CHAPTER 18

DATAFLOW COMPUTATIONS ON 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
E-mail: {chaojin, raj}@csse.unimelb.edu.au

This chapter 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 with a simple interface for programming applications with complex parallel patterns. The associated runtime system dispatches tasks onto distributed resources through a proposal online algorithm, called L-HEFT (Localized Heterogeneous Earliest-Finish-Time), and manages failures and load balancing in a transparent manner. The system has been implemented in a .NET-based enterprise Grid software platform, called Aneka. Evaluates of the scalability and fault tolerance properties of the system has been performed. 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

In recent years, parallel and distributed computing techniques have been applied to execute e-Science [31] and e-Business [26] applications over P2P [8] and Grid computing [16] platforms. The complex nature of these distributed applications has led into research of simplifying development and deployment over large scale distributed environments. Large scale distributed systems within an organization, also called Enterprise Grids or Desktop Grids, have been pioneered by systems, such as Condor [22], XtremWeb [9], SETI@Home [7], 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 [34] 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 execution and (b) supports namespace for data generated during the dataflow execution.

In the rest of this section, we introduce the dataflow computation model, and then discuss the advantage to support dataflow computation in enterprise Grid environments and related scheduling of DAG tasks in Grid environments.

1.1 *Dataflow Model*

Dataflow computation model [33, 34] is a powerful model of parallel computation, whose inherently parallel nature can be used to freely express parallelism and avoid the single point execution bottleneck common in traditional sequential computation model. In the dataflow model, computation can be represented as a directed graph, which consists of actors connected by directed arcs. An actor can be a single instruction or a sequence of instructions, while the data required by actors flows through arcs. An actor can be fired (executed) whenever all the data it requires are ready. In a dataflow execution, many actors may be ready to fire at the same time.

Fig.1 shows a simple example of dataflow graph for computing $C = (B * A) + (A/15)$. In this figure, circles represent actors and arrows represent arcs and each actor consists of one instruction. The square represents a constant value. Different from the execution in a traditional sequential model, the multiplication and division actors can be fired at the same time in a dataflow model.

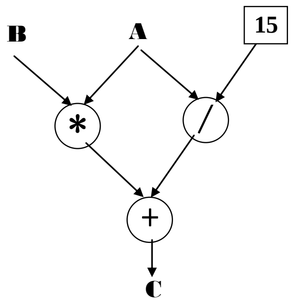

Fig. 1. A dataflow example.

When the concept of dataflow was first proposed in 1970s, it was mainly used in the domain of computer architecture for massive parallelism, as an alternative of "von Neumann" architecture. Dennis dataflow graph [18] uses a dataflow program graph to represent and exploit the parallelism in programs. In particular, operations are specified by actors, which are enabled for execution when all required data are produced by dependent actors. The dependency relationships between pairs of actors are defined with the arcs of graph, while conditional expressions and iterations are represented by decision and control actors respectively. Kahn process network [11] replaces actors in Dennis graph with sequential processes, which can communicate by sending messages through unbounded FIFO channels.

However, the parallelism used in pure dataflow computation operates at a too fine grain level, which leads to "excessive consumption of resources" in the real system and introduces unnecessary cost for sequential applications compared with von Neumann model. To avoid the overhead from fine grained actors in pure dataflow model, in 1990s, the hybrid dataflow/von Neumann model was proposed to support more coarse grained actors [5].

The hybrid dataflow model is motivated by the recognition that did not take dataflow and von Neumann techniques as two concepts which are mutually exclusive and irreconcilable. On the contrary, they can be taken as the two extremes of a continuum of possible computer architectures. From the perspective of parallelism granularity, pure dataflow model works on a fine grain level, while thread model over von Neumann architecture works on a comparatively coarse grain level. Sterling [32] explored the performance of different levels of granularity in dataflow machines. His result indicates that both fine-grained (in pure dataflow model) and coarse-grained (in sequential execution) dataflow cannot offer better parallel performance than a medium-grained approach. The hybrid model is flexible in combining the advantages of dataflow and control-flow, as well as in exposing parallelism at a desired level. Through the hybrid model, a region of actors within a dataflow graph can be grouped together as a coarse-grained thread to be executed sequentially, while the data-driven method of dataflow can be used to activate and synchronize the execution of threads. Example hybrid models include EARTH [12], P-RISC [28], and McGill dataflow architecture [10].

The recent evolution of dataflow model is large-grained dataflow. Large-grained dataflow can begin with a fine-grained dataflow graph, which can be analyzed and partitioned into sub-graphs. The sub-graphs can be compiled into sequential von Neumann processes, which can run in a multithread environment according to dataflow scheduling principles. Normally these processes are called macroactors. Recently, these macroactors can be easily programmed in an imperative language, such as C or Java rather than derived from fine-grained dataflow graph. Component techniques can also be used to implement macroactors. A large number of computer architectures and computation models are inspired by dataflow models as discussed in [20, 34].

The dataflow approach has had important influences on many areas of computer science and engineering research such as programming language, signal processing, architecture design, parallel compilation and distributed computing. Currently, dataflow as a programming model still has advantages on exposing a natural parallel interface and smoothly bridging parallel applications and underlying execution environments. Especially in an epoch when large scale parallel and distributed environments and multi-core CPUs are becoming popular, the natural parallel interface of dataflow model makes an important promise to meet the challenge of pervasive parallelism [21].

1.2 *Enterprise Grids*

Enterprise Grid computing systems are oriented towards enabling virtualization and harnessing of various types of distributed IT resources within an enterprise. They are specifically focused on provisioning resources dynamically to different projects depending on their priorities, which are often driven by business goals. They need to support various types of workloads and applications. That means, enterprise Grids need to provide a comprehensive environment for developing various types of applications based on different parallel programming models and abstractions. They also need to embrace emerging hardware and software architectural models such as multi-core processors and service-oriented architecture and build on standards based platforms such as Microsoft's .NET and J2EE.

It is important to borrow mature experiences from high-end supercomputing, but the current application programming models are difficult to use for expressing, debugging and testing parallel programs [13] and especially not suitable for novice concurrent programmers. Parallel programs need to manage threads, locks, and explicit synchronization mechanisms. This puts limitation on exploration of modern software development approaches such as those based on components. To enable programmers to build parallel applications easily, we need high-level constructs and abstractions, which are easier to understand, express, and use than those offered by threads and locks.

The complex nature of these parallel/distributed applications has led to research into simplifying development and deployment over large scale distributed environments. Grid computing platforms such as Condor [6, 18], Entropia [2], Pegasus [7], and ASKALON [30], 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 data communication between tasks, but emphasizes on the high level problem of matching 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 [27] aims to simplify the development of Grid applications with a different method, wherein users can write sequential programs 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 [29] is a scientific workflow system that allows composition of both data and control flows. It also provides a graph interface for programming.

We developed a Next Generation Desktop Grid software system, called Aneka, that serves as a flexible and extensible software farmework for realising multiple application models, security solutions, communication protocols and persistence without affecting an existing ecosystem. Aneka was conceived with the aim of providing a set of services that make grid construction and development of applications as easy as possible without sacrificing flexibility, scalability, reliability and extensibility.

This chapter presents realisation of a macro-dataflow model and its implementation using Aneka enterprise Grid software services. As a result, our macro-dataflow model works in .NET-based distributed network computing environment and support interface for composition of coarse-grained dataflow graphs, which can be converted into a DAG task graphs and scheduled for execution on distributed computing systems. Addition details on macro-dataflow programming model can be found in Section 2.

1.3 *Dynamic Scheduling of DAG tasks*

To efficiently execute the macro-dataflow computation in distributed environments, we need an efficient mapping algorithm that assigns tasks in the DAG graph of the dataflow to distributed resources. Furthermore, it should be robust enough to handle heterogeneity and frequent failures common in the target execution environment (shared enterprise Grids) containing autonomous resources/contributors.

Meeting these requirements is a challenge [1] 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) [14] and genetic search [3], map DAG tasks to distributed resources prior to execution. Such static methods do not work effectively for dynamic distributed environments where the availability of resources and their capability varies dynamically at runtime.

To effectively schedule tasks of a 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 [19] and Virtual Grid [25] deploy a greedy matching algorithm to decide the mapping of each task during execution. It is difficult to achieve overall optimized mapping and efficiently utilization of global resources for this category of policies.

Another method is to start with a static scheduling plan, and then perform iterative rescheduling for adaptation to resource changes [36]. Although these policies can potentially achieve optimized overall efficiency, 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 [25]. The plan switching method [15] can construct a family of activity graphs beforehand and investigate the means of switching from one member to another 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. Furthermore, most heterogeneous scheduling algorithms do not pay attention to efficient failure handling.

We propose a *Localized Heterogeneous Earliest-Finish-Time* (L-HEFT) scheduling algorithm for our macro-dataflow system within a dynamic heterogeneous environment in which failures are common. 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 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 a 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 rest of the chapter is organized as follows. First, it presents a simple and powerful macro-dataflow programming model, which supports the composition of parallel applications for transparent deployment in a heterogeneous distributed environment, and an architecture and runtime machinery of a dataflow system. Then, it discusses an L-HEFT heuristic online scheduling algorithm and evaluates the scalability of our system and the performance of the scheduling policy with real applications in an enterprise Grid environment. Performance results demonstrate that our system is effective supporting data parallelism for real applications and our scheduling policy achieves the same performance target as existing dynamic policy with scheduling cost 30% to 50% less than the existing static mapping-based rescheduling policies.

2. Macro-Dataflow Programming Model

This section describes the macro-dataflow programming model which is used to compose the coarse-grained dataflow graph for the whole execution. With this model, users can implicitly specify the dependent relationship between the data generated during execution of the application and easily edit the execution logic for each actor in the coarse-grained dataflow.

The macro-dataflow graph is represented as 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 users 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, users can edit execution tasks and configure its input and output streams. We call each execution task as an *actor*.

2.1 *APIs*

The main APIs for composing dataflow graph are as follows:

- *Execute*.**Compute**(*InStream*[] *inputs, OutStream*[] *outputs*), which is inherited by users to add instructions to execute the actor.

- *Actor*.**SetExecute**(*Execute*) is used to specify the set of instructions of actor.
- *Actor*.**AddInput**(*Stream*) is used to specify input streams for each actor.
- *Actor*.**AddOutput**(*Stream*) is used to specify output streams for each actor.
- **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.

Through the above APIs, it is clear that the user does not need to specify the complex dependent relationship between data generated during execution. However, our system internally composes the dataflow graph through the implicit data relationship, and also provides other APIs to make the user check correctness of the graph.

2.2 *Namespace for Streams*

We expose a namespace to specify streams within macro-dataflow graph. Each stream 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 category of streams; *Version* denotes the index for the stream along the time axis during the computing process; *Space* denotes the index along the space axis during execution. In the following text, we call the name of stream as *name*. In particular, *Category* is a string; the type of *Version* is integer and the type of *Space* is integer array.

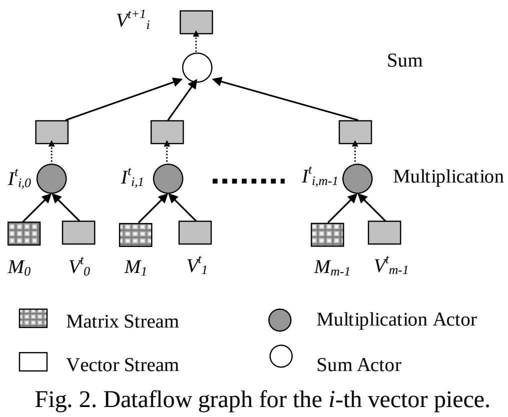

Fig. 2. Dataflow graph for the *i*-th vector piece.

We use an example to illustrate how to use the namespace to specify streams. It 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 stream. To name them, *Category* = *M* denotes the matrix vertices and *Category* = *V* denotes the vector vertices. For ***i***-th vector vertex, the dependency relationship between streams should be specified as:

<*V*, *t*, *i*>←{<*M*, *0*, *i*>, <*V*, *t*-1, *j*>} (j=1…*m*).

2.3 *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 sub- macro-dataflow graph is illustrated in Fig. 2.

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

Given *m* partitions and *T* iterations, Fig.3 illustrates how to compose the data flow graph for this example.

```
produce ComposeGraph(T, m, mulExec, sumExec)
/* T ←   iteration times
  m ←   partition number
  mulExec ←   instructions for multiplication actor
  mulExec ←   instructions for sum actor
*/
    for (t = 0; t < T; t++) {
      for (i = 0; i < m; i++) {
        matriStream = Name ("M", 0, i);
            vecStream = Name ("V", 0, i);
        for (j = 0; j < m; j++) {
          /*multiplication  result*/
          interStream = Name ("I", t, i, j);
          mul = CreateActor("Multiplication");
          mul.SetExecute(multiExec);
          mul.AddInput(matriStream);
          mul.AddInput(vecStream);
          mul.AddOutput(interStream);
        }
        sumStream= Name("V", t, i); //sum result
        sum = CreateActor("Sum");
             sum.SetExecute(sumExec);
        for (j = 0; j < m; j++) {
          interV = Name("I", t-1, i, j);
          mul.AddInput ( interStream);
        }
             mul.AddOutput(sumStream)
      }
    endfor
return
```

Fig. 3. Composition of the dataflow graph.

Finally users set the input files for the matrix and vector pieces through SetInitialStream(). Also users need to specify to collect the result streams through SetResultStream().

3. Architecture and Design

This section describes the dataflow system which supports the execution of our 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.

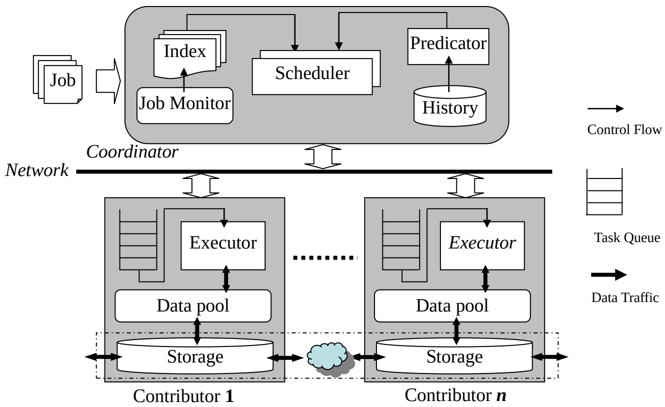

Fig. 4. Architecture of Dataflow System.

3.1 *System Overview*

The key components of dataflow computing system are: *coordinator* and *contributors*, as illustrated in Fig. 4. The coordinator is responsible for accepting jobs from users, organizing contributors to work cooperatively. For example, it monitors the availability of resources, sends execution requests to contributors, and handles 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 [35], which is a .NET-based enterprise Grid software platform and can support the creation of enterprise Grid environment. We utilize the existing Grid services such as Resource Monitor Service supported in Aneka to simplify our implementation.

3.2 *Structure of Coordinator*

Coordinator consists of a set of key subcomponents, 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 execution. 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 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 and maintains a task queue to buffer the commands from the 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*: fetches 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. In order to reduce the cost of replicating intermediate data for tasks in every level (see section 0), 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 when 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 the available 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 makes tasks within the 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 [17]; 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 historical performance information.

4.1 *DAG Execution Model*

We take the macro-dataflow graph as our execution model. With our macro-dataflow programming model, the dataflow graph is converted into a directed acyclic graph (*DAG*). Please refer to section 0 for the details on dataflow graph. For each actor in the graph, if all of its input streams are available, it is ready to execute.

4.2 *Performance prediction*

In dataflow execution, different tasks may share the same execution instructions. To predict execution time of *actor.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 the corresponding input stream. α is a value selected between 0 and 1. A larger value of α gives higher weights to recent executions and Equation 1 also takes the weight of input size into consideration as illustrated.

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

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 which 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.

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

If it is the first time to execute $v_i.Exec$ in all of the available contributors, we use the prediction value provided by the 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 prior works [24, 30]. 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 has a level number less than k and at least one vertex is in level k-1.

The result of clustering is to partition $\boldsymbol{G}$ into L blocks numbered consecutively from 0 to L-1, and execution actors 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. Fig. 5 shows the result of clustering for a FFT dataflow graph. This partition phase can be done during the execution of the dataflow graph.

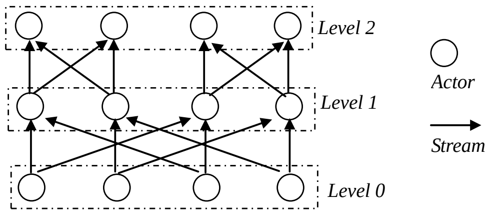

Fig. 5. The dataflow graph of FFT with 4 points.

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 the 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 [14] is a static scheduling algorithm which can potentially achieve an overall optimized mapping with a relatively 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 the whole graph and the 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 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 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.

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 the 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 map them to suitable contributors. 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 Fig. 6. 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 denoted as e_{ij}. $OS(v_j)$ is the set of contributors which holds e_{ij}.

Fig. 7 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 according to non-increasing predicted execution time. This phase of assigning rank and sorting tasks is totally different from HEFT algorithm, and we do not require the phase

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

of recursive 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 requested input streams, and then assign other tasks to the contributor which 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 0, each contributor has a priority tasks queue which contains all assigned tasks. On the side of the 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 and $left\_time(f_r)$ is the sum of execution time for tasks in *fr*. Therefore, $EFT(t, r) = EST(t, r) + Predict(t, r)$, where $Predict(t, r)$ is our algorithm to predict

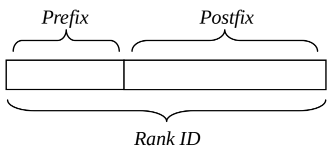

Fig. 6. Rank ID

the execution time of task *t* on contributor *r*, which is based on the history execution information as indicated in Section 0.

During the initial phase of our scheduling model, we deploy a greedy policy on initial tasks in block 0. During the scheduling, to prevent the worst case where some nodes hold a schedule queue with an estimated time much longer than others' 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.

```
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_i in R
       Compute its traffic size s_i, if t_0 is assigned to r_i
     Endfor
       Append t_0 to f_m , where s_m=max{s_i} (f_m ∈ F)
       else
     foreach r_i in R
       eft[r_i] =EFT (t_0, r_i, C, f_i, R)
     endfor
     Append t_0 to f_m , where eft[r_m]= min{eft[r_i]}
   endif
 endwhile
return F
```

Fig. 7. Level-based HEFT algorithm.

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 some contributors leave the system due to failures or being dominated by interactive users, 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 actors in order to regenerate the lost intermediate streams. The tasks to be re-executed 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 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 Jacobi iteration [23].

5.1 *Environment Configuration*

The experiments are executed in an 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 runs Windows XP.

5.2 *Sample Applications*

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

- **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.

- **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. Fig. 5 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 actors to execute.
- **Jacobi Iteration:** Jacobi 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 Jacobi 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 the single coordinator, which is a sequential behavior.

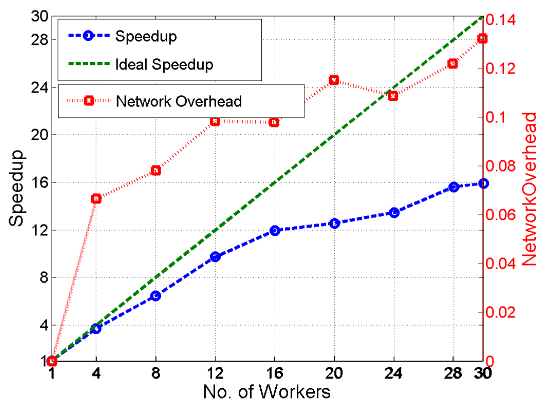

Fig. 8. Scalability of Performance.

Fig. 8 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 Fig. 8, 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 [36] through rescheduling on static HEFT mapping, which we term here as D-HEFT. We have implemented D-HEFT as mentioned in prior work 35, and rescheduling is triggered when the performance of resource is changing. In our implementation, the rescheduling is overlapped with the execution 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 while varying the ratio of computation vs. communication and the size of dataflow graph through Jacobi iteration and FFT benchmarks. For

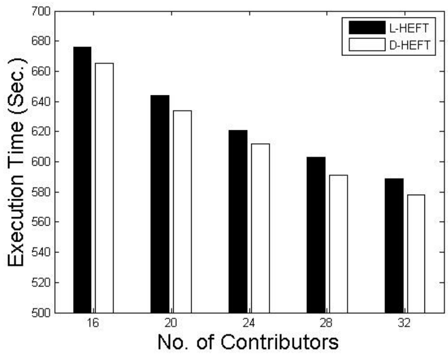

Fig. 9. Scheduling on Jacobi DAG with 640 tasks.

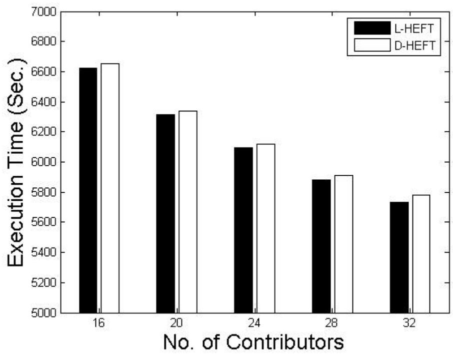

Fig. 10. Scheduling on Jacobi DAG with 6400 tasks.

Jacobi iteration, every actor executes same set of instructions. We choose α in equation 1 to be equal to 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 sizes. We use a Jacobi iteration benchmark with 10 iterations and 100 iterations. Therefore the corresponding dataflow graph respectively holds 640 and 6400 tasks. As illustrated in Fig .9, L-HEFT scheduling policy provides worse results than D-HEFT policy, while in Fig. 10, 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.

Finally we run the FFT benchmark, whose communication pattern is more complex than that of Jacobi. The result is shown in Fig. 11. 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 Jacobi dataflow, because the task number in FFT dataflow is not large enough, which is only 1664.

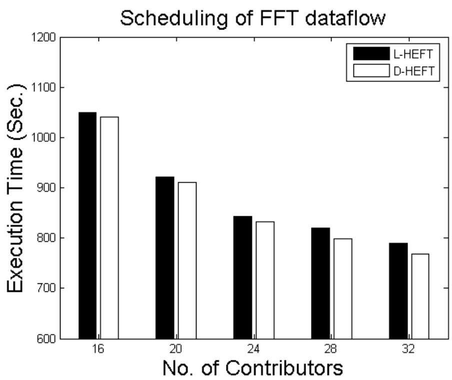

Fig. 11. Scheduling of FFT dataflow.

6. Thoughts for Practitioners

In this section, we mainly discuss the reason why L-HEFT performs better than D-HEFT and then present the effect of L-HEFT on handling the dynamics of resources in the enterprise Grid environment.

6.1 Scheduling Cost

Fig. 12 and Fig. 13 illustrate the total execution time of L-HEFT and D-HEFT during the scheduling for Jacobi examples with small and large number of tasks respectively. We can see the scheduling cost of D-HEFT is much better than L-HEFT.

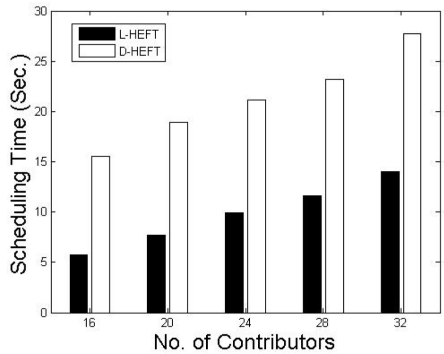

Fig. 12. Scheduling cost of Jacobi DAG with 640 tasks.

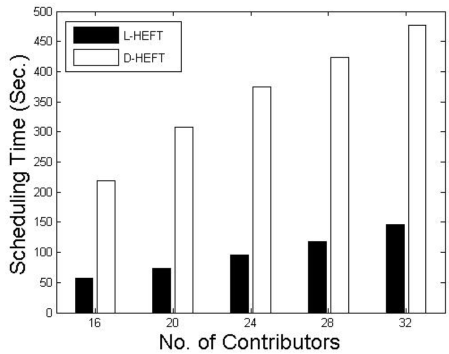

Fig. 13. Scheduling cost of Jacobi DAG with 6400 tasks.

We use a simple model to explain why the rescheduling cost of L-HEFT is better. According to [14], 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 is half the whole graph on average for each time of rescheduling. 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 comparatively 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.

Table 1 The number of rescheduling operations in D-HEFT

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

Next, we compare two scheduling polices with varied computation-to-communication ratio for a Jacobi dataflow of 10 iterations. During the experiment, we adjust the ratio of computation to communication from 2 to 12 as illustrated in Fig. 14. 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.

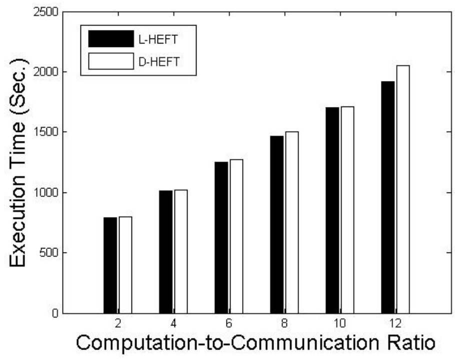

Fig. 14. Scheduling with varied ratio of Computation-to-Communication.

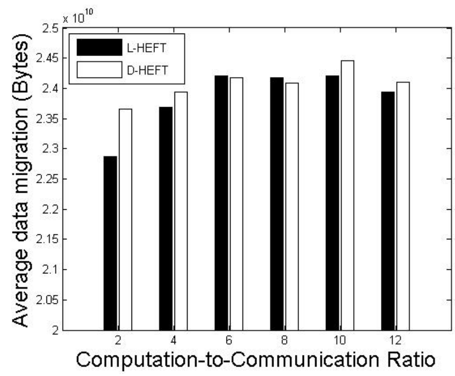

Fig. 15. Average amount of data migration between contributors.

Furthermore, we compared the amount of stream data migration between contributors under the scheduling of L-HEFT and D-HEFT, as in Fig. 15. The result shows that for the Jacobi iteration application, both L-HEFT and D-HEFT generate nearly the same amount of data migration.

6.2 *Handling Joining Contributors*

This section compares the two dynamic models for handling new resources. We use a Jacobi 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 number of finished tasks 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 actors, as illustrated in Fig. 16 and Fig. 17. These two figures show that the response time of L-HEFT algorithm to handle new joining resources is a bit faster than D-HEFT. The reason is the low cost of L-HEFT algorithm.

Fig. 16. Handling new resources for FFT dataflow.

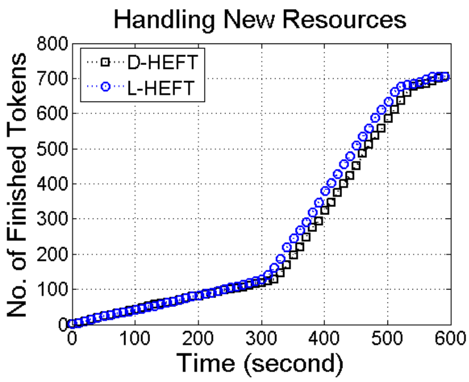

Fig. 17. Handling new resources for Jacobi dataflow.

6.3 *Handling Failures of Contributors*

This section evaluates the mechanisms dealing with failures of contributors in the dataflow system. We use the Jacobi iteration example with 40 iterations across 20 contributors. On the coordinators side, we measure the number of finished actors. If lost intermediate data need be re-generated by re-executing those actors due to the failure of contributors, we just reset those actors as unfinished. L-HEFT algorithm evaluated in this section is assisted by the replication support with the size of replication step equal to 5, while D-HEFT does not support failures through replication methods. After the system runs 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 regenerate lost intermediate data to continue the execution. However, the number of tasks which have to be regenerated by L-HEFT is pretty smaller. As Fig. 18 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.

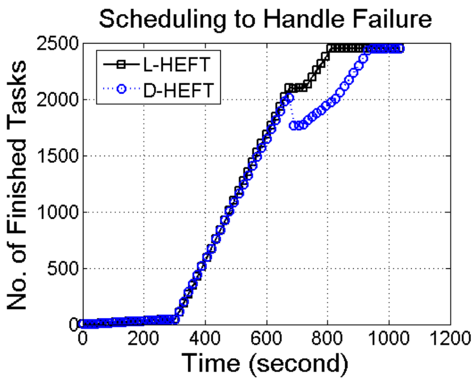

Fig. 18. Fault tolerant scheduling.

7. Directions for Future Research

Our macro-dataflow programming model exposes a DAG interface for users to compose a dataflow graph and the runtime system can schedule the execution of the graph in an enterprise Grid environment and handle load balancing and fault tolerance with a transparent manner. However, it is still a challenge to meet the quality of service (QoS) requirements. Our future work focuses on extending dynamic scheduling policy to support advanced user QoS requirements by building on Aneka’s advanced resource reservation and service level agreement (SLA) capabilities.

8. Conclusion

This chapter 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.

9. Acknowledgement

This work is partially supported by research grants from the Australian Research Council (ARC) and the Australian Department of Innovation, Industry, Science and Research (DIISR). We thank Srikumar Venugopal, Suraj Pandey, Charity Laplap and James Broberg for their comments on this chapter.

10. References

1. 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 Cluster 2007 Conference, Austin, TX, USA, 2007.
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), pp.597-610, Academic Press, USA, 2003.
3. 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), pp.795-812, IEEE Computer Society Press, USA, 1999.
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, Singapore , 2007.
5. C. Kim, J.L. Gaudiot, *Data-flow and Multithreaded Architectures*, Encyclopedia of Electrical and Electronics Engineering, Wiley Press, USA, 1997.
6. D. Thain, T.Tannenbaum, and M. Livny. *Distributed computing in practice: The Condor experience.* Concurrency and Computation: Practice and Experience, 17(2-4), Wiley Press, USA, 2005.
7. E. Deelmana,G. Singha, and M.Sua etal, *Pegasus: A framework for mapping complex scientific workflows onto distributed systems*, Scientific Programming Journal, 13(3), pp. 219–237, IOS Press, The Netherlands, 2005.

8. E. Korpela, et al, *SETI@home-massively distributed computing for SETI.* Computing in Science & Engineering, 3(1), pp.78-83, Joint Press of IEEE Computer Society and the American Institute of Physics, USA, 2001.
9. G. Fedak, C. Germain, et al. *Xtremweb: A generic global computing system*. Proceedings of the 1st International Symposium on Cluster Computing and the Grid, Brisbane, Australia, 2001.
10. G.R. Gao, *An efficient hybrid dataflow architecture model*, Journal Parallel and Distributed Computing, 19(4), pp.293-307, Elsevier Press, The Netherlands, 1993.
11. G. Kahn, *The semantics of a simple language for parallel programming*, Proceedings of the International Federation for Information Processing (IFIP) Congress 74, Stockholm, Sweden, 1974.
12. H.H.J. Hum, O. Maquelin, K.B. Theobald, X. Tian, G.R. Gao, and L.J. Hendren, *A study of the earth-manna multithreaded system*, Journal of Parallel Programming, 24 (4), pp.319-347, Plenum Press, USA, 1996.
13. H. Sutter, J. Larus, *Software and the Concurrency Revolution*, ACM Queue, 3(7), pp. 54–62, ACM, USA, 2005.
14. 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), pp.260-274, IEEE Computer Society Press, USA, 2002.
15. 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, Rhodes Island, Greece, 2006.
16. I. Foster and C. Kesselman, *The Grid Blueprint for a Future Computing Infrastructure*, Morgan Kaumann Publishers, USA, 1999.
17. J. Blythe, S. Jain, and E. Deelman, et al, *Task Scheduling Strategies for Workflow-based Applications in Grids*, Proceeding of 4th IEEE International Symposium on Cluster Computing and Grid, Singapore, 2005.
18. J. Dennis and D. Misunas, *A Preliminary Architecture for a Basic Data-Flow Processor*, The Second IEEE Symposium on Computer Architecture, 1974.
19. 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, Redondo Beach, California, USA, 2001.
20. J.L. Gaudiot, L. Bic, *Advanced Topics in Data-Flow Computing*, Prentice-Hall, USA, 1991.
21. K. Asanovic, R. Bodik and B. C. Catanzaro, *The Landscape of Parallel Computing Research: A View from Berkeley*, Technical Report, No. UCB/EECS-2006-183, University of Berkeley, USA, 2006.
22. M. Litzkow, M. Livny, M. Mutka, *Condor - A Hunter of Idle Workstations*, Proceedings of the 8th International Conference of Distributed Computing Systems (ICDCS 88), IEEE Computer Society Press, USA, 1988.
23. M. T. Heath, *Scientific Computing: An Introductory Survey*, McGraw-Hill Science Engineering, USA, 2001.
24. M. Maheswaran, H. Siegel, *A Dynamic Matching and Scheduling Algorithm for Heterogeneous Computing Systems*, Proceedings of the 7th Heterogeneous Computing Workshop, Florida, USA, 1998.

25. R. Huang, H. Casanova, A. Chien, *Using Virtual Grids to Simplify Application Scheduling*, in Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS'06) , Rhodes, Greece, 2006.
26. R. Kalakota and M. Robison, *E-business: roadmap for success*, Addison-Wesley Longman Publishing Co., Inc. USA, 1999.
27. R. M. Badia, Jesús Labarta, and Raúl Sirvent, et al, *Programming Grid Applications with GRID superscalar*, Journal of Grid Computing, Springer Verlag, Germany, 2004.
28. R.S. Nikhil, Arvind, *Can dataflow subsume von Neumann computing?* Proceedings of the International Symposium on Computer Architecture (ISCA-16), Jerusalem, Israel, 1989.
29. 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, Georgia, USA, 2006.
30. 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*, Proceedings of the 6th International Workshop on Grid Computing, Seattle, USA, 2005.
31. 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 Press, The Netherlands, 2002.
32. T.Sterling, J.Kuehn, M. Thistle, and T. Anastasis, *Studies on Optimal Task Granularity and Random Mapping*, Advanced Topics in Dataflow Computing and Multithreading. pp.349–365, IEEE Computer Society Press, USA, 1995.
33. W.A. Najjar, E.A. Lee, and G.R. Gao, *Advances In The Dataflow Computational Model*, Parallel Computing, 25, pp. 1907-1929, Elsevier Press, The Netherlands, 1999.
34. W. M. Johnston, J. R. P. Hanna, and R. J. Millar. *Advances in Dataflow Programming Languages.* ACM Computing Surveys, 36(1), pp.1–34, ACM Press, USA, 2004
35. X. Chu, K. Nadiminti, C. Jin, 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, 2007.
36. 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, CA, USA, 2007.

Extra Material

11. Terminologies

[1] **Dataflow model**: Dataflow execution model is an inherently parallel model and the execution of instruction/operation is triggered as soon as the required input data is (made) available.

[2] **Control flow model**: traditional execution model of "von Neumann" architecture.

[3] **Hybrid dataflow model**: The hybrid model combines the advantages of dataflow and control-flow and expose parallelism at a desired level.

[4] **Grid**: Grid computing is one of the recent paradigms for parallel and distributed computing. It provides online access to computation or storage as a service supported by a pool of distributed computing resources.

[5] **Enterprise Grid**: Enterprise Grid aims to enable the virtualization and harnessing of (idle) IT resources within an enterprise

[6] **DAG**: Directed Acyclic Graph (DAG) is used to express the dependency relationships between tasks within an application.

[7] **DAG scheduling**: DAG scheduling can map the tasks of DAG onto distributed resources.

[8] **Off-line DAG scheduling**: Off-line DAG scheduling maps the tasks in a DAG onto distributed resource prior to the execution.

[9] **On-line DAG scheduling**: On-line DAG scheduling maps the tasks in a DAG onto distributed resource during the execution.

[10] **Heterogeneous system:** A system consists of resources with heterogeneity in terms of architecture, capability, policies, and access interfaces, etc.

12 Questions

1. What is a dataflow execution model?

Answer: Dataflow execution model is an inherently parallel model, which can be used to freely express parallelism with a dataflow graph.

2. What is the difference between the dataflow and control flow execution models?

Answer: Normally in control flow model, programs are executed sequentially, while with the dataflow model, programs can be expressed as a dataflow graph in a natural parallel manner, which can be used to avoid the bottleneck in the control flow model.

3. What is a hybrid data model?

Answer: The hybrid model is flexible in combining the advantages of dataflow and control-flow, as well as in exposing parallelism at a desired level. Through the hybrid model, a region of actors within a dataflow graph can be grouped together as a coarse-grained thread to be executed sequentially, while the data-driven method of dataflow can be used to activate and synchronize the execution of threads.

4. What is Grid computing?

Answer: Grid computing is one of the recent paradigms for parallel and distributed computing. It provides online access to computation or storage as a service supported by a pool of distributed computing resources.

5. What is enterprise Grid?

Answer: Enterprise Grid computing systems aim to enable virtualization and harnessing of various types of distributed IT resources within an enterprise.

6. What is the advantage of enterprise Grid?

Answer: Enterprise Grids are specifically focused on provisioning resources dynamically to different projects depending on their priorities with idle resources in an enterprise.

7. What is the general challenge of DAG scheduling algorithm?
Answer: A scheduling is a process that maps and manages the execution of inter-dependent tasks in a DAG onto distributed resources, which is known as a NP-complete problem.

8. What is the advantage of HEFT scheduling algorithm?
Answer: The HEFT (Heterogeneous Earliest-First-Time) algorithm is effective on scheduling tasks in a DAG onto heterogeneous distributed resources.

9. What is the advantage of L-HEFT algorithm?
Answer: The L-HEFT algorithm is effective on rescheduling of DAG tasks in case of failures or new resources.

10. Why the scheduling cost of L-HEFT algorithm is better than D-HEFT algorithm?
Answer: L-HEFT reschedules the DAG tasks just for partial graph, while D-HEFT algorithm reschedules the whole graph. This makes rescheduling cost of D-HEFT is higher than L-HEFT.