterprise Grid environment.

Our macro-dataflow programming model aims at scientific applications, which consist of many repetitive tasks. To support adaptation, repetitive tasks are partitioned into fine granularity, and as a result, the number of tasks is much larger than the number of resources. Taking the large number of tasks into consideration, our dynamic policy aims to efficiently map dataflow tasks onto heterogeneous resources with frequent changes. Our policy optimizes the scheduling efficiency based on available partial part of the graph which contains ready tasks gradually generated during the execution. Our policy works in two phases. In the first phase, it partitions the tasks in the graph into clusters and to make tasks within same cluster as independent. This partition phase can be deployed prior to the execution time or during execution. In the second phase, our policy achieves a local optimization on scheduling of ready tasks with priority on reducing data migration using our L-HEFT heuristic algorithm.

Compared with just-in-time dynamic policies, our method aims to decrease the unnecessary data movement between resources through online analysis, which is especially important for data-intensive application [15]; and at the same time, it does not require the phase of complex rank assignment, which is the prerequisite for global optimization policies, and is always based on inaccurate estimation of data transfer and computing cost. To cope with the repetition property of tasks in application, we adopt a performance prediction algorithm to improve the efficiency of L-HEFT scheduling, which is based on the historical performance information.

4.1 Scheduling Model

With our macro-dataflow programming model, the dataflow graph is converted into a directed acyclic graph (*DAG*). Given a DAG, $\boldsymbol{G}=(\boldsymbol{V}, \boldsymbol{E})$, the set of vertices $\boldsymbol{V} = \{v_1, v_2, .., v_n\}$ represents the set of tasks to be executed, and the set of directed edges $\boldsymbol{E}$ represents communication between tasks, where $e_{ij} = (v_i, v_j) \in \boldsymbol{E}$ indicates communication from task v_i to v_j. We call each communication data as a *stream* and user can specify a unique name for each stream. *Initial streams*, which are not generated by any tasks, are actually mapped to external files, e.g. the input for dataflow execution. *Result streams*, which have no receiver tasks, are the results of dataflow execution. With the name of each stream, user can edit execution tasks and configure its input and output streams. We also call each execution task as a *token*. For each token, if all of its input streams are available it is ready to execute.

4.2 Performance predication

In dataflow execution, different tasks may share the same execution instructions. To predict execution time of *token.Execute* on contributor *r*, we use Equation 1. $E_i(r)$ is the time of *i*-th execution of v_i*.Exec* on contributor *r* and I_n is the size of corresponding input stream. α is a value selected between 0 and 1. A larger value of α gives more weights to recent executions and Equation 1 also considers the weight of input size as illustrated.

$$P(v_i.Exec, r) = \left(\alpha * \frac{E_n(r)}{I_n} + (1-\alpha) * \frac{E_{n-1}(r)}{I_{n-1}} + \cdots + (1-\alpha)^i \frac{E_{n-i}(r)}{I_{n-i}} + \cdots \right) * v_i.input\_size \quad (1)$$

If v_i*.Exec* has not executed on contributor *r*, to predict the execution time of task v_i on contributor *r*, we use the average of prediction on all other contributors who have executed

$v_i.Exec$, as Equation 2, where $S(v_i.Exec)$ is the set of contributors who have executed $v_i.Exec$ and $E(r_i)$ is the execution time of $v_i.Exec$ per byte.

$$P(v_i.Exec, r) = \left[\sum_{r_i \in S(v_i.Exec)} E(r_i) \right] * v_i.input\_size \quad (2)$$

If it is the first time to execute $v_i.Exec$ in all of the available contributors, we use the prediction value provided by user.

4.3 Level-based Clustering

The algorithm used in the first phase is a technique for ordering the nodes based upon their precedence constraints, called level sorting, which have been adopted by many priori works [21][29]. We can define the level sorting in a recursive way. Given a directed acyclic graph $\boldsymbol{G}=(\boldsymbol{V}, \boldsymbol{E})$, level 0 contains all vertices v_j such that there is no vertex v_i with $e_{ij} \in \boldsymbol{E}$(i.e. v_j does not has any incident edges). Level k consists of all vertices v_j such that, for all $e_{ij} \in \boldsymbol{E}$, every vertex v_i is in a level less than k and at least one vertex is in level k-1.

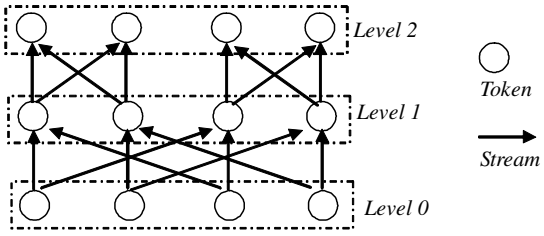

Figure 2 Dataflow graph of FFT with 4 points

The result of clustering is to partition $\boldsymbol{G}$ into L blocks numbered consecutively from 0 to L-1, and execution tokens within each block are independent, i.e. there is no data precedence constraint between them. All tasks that send data to a task in block k must be in any blocks 0 to k-1; for each task v_j in block k, there exists at least one stream from task v_i in block k-1. Block 0 contains all initial tasks whose input streams are initial streams. Figure 2 shows the result of clustering for a FFT dataflow graph.

This partition phase can be done during the execution of the dataflow graph.

4.4 A Localized HEFT algorithm

Our aim is to minimize the execution time of tasks within each block, and as a consequence, the overall execution of whole dataflow graph could be potentially optimized. We propose a scheduling algorithm which is called as L-HEFT (Localized Heterogeneous Earliest Finished Time) algorithm. The HEFT (Heterogeneous Earliest Finished Time) algorithm [11] is a static scheduling algorithm which can potentially achieve an overall optimized mapping with a relative low cost. HEFT first assign a rank to each task through recursively traversing its successor tasks and computing the weight based on predicted performance and network traffic until result task is reached. After that, HEFT dispatches each task to resources which can finish it fastest according to the rank order. Therefore it needs global knowledge of whole graph and execution environment. On the contrary, L-HEFT algorithm does not require the global knowledge of whole graph for the complex ranking phase as HEFT and aims to optimize the mapping of local ready tasks in currently available tasks of partial graph.

Since we cannot optimize the scheduling in the manner of global mapping, this may lead to some unnecessary data traffic and may not give more weights to tasks on the critical path. So our policy puts more priority on data location as compensation and use the

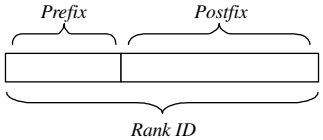

Figure 3 Rank ID

increasing order of level number to ensure the priority from dependency constraints. Our algorithm is focusing to decrease the data migration between resource nodes as much as possible while distributing work load across resources according to their ability. Given task *t*, we do not schedule it immediately when it is just ready. On the contrary, we put it in a schedule queue. When the queue buffers enough tasks or there are contributors found soon to be idle, the L-HEFT will be invoked.

$$\begin{cases} rank(v_j).prefix = v_j.level\_number \\ rank(v_j).postfix = \left[\sum_{i \in pred(v_j)} sizeof(c_{ij}) - \max_{n_k \in OS(v_j)} \left(\sum_{C_{ij}.owner = n_k} sizeof(e_{ij}) \right) \right] \end{cases} \qquad OS(v_j) = \bigcap_{i \in pred(v_j)} e_{ij}.owner \tag{3}$$

We first assign the priority of each ready task according to its level number. A task in a lower level has higher priority than a task in a higher level. Within the same level, however, we give high priority to the task which can be mapped to a contributor where its execution does not need data traffic over network. For those tasks, whose execution definitely needs data communication whatever node it will be assigned to, we deploy an EFT (Earliest Finished Time) heuristic algorithm to assign its execution owner. As a result, each contributor in the resource pool has a schedule queue, which holds the assigned tasks waiting for execution.

Each task is assigned with a rank ID, which consists of prefix and postfix parts as illustrated in Figure 3. The prefix part is the level number. The postfix part actually means the possible minimal traffic size if the corresponding task is executed. Equation 3 illustrates the ranking function used by L-HEFT. Given a task v_j and its predecessor task v_i, its input stream is noted as e_{ij}. $OS(v_j)$ is the set of contributors which holds e_{ij}.

Figure 4 shows the algorithm of L-HEFT heuristic. We assign a rank for each ready task using equation (3), and then we sort the ready tasks by increasing order. For tasks which have same rank priority, we sort them through predicated execution time by non-increasing order. This phase of assigning rank and sorting tasks is totally different from HEFT algorithm, and we do not require the phase of recursively traversing to calculate successor's network traffic and their estimated average computation costs. In the mapping phase, we first assign tasks which may not need network traffic for execution to the contributor which holds all of requested input streams, and then assign

```
produce Level-HEFT-Schedule(T, R, F, C)
/* T ← The list of ready tasks to schedule
   R ← The list of currently available contributors
   F ← The set of priority task appending Queue of
currently available contributors
   C ← Current time point
*/
   foreach t_i in T
      Compute t_i.rank with the algorithm of Equation (3)
   endfor
   Sort each ready task, t_i, in T by increasing order of t_i.rank
   while there are unscheduled tasks in T do
      Choose the first task t_0 in T
      if(t_0.rank.postfix is zero)
         foreach r_j in R
            Compute its traffic size s_j, if t_0 is assigned to r_j
         Endfor
         Append t_0 to f_m , where s_m=max{s_j} (f_m∈ F)
      else
         foreach r_j in R
            eft[r_j] =EFT (t_0, r_j, C, f_j, R)
         endfor
         Append t_0 to f_m , where eft[r_m]= min{eft[r_j]}
      endif
   endwhile
return F
```

Figure 5 Level-based HEFT Algorithm

other tasks to the contributor who can finish them earliest, based on calculate *EFT*(Earliest Finish Time).

During the execution, to compute *EFT* for task t on contributor r, we need to know the execution time of waiting tasks on r, which are waiting for execution. As indicated in Section 3, each contributor has a priority tasks queue which contains all assigned tasks. On the side of coordinator, there is also a queue, f_r, for each contributor recording sent tasks and their estimated execution time. When a task is finished on contributor r, f_r will be updated correspondingly to correct the estimated execution time of tasks on r. The time point of correction is recorded as $p(f_r)$. We use f_r to compute the *EST* (Earliest Start Time) for t on contributor r, so that $EST(t, r) = left\_time(f_r)-(C- p(f_r))$, where C is the current time. Therefore, $EFT(t, r) = EST(t, r) + Predict(t, r)$, where $Predict(t, r)$ is our algorithm to predict the execution time of task t on contributor r, which is based on the history execution information as indicated in Section 4.2.

To trigger the starting of our scheduling model, we deploy a greedy policy on initial tasks in block 0. During the scheduling, to prevent an the worst case where some nodes hold a schedule queue with an estimated time much longer than other queue, there is a thread running in the background which frequently checks the length of scheduling queue of each contributor. It will re-assign some tasks from the tail of the longest scheduling queue to other contributors having light load.

4.5 Handling New Resources and Failed Resources

When new resources are found in the system, the list of available resources will be updated to include them after they are ready to join the dataflow execution. Then L-HEFT scheduling algorithm will be invoked to map ready tasks in the queue to resources, including the new resources.

When there are contributors found to depart the system, it is possible that a number of intermediate data is lost due to the departed contributors. If these lost intermediate streams are necessary for continuing the execution, *Job monitor* component on the coordinator will explore to re-execute corresponding tokens in order to re-generate the lost intermediate streams. The tasks to re-execute will be put into the task buffer and wait for scheduling of L-HEFT algorithm.

5. Performance Evaluation

We have implemented our dataflow programming model, system and scheduling algorithm over the Aneka [32] platform and deployed it in an environment consisting of desktop machines from different laboratories in Melbourne University, and shared with students and researchers. In this section, we evaluate the performance of our dataflow system and L-HEFT online scheduling algorithm through three applications. The first simple one is matrix multiplication; the other two complex ones are FFT (Fast Fourier Translation) computation and Jacob iteration [20].

5.1 Environment Configuration

The experiments are executed in a enterprise Grid consisting of 33 nodes drawn from 3 student laboratories. During testing, one machine works as *coordinator* and the others work as *contributors*. 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 Sample Applications

We implemented three same applications using our macro-dataflow APIs indicated in Appendix.

1) **Matrix Multiplication:** Each matrix consists of 4000 by 4000 randomly generated integers. Each matrix needs about 64M bytes. Each matrix is partitioned into 250 by 250 square blocks, and therefore there is a total of 16*16 blocks with 128KB per block. There are 1,024 initial streams in dataflow graph for the two matrix and 512 result streams as the result matrix.

2) **FFT (Fast Fourier Transform):** This algorithm is widely used in digital signal processing and can also be used to solve Discrete Poisson Equation for physical simulation. Figure 2 is a typical dataflow graph of FFT computing in a small scale. The input of FFT example used in the experiment is 16M complex number, and is uniformly divided into 64 pieces. Therefore, there is a total of 1,664 tokens to execute.

3) **Jacob Iteration:** Jacob method is a simple way to solve PDE (Partial Differential Equations). Its iteration pattern of parallelization is shared by a large number of numerical programs and more complicated PDEs. The working space of Jacob iteration in our experiment is a 16,384 by 16,384 matrix. The matrix is partitioned by rows into 64 pieces. During the experiment, we varied the ratio of computation vs. communication and iteration times.

5.3 Scalability of System

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

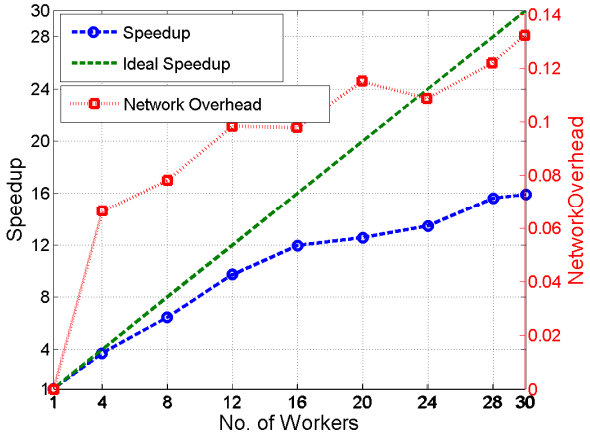

Figure 6 Scalability of Performance

Figure 5 illustrates the speedup of performance with an increasing number of coordinators. There are 2 main factors that determine the execution time of the matrix multiplication: the distribution of blocks between the contributors and the overhead introduced by the transmission of blocks between the contributors. The network overhead is measured here as the ratio of the time taken for communication to the time taken for computation. As can be seen from Figure 5, for larger number of contributors, while the speedup improves, the network overhead is also substantially increased. The speedup line starts diverging from the ideal when the network overhead increases to more than 10 % of the execution time.

5.4 Scheduling Policy

This section evaluates our L-HEFT scheduling policy. We compare it with a dynamic scheduling model [33] through rescheduling on static HEFT mapping, that we term here as D-HEFT. We have implemented D-HEFT as mentioned in prior work [33], and rescheduling is triggered when the performance of resource is changing. In our implementation, the rescheduling is overlapped with the executing of tasks. This means that until the remapping of tasks is completely finished, they are still submitted to contributors to which they were mapped in the prior iteration of

rescheduling. In this section we compare these two scheduling models with varying the ratio of computation vs. communication and the size of dataflow graph through Jacob iteration and FFT benchmarks. For Jacob iteration, every token executes same set of instructions. So we choose α in equation 1 as 0.9. If the real execution time is different from the predicted value by a factor of 2, we take it as a performance variation and correspondingly trigger rescheduling in D-HEFT.

First, we look at the result of these two polices on dataflow graph with different size. We use a Jacob iteration benchmark with 10 iterations and 100 iterations. Therefore the corresponding dataflow graph respectively holds 640 and 6400 tasks. As illustrated in Figure 7, L-HEFT scheduling policy does not compete with D-HEFT policy, while in Figure 8, L-HEFT marginally outperforms D-HEFT. The reason is the scheduling cost of D-HEFT is larger than that of L-HEFT, due to frequent variations in the performance availability of resources across contributors. For a large dataflow graph, rescheduling cost of D-HEFT is even higher. Figure 9 illustrates the total execution time of L-HEFT during the scheduling, while Figure 10 shows the total rescheduling cost of D-HEFT.

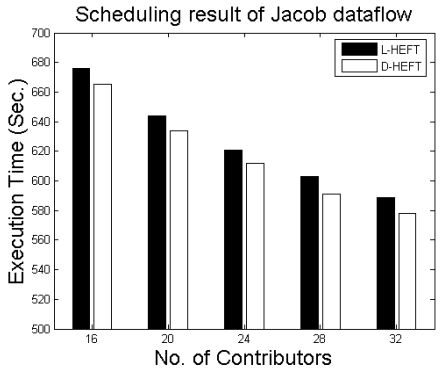

Figure 7 Scheduling on Jacob DAG with 640 tasks

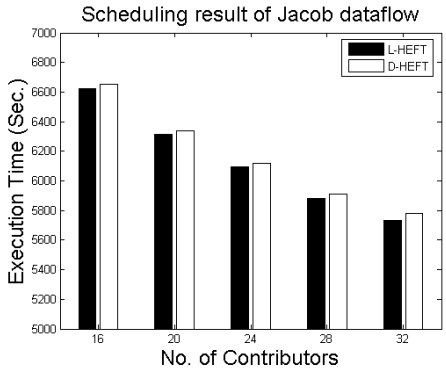

Figure 8 Scheduling on Jacob DAG with 6400 tasks

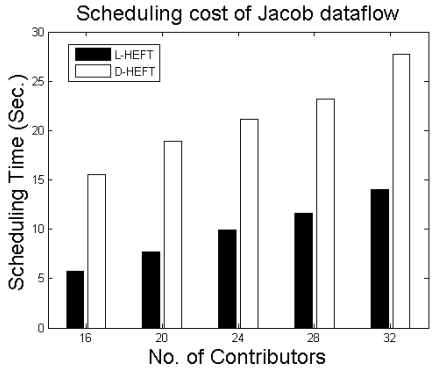

Figure 9 Scheduling cost of Jacob DAG with 640 tasks

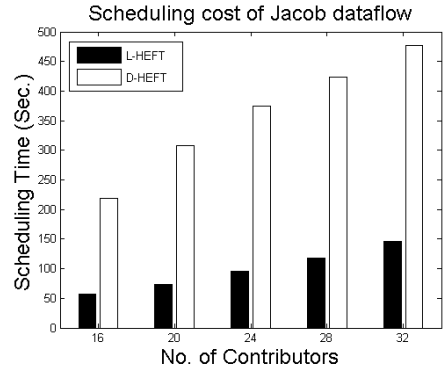

Figure 10 Scheduling cost of Jacob DAG with 6400 tasks

We use a simple model to explain why the rescheduling cost is higher. According to [11], the scheduling cost of HEFT algorithm is $C_H = O(e \times q)$, where e is the average number of edges and q is the number of contributors. For rescheduling, the cost depends on the size of partial graph. We assume the size of partial graph for each time of rescheduling is half the whole graph on average. Therefore, each time of rescheduling cost is $C_r = C_H / 2$. Given n_r as the number of rescheduling, total cost of rescheduling is $(n_r \times C_r)$. However, for L-HEFT algorithm, we can know its cost $C_L < C_H$. From Table 1, we can see that the number of rescheduling is pretty high

and in each time of rescheduling, the D-HEFT algorithm needs to assign ranks for left over tasks and then sort the tasks for re-mapping. A large number of repeated rescheduling introduces high scheduling cost. As a result, L-HEFT can outperform D-HEFT for large scale dataflow graph.

Contributors#	16	20	24	28	32
640 tasks	82	102	62	58	63
6400 tasks	913	987	659	752	732

Table 1 The number of rescheduling operations in D-HEFT

Next, we compare two scheduling polices with varied computation-to-communication ratio for a Jacob dataflow of 10 iterations. During the experiment, we adjust the ratio of computation to communication from 2 to 12 as illustrated in Figure 11. We can see that for application with a larger ratio of computation to communication, D-HEFT performs better than L-HEFT. The reason is the rescheduling cost is gradually compensated by the large execution time of tasks.

Furthermore, we compared the amount of stream data migration between contributors under the scheduling of L-HEFT and D-HEFT, as in Figure 12. The result shows that for the Jacob iteration application, both L-HEFT and D-HEFT generate nearly same amount of data migration.

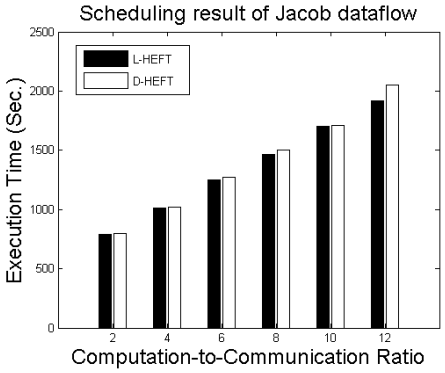

Figure 11 Scheduling with varied ratio of Computation-to-Communication

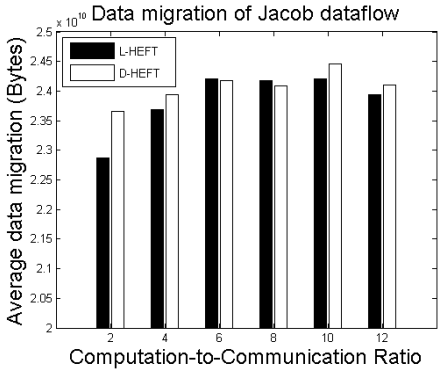

Figure 12 Average amount of data migration between contributors

Finally we run the FFT benchmark, whose communication pattern is more complex than that of Jacob. The result is showed in Figure 12. For this FFT benchmark, the ratio of Computation-to-Communication is about 3. The result shows L-HEFT can compete with D-HEFT. This result is consistent with the scheduling result of Jacob dataflow, because the task number in FFT dataflow is not large enough, only 1664.

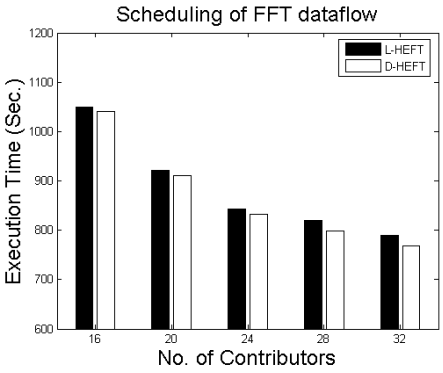

Figure 13 Scheduling of FFT dataflow

5.5 Handling Joining Contributors

This section compares the two dynamic models during handling new resources. We use a Jacob dataflow with 10 iterations and FFT dataflow. In the experiments, we first start with 16 contributors and after 3 minutes, we gradually add 2 new contributors every minute. We measured the finished task number on the side of coordinator. The slope of measure curves will increase during continuous joining of new resources, because more contributors can accelerate to execute

ready tokens, as illustrated in Figure 14 and Figure 15. These two figures show the response time of L-HEFT algorithm to handle new joining resources are a bit faster than D-HEFT. The reason is the low cost of L-HEFT algorithm.

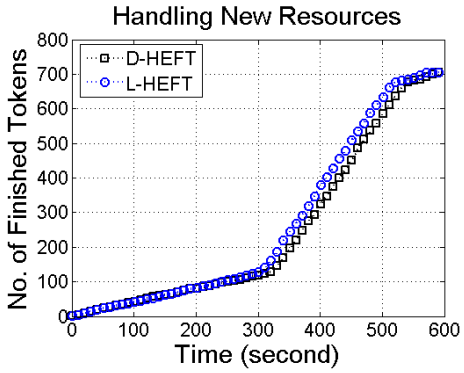

Figure 14 Handling new resources for Jacob dataflow

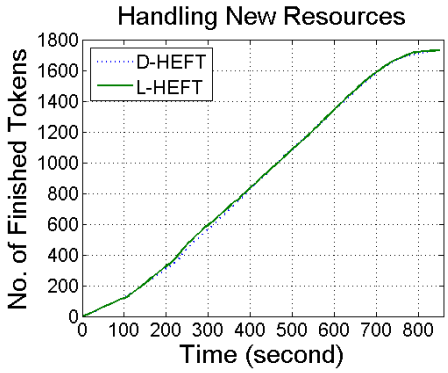

Figure 15 Handling new resource for FFT dataflow

5.6 Handling Failures of Contributors

This section evaluates the mechanisms dealing with failures of contributors in the dataflow system. We use the Jacob iteration example with 40 iterations across 20 contributors. On the coordinators side, we measure the number of finished tokens. If lost intermediate data need be re-generated by re-executing those tokens due to the failure of contributors, we just reset those tokens as un-finished. L-HEFT algorithm evaluated in this section is assisted by the replication support with the size of replication step as 5, while D-HEFT does not support failures through replication methods. After the system run for 12 minutes, we manually turn off one contributor to simulate one node failure. Without replication support, D-HEFT has to re-execute all tasks to re-generate lost intermediate data to continue the execution. However, the number of tasks who have to be re-generated by L-HEFT is pretty smaller. As Figure 15 shows, with failure support from the replication mechanism of macro-dataflow system, L-HEFT outperforms D-HEFT. Therefore, the final performance of L-HEFT is better.

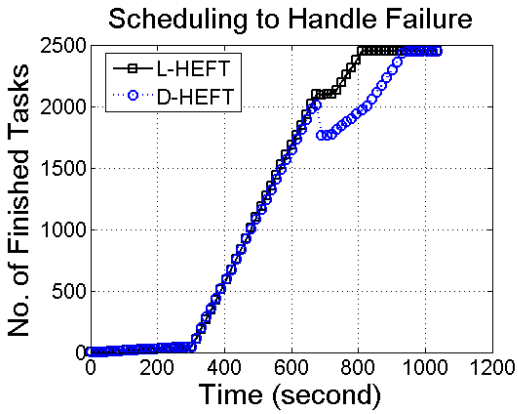

Figure 16 Fault tolerant scheduling.

6. Conclusion and Future Work

This paper presents a dataflow computing platform within a shared enterprise Grid environment. Through a macro-dataflow interface, users can freely express their parallel applications through specifying the dataflow relationship and easily deploy applications in a heterogeneous distributed environment with failures. The L-HEFT scheduling algorithm proposed for our dataflow system achieves effective mapping with fairly low cost due to heavily decreased rescheduling cost, compared with static mapping-based rescheduling techniques. At the same time, it supports

scalable performance and transparent fault tolerance based on the evaluation of example applications.

Our future work focuses on extending dynamic scheduling policy to support advanced user QoS (quality of service) requirements by building on Aneka's advanced resource reservation and service level agreement capabilities.

Acknowledgement

This work is partially supported by research grants from the Australian Research Council (ARC) and Australian Department of Education, Science and Training (DEST). We thank Srikumar Venugopal, Suraj Pandey and James Broberg for their comments on improving the quality of the paper.

References

[1] A. Y. Zomaya, C. Ward, and B. Macey, *Genetic Scheduling for Parallel Processor Systems: Comparative Studies and Performance Issues*. IEEE Transactions on Parallel and Distributed Systems, 10(8):795-812, August 1999.

[2] A. Chien, B. Calder, S. Elbert, K. Bhatia, *Entropia: Architecture and Performance of an Enterprise Desktop Grid System*, Journal of Parallel and Distributed Computing, 63(5), Academic Press, USA, May 2003.

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

[4] C. Jin, R. Buyya, L. Stein, and Z. Zhang, *A Dataflow Model for .NET-based Grid Computing Systems*, Proceedings of the 3rd International Workshop on Grid Computing and Applications, June 6-7, 2007, World Scientific Press, Singapore.

[5] 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.

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

[7] E. Deelmana,G. Singha, and M.Sua etal, *Pegasus: A framework for mapping complex scientific workflows onto distributed systems*, Scientific Programming 13 (2005) 219–237.

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

[9] E. Lee, and D. Messerschmitt, , *Static scheduling of synchronous dataflow programs for digital signal processing.* IEEE Trans. Comput. C-36, 1, 24–35, 1987.

[10] 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.

[11] H. Topcuouglu, S. Hariri, and M.-Y. Wu, *Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing*, IEEE Transactions on Parallel and Distribution Systems, 13(3):260-274, 2002.

[12] H. Yu, D. Marinescu, and A. Wu, et al, *Plan switching: An Approach to Pan Execution in Changing Environments*, Proceedings of the 20th International Parallel and Distributed Processing Symposium, Greece, April 25 -29, 2006.

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

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

[15] J. Blythe, S. Jain, and E. Deelman, et al, *Task Scheduling Strategies for Workflow-based Applications in Grids*, IEEE International Symposium on Cluster Computing and Grid (CCGrid), 2005.

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

[17] 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.

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

[19] M. Litzkow, M. Livny, M. Mutka, Condor - A Hunter of Idle Workstations, *Proceedings of the 8th International Conference of Distributed Computing Systems (ICDCS 88)*, San Jose, CA, IEEE, CS Press, USA, 1988.

[20] M. T. Heath, *Scientific Computing: An Introductory Survey*, McGraw-Hill Science Engineering, July 2001.

[21] M. Maheswaran, H. Siegel, *A Dynamic Matching and Scheduling Algorithm for Heterogeneous Computing Systems*, Proceedings of the 7th Heterogeneous Computing Workshop. IEEE Computer Society Press.

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

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

[24] R. Huang, H. Casanova, A. Chien, *Using Virtual Grids to Simplify Application Scheduling*, in Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'06) , Rhodes, Greece, April 2006.

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

[26] R. M. Badia, Jesús Labarta, and Raúl Sirvent, et al, *Programming Grid Applications with GRID superscalar*, Journal of GRID Computing, December 2004.

[27] S. Bowers, B. Ludaescher, A. H.H. Ngu, T. Critchlow, *Enabling Scientific Workflow Reuse through Structured Composition of Dataflow and Control-Flow,* Proceedings of the 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] T. Fahringer, R. Prodan, R. Duan, F. Nerieri, S. Podlipnig, J. Qin, M. Siddiqui, H.-L. Truong, A. Villazon, M. Wieczorek, *ASKALON: A Grid Application Development and Computing Environment*, *6th International Workshop on Grid Computing*, (C) IEEE Computer Society Press, November 2005, Seattle, USA

[30] 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.

[31] 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

[32] X. Chu, K. Nadiminti, J. Chao, S. Venugopal, and R. Buyya, *Aneka: Next-Generation Enterprise Grid Platform for e-Science and e-Business Applications,* Proceedings of the 3rd IEEE International Conference and Grid Computing, Bangalore, India, Dec. 10-13, 2007 (to appear).

[33] Z. Yu and W. Shi, *An Adaptive Rescheduling Strategy for Grid Workflow Applications*, in Proceedings of the 21st International Parallel and Distributed Processing Symposium, Long Beach, Mar 26 -30, 2007.

[34] A. Benoit, V. Rehn, and Y. Robert, *Complexity results for throughput and latency optimization of replicated and data-parallel workflows*, Proceedings of the HeteroPar'2007: International Workshop on Heterogeneous Computing, in conjunction with with Cluster'2007, IEEE Computer Society Press, 2007.

Appendix: Macro-dataflow programming model

In macro-dataflow programming model, user can implicitly specify the dependent relationship between the data generated during the execution for application and easily edit the executing logic for each execution token in the coarse-grained dataflow. The main APIs for composing dataflow graph are as follows:

- *Execute*.**Compute**(*InStream*[] *inputs, OutStream*[] *outputs*), which is heritaged by users to add instructions to execute the token.
- *Token*.**SetExecute**(*Execute*) is used to specify the set of instructions of token
- *Token*.**AddInput**(*Stream*) is used to specify input streams for each token.
- *Token*.**AddOutput**(*Stream*) is used to specify output streams for each token.
- **SetInitialStream**(*Stream*, *file*) is used to set the input files for the whole dataflow graph.
- **SetResultStream**(*Stream*, *file*) is used to set the output files to contain the result of dataflow execution.

When the APIs are used, the user need not specify the complex dependent relationship between data generated during the execution. Our system internally composes the dataflow graph through the implicit data relationship, and provides APIs to allow users to check correctness of the graph.