Supporting Dynamic Migration in Tightly Coupled Grid Applications

Liang Chen  Qian Zhu  Gagan Agrawal
Department of Computer Science and Engineering
Ohio State University, Columbus OH 43210
{chenlia,zhuq,agrawal}@cse.ohio-state.edu

ABSTRACT
In recent years, there has been a growing trend towards supporting more tightly coupled applications on the grid, including scientific workflows, applications that use pipelined or data-flow like processing, and distributed streaming applications. As availability of resources can vary over time in a grid environment, dynamic reallocation of resources is very important for these applications, particularly because of their long-running nature, and because they often require large-volume data transfers between processing stages.

This paper considers the problem of supporting and efficiently implementing dynamic resource allocation for tightly-coupled and pipelined applications in a grid environment. We provide an alternative to basic checkpointing, using the notion of Light-weight Summary Structure (LSS), to enable efficient migration. The idea behind LSS is that at certain points during the execution of a processing stage, the state of the program can be summarized by a small amount of memory. This allows us to perform low-cost process migration, as long as such memory can be identified by an application developer, and migration is performed only at these points.

Our implementation and evaluation of LSS based process migration has been in the context of the GATES (Grid-based Adaptive Execution on Streams) middleware that we have been developing. We also present an algorithm for dynamic resource allocation, and have shown an architecture for resource monitoring and allocation. We have extensively evaluated our implementation using three stream data processing applications, and show that the use of LSS allows efficient process migration.

1. INTRODUCTION
One important characteristic of a grid environment is that availability of resources can vary very significantly over time. Execution and scheduling of applications in a grid environment must take such resource variability into account.

Most of the earlier work on grid computing focused on bag of tasks or the master-worker class of applications [11, 1, 10]. In scheduling the tasks associated with these applications, it is relatively simple to consider resource variability. However, in recent years, there has been a growing trend towards supporting more tightly coupled applications. Examples of such applications classes include scientific workflows [2, 21], applications that use pipelined or data-flow like processing [6], and streaming applications [17, 40, 41].

Dynamic reallocation of resources is even more important for these applications, because of two major reasons. The first is the long-running nature of these applications. The second is that these applications often require large volumes of data transfer between processing stages, and besides variability in availability of CPU cycles and memory, changes in network bandwidth can impact their execution very significantly. At the same time, implementing dynamical resource allocation is harder for these applications. This is because significant amount of state can be associated with each processing stage, and allocation of resource for each stage cannot be done independently.

This paper considers the problem of supporting and efficiently implementing dynamic resource allocation for tightly-coupled and pipelined applications in a grid environment. We provide an alternative to basic checkpointing [36]. We use the notion of Light-weight Summary Structure (LSS) to enable efficient migration. The idea behind LSS is that at certain points during the execution of a processing stage, the state of the program can be summarized by a small amount of memory. This allows us to perform low-cost process migration, as long as such memory can be identified by an application developer, and migration is performed only at these points.

Our implementation and evaluation of LSS based process migration has been in the context of the GATES (Grid-based Adaptive Execution on Streams) middleware that we have been developing [17, 16]. GATES system has been designed to support processing of distributed data streams in a wide-area environment. In the stream model of processing, data arrives continuously and needs to be processed in real-time, i.e., the processing rate must match the arrival rate.

The overall contributions of this paper are as follows.

- We have proposed the notion of LSS, and shown how it can enable efficient process migration.
- We have demonstrated an implementation of process migration using LSS in the context of the GATES middleware.
- We have presented an algorithm for dynamic resource allocation in a middleware that supports pipelined processing on streaming data, and have also shown an architecture for resource monitoring and allocation.

We have extensively evaluated our implementation using three stream data processing applications. The main observations from...
our experiments are as follows. First, the use of LSS reduces the size of process state by a factor of 30-120, and enables efficient process migration. Second, the use of LSS and migration interface introduces a very small overhead for GATES applications. Third, we show that dynamic process migration can significantly improve the performance of long-running applications. Finally, we also show that our process migration implementation does not impact the accuracy of the processing.

The rest of this paper is organized as follows. In Section 2, we summarize the main design features of the GATES system. The notion of LSS and implementation of LSS-based process migration in GATES is presented in Section 3. Our algorithm and architecture for monitoring and resource allocation is discussed in Section 4. In Section 5, we describe the applications we have implemented, and how LSS is used for each of these applications. Our experimental evaluation is presented in Section 6. We compare our work with related research efforts in Section 7 and conclude in Section 8.

2. OVERVIEW OF THE GATES SYSTEM

This section describes the motivation and the major design aspects of the GATES system.

2.1 Motivation

Increasingly, a number of applications across computer sciences and other science and engineering disciplines rely on, or can potentially benefit from, analysis and monitoring of data streams. In the stream model of processing, data arrives continuously and needs to be processed in real-time, i.e., the processing rate must match the arrival rate. There are two trends contributing to the emergence of this model. First, scientific simulations and increasing numbers of high precision data collection instruments (e.g. sensors attached to satellites and medical imaging modalities) are generating data continuously, and at a high rate. The second is the rapid improvements in the technologies for Wide Area Networking (WAN), as evidenced, for example, by the National Lambda Rail (NLR) effort and the interconnectivity between the TeraGrid and Extensible Terascale Facility (ETF) sites. As a result, often the data can be transmitted faster than it can be stored or accessed from disks within a cluster.

The important characteristics that apply across a number of stream-based applications are: 1) the data arrives continuously, 24 hours a day and 7 days a week, 2) the volume of data is enormous, typically tens or hundreds of gigabytes a day, and the desired analysis could also require large computations, 3) often, this data arrives at a distributed set of locations, and all data cannot be communicated to a single site, 4) it is often not feasible to store all data for processing at a later time, thereby, requiring analysis in real-time.

We briefly describe two representative examples. The first application we consider is online network intrusion detection, which is a critical step for cyber-security. Online analysis of streams of connection request logs and identifying unusual patterns is considered useful for network intrusion detection [22]. To be really effective, it is desirable that this analysis be performed in a distributed fashion, and connection request logs at a number of sites be analyzed. Large volumes of data and the need for real-time response make such analysis challenging. The second example is computer vision based surveillance. Multiple cameras shooting images from different perspectives can capture more information about a scene or a set of scenes. This can enable tracking of people and monitoring of critical infrastructure [8]. A recent report indicated that real-time analysis of the capture of more than three digital cameras is not possible on current desktops, as the typical analysis requires large computations. Distributed and grid-based processing can enable such analysis, especially when the cameras are physically distributed and/or high bandwidth networking is available.

We view the problem of flexible and adaptive processing of distributed data streams as a grid computing problem. We believe that a distributed and networked collection of computing resources can be used for analysis or processing of these data streams. Computing resources close to the source of a data stream can be used for initial processing of the data stream, thereby reducing the volume of data that needs to be communicated. Other computing resources can be used for more expensive and/or centralized processing of data from all sources. Because of the real-time requirements, there is a need for adapting the processing in such a distributed environment, and achieving the best accuracy of the results within the real-time constraint.

2.2 Key Goals

There are three main goals behind the design of the system.

1. Enable the application to achieve the best accuracy, while maintaining the real-time constraint. For this, the middleware allows the application developers to expose one or more adaptation parameters at each stage. An adaptation parameter is a tunable parameter whose value can be modified to increase the processing rate, and in most cases, reduce the accuracy of the processing. Examples of such adaptation parameters are, rate of sampling, i.e., what fraction of data-items are actually processed, and size of summary structure at an intermediate stage, which means how much information is retained after a processing stage. The middleware automatically adjusts the values of these parameters to meet the real-time constraint on processing. This is achieved through a self-adaptation algorithm [17, 16].

2. Support distributed processing of one or more data streams, by facilitating applications that comprise a set of stages. For analyzing more than one data stream, at least two stages are required. Each stage accepts data from one or more input streams and outputs zero or more streams. The first stage is applied near sources of individual streams, and the second stage is used for computing the final results. However, based upon the number and types of streams and the available resources, more than two stages could also be required. All intermediate stages take one or more intermediate streams as input and produce one or more output streams. GATES’s APIs are designed to facilitate specification of such stages.

3. Enable easy deployment of the application. This is done by supporting a Lauchner and a Deploer. The system is responsible for initiating the different stages of the computation at different resources. The system also allows the use of existing grid infrastructure. Particularly, the current implementation is built on top of the Open Grid Services Infrastructure (OGSI) [24], and uses its reference implementation, Globus Toolkit (GT) 3.0.

GATES is also designed to execute applications on heterogeneous resources. The only requirements for executing an application are: 1) support for a Java Virtual Machine (JVM), as the applications are written in Java, 2) availability of GT 3.0, and 3) a web server that supports the user application repository. Thus, the applications are independent of processors and operating systems on which they are executed. Further details of how our middleware uses GT 3.0 are documented in an earlier publication [17].
3. IMPLEMENTING DYNAMIC MIGRATION USING LSS

This section describes our LSS based approach for supporting dynamic migration in streaming applications.

3.1 Motivation

Availability of resources in a grid environment usually varies dramatically. Therefore, dynamically allocating new grid resources and migrating applications from resources with high utilization to new resources could improve performance significantly. In particular, long running applications, such as applications that process streaming data over a long period of time, critically need dynamic migration. This is not only because bandwidth and/or CPU availability for the resources they are using can change, but new resources can also become available over time.

To support dynamic migration, checkpointing is usually used. Specifically, a snapshot of system’s runtime state, including processes’ execution points, memory stacks (pages), and CPU status, are taken and a checkpoint is created. This checkpoint is then transmitted to a new node, where the original execution environment is restored, and processes are restarted at the points when the checkpoint was taken.

Applying such a methodology in a grid environment poses several challenges. First, checkpoints are usually platform-dependent, i.e., they include information such as CPU status and memory image, which are dependent on the hardware and operating system. A grid comprises heterogeneous resources, and grid standards have been designed to support applications that are independent of hardware and operating system. Thus, using basic checkpointing to migrate applications is not practical in a grid environment.

Furthermore, large-volume checkpoints can result in inefficient migration, especially for data stream applications. The size of a checkpoint could be considerable because it includes image of memory segments used by the application. Data stream applications process very large amounts of data, and this processing typically is done in memory. Thus, the size of the required memory is quite large, which results in large-sized checkpoints. Taking and transferring these large-volume checkpoints and restoring the original execution environment in a wide-area setting can incur significant overheads. The impact of this is even more severe on applications that process data streams, as they need to meet real-time constraint on processing of data that is arriving continuously.

3.2 Light-Weight Summary Structure

We now describe our approach for supporting migration, which is based on the notion of a Light-Weight Summary Structure (LSS).

The design of LSS is based on the observation that for many application classes, including data stream processing and other pipeline/data-flow like systems [6], the processing structure is as follows:

... 
while(true) 
{
    read_data_from_streams();
    process_data();
    accumulate_intermediate_results();
} 
...

During each loop, a number of data items from the stream are read and processed. The processing results are accumulated to form a summary information. We name the data structure storing such summary information as the Light-Weight Summary Structure (LSS). Other data structures used by the application are considered

1. //Initialize auxiliary structures 
   initializeAuxiliaryStructures();
2. //Get an LSS instance from GATES 
   counting-lss lss = GATES.getLSS("counting-lss");
3. //Process streaming data 
   while(true) 
   {
       readDataFromStreams();
       processData();
       accumulateIntermediateResults_toLSS(lss);
       initializeAuxiliaryStructures();
       //check if migration is needed 
       if(GATES.ifMigrationNeeded())
       {
           GATES.migrate(lss);
           break;
       }
   } 

Figure 1: Pseudo-code for An Application using GATES Migration API

Auxiliary Structures. Memory locations in auxiliary structures are always reset to initial values at the end of each loop.

Two additional observations are important with respect to LSS. First, for most applications, the size of LSS is much smaller than that of auxiliary structures, and therefore, also much smaller than the total memory used by the application. Second, since auxiliary structures are anyways reset at the end of each loop iteration, they do not need to be migrated, if the migration is occurring at the end of a loop iteration.

Based on these observations, LSS can be used for supporting dynamic migration. The middleware can provide functions to specify a block of memory to be LSS. When a migration operation is needed, only LSS is automatically transmitted to a new node. The middleware will also be responsible for restoring the LSS at the new node. As noted earlier, migration can only be supported at the end of each loop iteration. After migration, applications can resume from the start, instead of the specific execution point at which LSS is migrated.

The migration supported in this fashion is distinct from migration using normal checkpointing in the following ways:

- As noted earlier, migration procedure becomes more efficient. Only LSS, a much smaller portion of the overall memory needed by the applications, is migrated.
- Despite achieving efficiency, we do not negatively impact the accuracy of processing.
- An application developer’s effort in making the application capable of migration is also quite small. They only need to identify which variables belong to LSS, when implementing a processing stage. Other steps are taken care of by the middleware, as we will explain later.
- LSS is a logical data-structure and its contents are quite independent of the specific platform. We do not need program contexts to migrate, and therefore, it is easier to support migration across heterogenous platforms.

3.3 Middleware Implementation
We now present details of how LSS is used in GATES for supporting migration.

The pseudo-code in Figure 1 shows how an application utilizes migration API functions. Before calling any migration API functions, an application needs to implement an LSS class that declares the summary structure as its member. In the application shown here, the LSS class is `counting-lss`.

We can use `getLSS(Name-of-LSS-Class)` to specify a LSS class to GATES. GATES will return an instance of the LSS class, which is either created locally, or cloned from the LSS instance at a remote node if the application migrates from that node. At the end of each loop, `if(MigrationNeeded())` is invoked to inquire GATES whether the condition to migrate is met. If migration is needed, the `migrate(Instance-of-LSS-Class)` function is called, in which the LSS instance is cloned to a new node. After the execution of the `migrate()` function, the application at the current node ends. GATES implementation of the migration procedure is explained below.

**Migration Procedure:** Figure 2 shows the overall procedure. We assume that A, B and C are nodes where three pipelined stages of an application built on GATES are executing. The second stage, originally at B, is being migrated to a new node, B'. The procedure is triggered when B invokes the function `migrate()`. The migration procedure comprises 8 steps, as we will describe now. These steps occur within GATES and are hidden from the application.

First, a new path from A to B' to C is created. This step involves launching a GATES grid service at B', establishing socket connections between A, B, B' and C, and initializing internal buffers for these socket connections. Second, we stop sending data from output buffers of A and B. At the third step, C is notified to move data which is in the `obsolete` input buffer to the new one. Even though no data is sent out as the result of the second step, there might be some data items still residing in system’s buffers or being transmitted in networks. Therefore, the third step does not end unless all data in system buffers and networks passed onto the obsolete buffer. The fourth step involves copying data from the output buffer at B to the one at B', using a socket connection established between B and B' earlier. In the fifth step, the LSS instance is serialized at B and sent in the form of a byte stream to B', where a cloned LSS instance is deserialized from the byte stream. Thus, whenever `getLSS()` is invoked by the application at B', the cloned LSS instance is returned. The step 6 is similar to the step 4: we wait until all data in socket buffers and networks empty into the input buffer at B, and then copy them to the input buffer at B'. In the step 7, we load the corresponding application code into B' and initiate its execution. Finally, as the application at B' is processing data, A is notified to move data within the obsolete output buffer to the new one. In the scenarios where B is connected to multiple upstream stages, the above procedure can still be applied.

### 4. **DYNAMIC RESOURCE ALLOCATION**

In the last section, we had described how dynamic migration is implemented in GATES. We now focus on when and where a processing stage is migrated. We first introduce the overall architecture for resource monitoring and dynamic allocation that we have implemented, and then describe our dynamic resource allocation scheme.

#### 4.1 Architecture for Resource Monitoring and Allocation

In our current implementation, we make use of information services [20] provided by Globus toolkit 3.0 (GT3.0) to collect resource information in clusters. More specifically, at every computing node that has installed GT3.0, an information service is always executing. This service periodically collects resource information from the node, such as the CPU load and the size of available memory. For every cluster, we also setup an Index Service to aggregate such resource information collected by the information services. Thus, to discover desired CPU and memory resources, we just need to query these index services. For network resources, we could utilize the Network Weather Service [51] to measure end-to-end TCP performance. We assume that the computing resources available to us are large clusters, which have adequate CPU and memory resources. Further, we are more concerned about the performance of network links connecting different clusters, since high-speed networks are generally available within a cluster. Therefore, we view inter-cluster bandwidth as the most critical resource. If CPU and memory resources become a bottleneck, the computing stages can be moved to a different set of nodes within the same cluster. On the other hand, if inter-cluster bandwidth become a bottleneck, a new series of clusters and corresponding inter-cluster network links have to be allocated.

Figure 3 shows the overall architecture. The important components are the Grid Resource Manager, the Deployer and GATES Monitoring Services. The Grid Resource Manager is responsible for: 1) collecting network topology and bandwidth information, 2) constructing a weighted graph and keeping it updated, 3) searching for an appropriate node (or a set of nodes) in a cluster by querying its index service, and 4) applying the static and the dynamic resource allocation algorithms to generate deployment configurations. A deployment configuration contains information about which nodes are allocated to an application and how they should connect to each other. In our earlier work [15], we had described how a static resource allocation algorithm generates a deployment configuration for our middleware.

According to deployment configurations, the Deployer instantiates GATES grid services at specific nodes and launches the application as described in [17]. When a GATES grid service is instantiated, a corresponding monitor, which is an instance of a GATES monitoring service, is created. A monitor has three main functions: 1) determining whether its corresponding grid service is the bottleneck in the pipeline, 2) if corresponding grid service is the bottleneck, requesting the Grid Resource Manager to find a new location for it, and 3) instructing its grid service to migrate to the new locations.

Here, we make two significant observations on the design we have presented. First, instead of integrating the monitoring function into the GATES grid service, we have implemented them separately. We believe that this significantly reduces the performance overheads incurred by monitoring activities, since the grid services can now focus on application execution only. Second, `push` (notification) approach is applied for the communication between the grid services and the monitoring services. Instead, if the `pull` (query) approach was used, the monitoring services’ continuous queries about the grid services’ buffer status could have resulted in much higher overheads.

#### 4.2 Dynamic Resource Allocation Scheme

There are two major components to our dynamic resource allocation scheme. The first is an approach for determining the bottleneck in a pipeline. The second is the our method for determining alternative path(s).

Applications are executed on our middleware in a pipelined fashion. Intuitively, to achieve the best performance from a pipeline, the pipeline needs to be balanced and no single stage should be allowed to become a significant bottleneck. Therefore, accurately determining the existing bottleneck is a prerequisite to any effec-
Step 2: Stop Sending Data

Step 1: Create a New Path

Step 8: Move Data

Step 6: Copy Data

Step 5: Clone LSS

Step 4: Copy Data

Figure 2: Migration Procedure

Figure 3: Architecture of Resource Allocation Components
tive dynamic resource allocation scheme. Our overall strategy is to model the system as a queuing network and decide the bottleneck by observing each queue’s load. Specifically, to get a queuing network model, we can model every stage as a server and view the input buffer of a stage as the queue of the server. Similarly, we can also model the network connecting two consecutive stages as a server and view the output buffer of the first stage as the queue of this server.

Such a queuing network comprises of several paths or pipelines. Each path in the queuing network could have a bottleneck. We just focus on a single path. We believe that a server is the bottleneck in a path if 1) the server’s queue is the currently overloaded, and 2) it is the last overloaded queue in the path. The reason for the second condition is that we prefer to identify at most one bottleneck, and given multiple overloaded servers, we will like to relocate the last overloaded one with the highest priority. If an input buffer of a stage instance is the last overloaded buffer, we conclude that the bottleneck is the stage instance and the CPU resources are scarce. In such a case, we need to find a new node with adequate CPU cycles, which we assume can be done within the same cluster. Instead, if the output buffer of a stage instance is the last overloaded buffer, then the bottleneck is the network connection. Therefore, a network with higher bandwidth is desired.

We determine a new path as follows. Assume that a node $N$ is outputting data to the bottleneck network link. We use the static algorithm to calculate a path from $N$ to the destination. This avoids the need for migrating any of the stages prior to the bottleneck. However, it is possible that no better alternative path may be available from $N$. Therefore, we check if the path computed by the static algorithm is identical to the one currently being used. If this is the case, we recompute a new path from the predecessor of $N$. This process can be repeated until a new path is determined or a data source is used to compute a new path.

In our framework, the above scheme is implemented as follows. Every monitor is responsible for monitoring the output buffer and input buffers of the corresponding grid service. When a buffer is overloaded in a grid service, the service sends an overload notification to its monitor. In turn, this monitor contacts the monitors for the later stages in the path to check whether the buffer being overloaded is the last overloaded buffer. If the monitor finds out the stage it is monitoring is the bottleneck, it will inform the grid resource manager to allocate a new node with more available CPU cycles in the same cluster. If the bottleneck is a network connection, the monitor will ask the grid resource manager to search a new path. The grid resource manager then applies the static algorithm to recompute a new path as described above. If a different path is found, a new deployment configuration will be created and the Deployer will be called to deploy the new GATES grid services. Afterwards, the new deployment configuration is sent to all impacted monitors. They then instruct the corresponding grid services to migrate to the locations specified by the new deployment configuration. How grid services migrate were explained in the previous section.

5. STREAMING APPLICATIONS

This section describes three applications that we have supported so far using GATES migration services.

**count-samps** is a distributed version of the counting samples problem. The classical counting samples problem is as follows [26]. A data stream comprises a set of integers. We are interested in determining the $n$ most frequently occurring values and their number of occurrences at any given point in the stream. Since it is not possible to store all values, a summary structure must be maintained to determine the frequently occurring values. Gibbons
and Matias have developed an approximate method for answering such queries with limited memory.

The problem we consider is of determining frequently occurring values from distributed streams. Our solution is to store $m$ frequently occurring values from each stream at a node close to the data source, and then merge them to determine the overall $n$ most frequently occurring values at a central location. The value of $m$ can be chosen to provide a tradeoff between the accuracy of the final results and the efficiency of processing. These $m$ values form the LSS at each remote node.

The second application is clustering evolving data streams [3], and is referred to as CluStream. Clustering involves grouping similar objects or data points from a given set into clusters. The particular problem considered here is clustering data arriving in continuous streams, especially as the distribution of data can change over time.

The algorithm we consider [3] approaches the problem as follows. The clustering process is divided into two major steps. The first steps involves computing micro-clusters that summarize statistical information in a data stream. In fact, the set of micro-clusters is considered the summary information of this application. The second step uses micro-clusters to compute the final clusters.

This two-step clustering algorithm can be easily implemented using the GATES middleware. Figure 4 shows the three stages that are used. The first stage is simply the data source, which sends streaming data to the second stage. The second stage computes micro-clusters. After a certain number of data points have been processed, it sends the computed micro-clusters to the third stage. The third and the final stage then applies the modified $k$-means algorithm [3] to create and output the final clusters.

To make the second stage capable of migration, we store micro-clusters as an LSS object and implement this stage as indicated in Figure 1.

The third application we have studied finds frequent occurring itemsets across a set of data streams. If the communication bandwidth is limited, this problem can be quite challenging.

The algorithm we consider is an extension of a proposed algorithm for finding frequent itemsets from distributed streams [34]. The algorithm addresses the problem stated above by arranging the nodes in a hierarchical structure. Figure 5 shows an example of such a structure. Each monitor node $M_i$ receives itemsets from the data source $S_i$ and inserts them to a list. After a certain number of itemsets are received, $M_i$ counts the frequencies of the itemsets appearing in the stream, sends this information to an intermediate node, purges all itemsets in the list and starts a new round. Intermediate nodes combine the frequency information received from their children and pass them up to their parent node. To reduce the communication load, the monitor and intermediate nodes should avoid sending less frequent itemsets over the links. Therefore, the algorithm uses an error tolerance parameter $\epsilon$ at every node, except the data sources. Only the itemsets with frequencies greater than this parameter are forwarded to the next node. Finally, the root node outputs the itemsets whose frequencies exceed the specified support threshold $\tau$.

To make monitors migratable, we have implemented a LSS class to store the unprocessed itemsets. This is based on the following observation: a monitor’s migration could happen when the monitor still accumulates itemsets. These accumulated itemsets, which have not yet been processed, need to be transmitted to the remote node. Therefore, we choose these itemsets as the summary information.

<table>
<thead>
<tr>
<th>Number of Micro-Clusters</th>
<th>Size of LSS (KB)</th>
<th>Size of Clustream (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1,007</td>
<td>1,143</td>
</tr>
<tr>
<td>20</td>
<td>1,143</td>
<td>1,437</td>
</tr>
<tr>
<td>40</td>
<td>25</td>
<td>1,437</td>
</tr>
<tr>
<td>80</td>
<td>50</td>
<td>1,673</td>
</tr>
<tr>
<td>100</td>
<td>62</td>
<td>2,382</td>
</tr>
</tbody>
</table>

6. EXPERIMENTAL EVALUATION

This section presents results from a number of experiments we conducted to evaluate the notion of Light-Weight Summary Structure and our implementation of dynamic resource allocation and process migration. Specifically, we had the following goals in our experiments:

- Demonstrate that LSS uses a small amount of memory for our target applications, compared with the total memory usage of the applications and the middleware.
- Show that dynamic migration with LSS is efficient and does not impact overall system performance.
- Show that applications benefit significantly from the dynamic migration when execution environments change dynamically.
- Show that accuracy of processing is not impacted by the migration using LSS.

6.1 Experiment Setup and Data Sets

For distributed processing of streaming data in a grid environment, we need high bandwidth networks. However, for our study, we did not have access to a wide-area network that gave high bandwidth and allowed repeatable experiments. Therefore, all our experiments were conducted within a single Linux cluster. The cluster consists of 24 computing nodes. Each node has a Pentium III 933MHz CPU with 512MB of main memory and 300GB local disk space, interconnected with switched 100 Mb/s Ethernet.

The experiments were conducted using the three streaming data mining applications described earlier in this paper. For the count-samps application, integer streams were generated by a simulator. For the CluStream application, we used the KDD-CUP’99 Network Intrusion Detection dataset. For Dist-Freq-Counting, we used a dataset generated by the IBM synthetic data generator [4]. We conducted 4 sets of experiments which we describe in the rest of this section. The first three were conducted by using both Clustream and Dist-Freq-Counting, and count-samps was used for the last experiment.

6.2 Memory Usage with LSS

This experiment demonstrates that compared to the entire application and the middleware, LSS uses a small fraction of memory.

GATES and its applications are implemented in Java. Unfortunately, Java does not provide any mechanism similar to C/C++’s sizeof() function to measure object sizes, making it difficult to get the exact size of an object in Java. We instead used two techniques to approximately estimate memory usage. We applied the technique described in [46] to measure the size of a GATES service and the application. To measure the size of an LSS object, we serialized the object into a file and then measured the file size.
For Clustream, LSS is the set of micro-clusters, the size of which depends on the number of micro-clusters. Therefore, we varied the number of micro-clusters and then measured memory usage of LSS and the entire application. The results are shown in Figure 6. When the number of micro-clusters were set to 10, 20, 40, 80 and 100, the LSS only occupied approximately 0.7%, 1.2%, 2.9% and 2.6%, respectively, of memory used by the entire application.

We further examined the memory usage for Clustream, when the number of micro-clusters was 100. The memory use by the middleware, the application, and LSS are 4,350KB, 2,382KB, and 62KB, respectively. Thus, LSS just used approximately 0.9% of the total memory consumed by GATES and the application. This clearly points to the efficiency of using LSS for checkpointing and migration.

We repeated the above experiment using Dist-Freq-Counting. Its LSS is the set of unprocessed transactions, and its size is proportional to the number of such transactions. This, in turn, depends on when migration occurs. Therefore, we migrated the application at six random time instances, and measured the LSS’s memory usage and the corresponding number of unprocessed transactions. The results are indicated in Figure 7. We also measured the average size of Dist-Freq-Counting, which is 16,422KB. Thus, the LSS only used on average 1.1% of the total memory consumed by the middleware and the application.

### 6.3 Efficient Migration

We conducted two groups of experiments to show that migration using LSS is efficient.

First, we measured the time Clustream takes to migrate, given different dataset sizes. We compared the migration time with the application execution time. Irrespective of the dataset size, migration occurs only once. The results are shown in Figure 8. As we would expect, migration time is not impacted by the dataset size. The migration time only accounts for 3% of the execution time when Clustream is executed with a dataset of size 3,200 KB. Therefore, for long running streaming applications, the time spent on migration is very small.

Second, we compared the performance of three different executions of Clustream. The first execution involves a version of the application that does not invoke GATES’s migration interfaces, and is referred to as the Without LSS version. The second and third execution involve a version that takes advantage of GATES’s migration support and has the ability to migrate. In the second execution, no migration actually occurs, while in the third execution, one stage migrates once. The second and the third executions are referred to as non-migration and migration executions, respectively.

We varied dataset sizes to 200KB, 400KB, 800KB, 1,600KB and 3,200KB and compared the execution time of these three executions. The results are indicated in Figure 10. Due to the overheads...
of invoking GATES’s migration interface, the version with LSS is slightly slower than the version without LSS. Similarly, the execution with migration takes additional time as compared to an execution that does not migrate. However, the difference between the two versions and two executions is quite small, which shows that overheads of supporting a migration interface and migrating during execution are quite small. Moreover, as shown in Figure 11, the time differences between executions with and without migration are always smaller than the corresponding migration time. This is because to reduce migration overheads, we carry out the step 7 of the migration procedure (Figure 2) with processing data in a new node.

Using Dist-Freq-Counting, we repeated the above experiments and obtained very similar results. They are presented in Figure 9, Figure 12, and Figure 13, respectively.

### 6.4 Benefits of Migration in a Dynamic Environment

In this subsection, we show that an application can benefit significantly from taking advantage of the GATES’s dynamic migration support in a dynamic environment, where CPU cycles and network bandwidths can vary. For our experiments, we only considered variation in network bandwidth.

We first considered Clustream. We varied the bandwidth of the network between a data source and an intermediate stage of
The migration version moved the intermediate stage as early as possible to a new node which continued to have a high-speed connection to the data source. The non-migration version, in comparison, stayed at the original nodes. We varied the network bandwidths from 1Mbps, 100Kbps, 10Kbps, to 1Kbps, then observed the execution time of these two versions. As shown in Figure 14, though two versions’ execution time is close in situations where the bandwidths are 1Mbps and 100Kbps, the migration version is 4 and 33 times faster, respectively, than the non-migration version when the bandwidths are 10Kbps and 1Kbps. Figure 15 considered Dist-Freq-Counting and the results are very similar.

6.5 Processing Accuracy and LSS Migration

We now investigate how migration using LSS impacts accuracy of data stream processing. The count-samps application was used to conduct this experiment. Since it is an approximate algorithm, two executions may not produce the same results. The methodology we followed for our evaluation was as follows.

We synthetically generated an integer stream in which 50% numbers are 1, 25% numbers are 2, 12.5% numbers are 3 and so forth. Thus, the top 10 frequently occurring numbers in the stream are 1, 2, 3, . . . 10. To compare accuracies of various counting results, we designed two criteria to quantify an output’s accuracy.

The first criterion considers how many numbers are correctly chosen. The ideal result is that numbers 1, 2, 3, . . .,10 are picked, regardless of their number of occurrences. Then the accuracy of the ideal result is 10, according to the first criterion. Similarly, the accuracy of the worst result, where no number in [1, 10] is picked, is 0. We conducted 5 rounds of experiments for each migration version and non-migration version. We calculated average accuracy of 5 results’ for each version, and it turned out the average accuracy for both versions were identical, i.e. 8.8. Note that even the sequential algorithm for counting samples is approximate, and will not be completely accurate.

The first criterion does not consider how close a number’s occurrence frequency in a counting result is to its true occurrence frequency in the stream. To overcome the first criterion shortcoming, we designed the second criterion as shown below. Let $R_1$ be the set of 10 most frequently occurring values determined by the algorithm. Then, we denote $R_1$ as

$$R_1 = R \cup \{1, 2, \ldots, 10\}$$

Let $t_i$ denote how frequently a value occurs in the stream, and let $T_i$ be the frequency reported by the count-samps application. Then, we can compute the accuracy of results, $A$, as:

$$S = \sum_{i \in R_1} t_i$$

$$S' = \sum_{i \in R} T_i$$

$$A = \sum_{i \in R_1} \frac{|t_i - T_i|}{S'} \times \frac{t_i}{S}$$

The idea of the criterion is to compare the counted frequency of each chosen number with its true frequency. The difference of two frequencies reflects how accurate a chosen numbers’ counted frequency is. The sum of these differences can reflect how accurately a set of number are chosen and counted.

We recalculate average accuracy of 5 results’ for both the migration and the non-migration versions, and they are 0.02725 and 0.02493, respectively, which shows that the results are very close. Thus, we can see that migration using LSS does not impact accuracy of processing.

7. RELATED WORK

We now compare our work with existing work checkpointing and process migration, and grid middleware and resource allocation.

Checkpointing and process migration has been widely studied in distributed systems. Here, we restrict ourself to the work done within parallel and grid computing.

Condor [48, 36] supports transparent migration of a process (through checkpointing) from one workstation to another. Our work is distinct in using LSS to make the checkpoints more efficient, and in focusing on streaming or pipelined applications. Krishnan and Gannon have focused on checkpointing for distributed components, in the context of XCAT [33]. They create a consistent global snapshot across multiple processes. We only support checkpointing and migration for a single processing stage, but make it more efficient using LSS. Vadhyanar and Dongarra have developed SRS, which is a system for developing malleable and migratable distributed applications [49].

Several researchers have addressed the problem of using an adaptive environment for executing parallel programs. In the context of Charm++, support for processor virtualization has been implemented using migratable objects [28]. Stellner developed a system called CoCheck [47], which performs process migration for MPI programs. Most of the earlier work considered a task parallel model or a master-slave model. In a version of PVM called Migratable PVM (MPVM) [12], a process or a task running on a machine can be migrated to other machines or processors. User Level Processes (ULP) [45] provides light-weight user-level tasks, which can be migrated from one machine to another. Piranha [25] was a system developed on top of Linda [7]. In this system, the application programmer has to write functions for adapting to a change in the number of available processors. Data Parallel C and its compilation system [38] have been designed for load balancing on a network of heterogeneous machines.

In the area of stream middleware, the work that is probably the closest to the GATES middleware is the dQUOB project [39, 40]. This system enables continuous processing of SQL queries on data.
streams. The GATES system is distinct in the following ways. First, we support an API to allow general processing, and not just SQL queries. Second, the processing can be done in a pipeline of stages. Third, it does not support self-adaptation. As part of the Linked Environment for Atmospheric Discovery (LEAD) project, work is ongoing to incorporate DQUOB-like support in grid environment [41]. Stampede is a cluster middleware for supporting streaming applications [42, 43]. Our work is again distinct in consider grid resources and adaptation for real-time processing. Mazucco et al. have looked at the specific support for merging multiple high speed data streams [35].

Data stream processing has also received much attention in the database community [27]. Prominent work in this area has been done at Stanford [5], Berkeley [13], Brown and MIT [9], Wisconsin [50], among others. The focus in this community has largely been on centralized processing of a single data stream. Our focus is quite different, as we consider distributed processing of distributed data streams, and use grid resources and standards. Aurora* is a framework for distributed processing of data streams, but only within a single administrative domain [18]. With increasing wide-area bandwidth, the potential for real-time wide-area distributed computing has been recognized by others as well. As part of the Optiputer project, Kim has outlined a proposal for using real-time programming techniques [32]. However, they do not consider stream-model of processing or adaptation of processing to achieve a real-time constraint.

Our work has some similarities with the grid-based (dynamic) workflow projects, including the SDSC Matrix project1, work by Abramson et al. at Monash University [2], and by Deelman et al. at ISI [21]. The GATES system is distinct in considering streaming data with real-time constraint on the processing, and this paper has considered dynamic reallocation of resources, and process migration.

Resource allocation has been an important topic in the grid community. Most of the initial work has been on static matching of the resource requirements and the available resources [23, 14, 52, 29, 44, 53, 48]. Generally, these efforts have not considered pipelined or streaming applications. Much work has been done on resource discovery[31, 37], often using mobile agents or objects to do efficient search. Our focus is on resource allocation, and we assume that one of the existing techniques has been used for resource discovery. Realtor [19] is a protocol for supporting survivability and information assurance by migrating components to safe locations under circumstances of external attack, malfunction, or lack of resources. Our work is distinct in considering resource degradation and application adaptation. Isert and Schwan have developed a system called ACDS, which includes a monitoring and steering tool for adapting stream based computations [30], including assigning alternative resources. In comparison, we consider a more restrictive class of applications, but automate the dynamic resource allocation process more.

8. CONCLUSIONS

This paper has considered the problem of supporting and efficiently implementing dynamic resource allocation for tightly-coupled and pipelined applications in a grid environment. We provide an alternative to basic checkpointing, using the notion of Light-weight Summary Structure (LSS), to enable efficient migration. The idea behind LSS is that at certain points during the execution of a processing stage, the state of the program can be summarized by a small amount of memory. This allowed us to perform low-cost process migration, as long as such memory can be identified by an application developer, and migration is performed only at these points.

Our implementation and evaluation of LSS based process migration has been in the context of the GATES (Grid-based AdapTive Execution on Streams) middleware that we have been developing. We have also presented an algorithm for dynamic resource allocation, and have also shown an architecture for resource monitoring and allocation. We have extensively evaluated our implementation using three stream data processing applications. The main observations from our experiments are as follows. First, the use of LSS reduces the size of process state by a factor of 30-120, and enables efficient process migration. Second, the use of LSS and migration interface introduces a very small overhead for GATES applications. Third, we show that dynamic process migration can significantly improve the performance of long-running applications. Finally, we also show that our process migration implementation does not impact the accuracy of the processing.

9. REFERENCES


