ACDS: Adapting Computational Data Streams for High Performance *

Carsten Isert and Karsten Schwan
Georgia Institute of Technology
College of Computing
Atlanta, Georgia 30332
{isert,schwan}@cc.gatech.edu

Abstract

Data-intensive, interactive applications are an important class of metacomputing (Grid) applications. They are characterized by large, time-varying data flows between data providers and consumers. The topic of this paper is the runtime adaptation of data streams, in response to changes in resource availability and/or in end user requirements, with the goal of continually providing to consumers data at the levels of quality they require. Our approach is one that associates computational objects with data streams. Runtime adaptation is achieved by adjusting objects' actions on streams, by splitting and merging objects, and by migrating them (and the streams on which they operate) across machines and network links. Adaptive streams also react to changes in resource availability detected by online monitoring.

1. Introduction

End users of high performance codes increasingly desire to interact with their complex applications as they run, perhaps simply to monitor their progress, or to perform tasks like program steering, or to collaborate with fellow researchers. For instance, in our own past research, we have constructed a distributed scientific laboratory with 3D data visualizations of atmospheric constituents and with parallel computations that simulate the distribution of chemistries in the earth's atmosphere. While an experiment is being performed, scientists collaborating within this laboratory may jointly inspect certain outputs, may create alternative data views on shared data or create new data streams, and may steer the simulations themselves to affect the data being generated. Similarly, for metacomputing environments, Alliance researchers are now investigating and developing the Access Grid [13] framework for accessing and using computations that are spread across heterogeneous, distributed machines.

The problem addressed by our research is the creation and management of the large-scale data streams existing in distributed high performance applications. The specific streams investigated in this paper are those emanating from data stores or from running simulations and consumed by visual displays that are employed by collaborating end users. Each such stream consists of a sequence of data events that flow from information providers to consumers, generated in response to requests from the consuming user interfaces and/or generated continuously by producers. This event-based description of a data stream provides a natural vehicle for associating computations with event generation, transport, and receipt, via event handlers located in producers, consumers, or in intermediate engines. The resulting *computational data streams* constitute the basis of our approach to online stream manipulation. Specifically, we adapt the behavior of the streams' event handlers in response to changes in end user capabilities or needs and/or in response to changes in resource availability.

This paper demonstrates how to adapt computational data streams in order to gain and maintain high performance. It also presents the ACDS framework for implementing adaptive computational data streams. ACDS supports runtime configuration actions that include (1) the migration of stream computations, (2) the specialization of these computations, and (3) the splitting and merging of stream computations to increase and decrease concurrency as per a stream's runtime needs.

2. Related Work

The ACDS system and its support for computational data streams rely on a high performance event infrastructure, called ECHo [2], layered directly on transport-level communication protocols and capable of moving data at rates exceeding those of high performance programming platforms like MPI. In comparison to its prior use in sys-

*This work was funded in part by NCSA/NSF grant C36-W63.

tem monitoring and control, ACDS employs ECHo both for transporting data and for controlling data transport. ECHo's event-based paradigm supports data streams with multiple producing or consuming subscribers; it can deal with dynamic subscriber arrivals and departures; and it supports runtime evolution for the types of data events a subscriber produces or consumes.

The Interactivity Layer Infrastructure (ILI) [11] represents our previous approach to online data stream adaptation. This paper extends that work by using *Active User Interfaces (AUIs)* [7] that emit control events so that ACDS can react to changes in end user needs as well as to changes in underlying system loads in both LAN and WAN environments. Furthermore, the ACDS framework offers adaptation capabilities and support beyond that provided by ILI, including robust split and merge operations, decision algorithms, monitoring support, etc.

ACDS-controlled computational data streams may be used with arbitrary parallel applications, including those written with meta-computing systems like Globus [4] or Legion [5]. In these contexts, ACDS addresses only the runtime control of the computational data streams linking such Grid computations to end users. In comparison, the load-balancing and resource management mechanisms included with the grid computing frameworks themselves concern the runtime management of the actual grid procedures, threads, or processes. It would be interesting to study how ACDS' data stream management interacts with load-balancing performed for grid computations. Finally, both Cumulvs and ACDS customize task migration by use of application-specified knowledge. However, as with the load-balancing performed in other metacomputing environments, Cumulvs does not know about entire data streams, nor does it support stream adaptations like component splitting or merging. ACDS' implementations of the split and merge operations could take advantage of previous approaches to component checkpointing and migration, e.g. [9]. However, the restrictions imposed by such approaches forced us to develop our own approach.

3. Sample Application

The sample application used in our research is a global atmospheric climate model [8]. The data streams of principal interest to this paper link the running model and/or data stored from previous model runs to visualizations employed by end users.

From our users' perspectives, useful views of this data display information about species concentration in grid form, where a grid point represents an area of approximately 5.6 x 5.6 degrees of latitude and longitude on the earth's surface. In order to provide this data view, however, the computational data stream producing it must first transform data from its model-resident or stored 'spectral' form to the grid-based form meaningful to end users. This transformation (termed 'Spec2Grid') may be performed on the receiving machine, on some intermediate node, or by the data producer. The resulting pipeline-structured computational data stream linking a single producer to two consumers is shown in Figure 1. This figure also shows an additional computation (termed 'Gridred') that filters the data being sent to the UI so that only those grid points currently requested by the end user are actually sent. The actual display processing is done in the elements termed AUI.

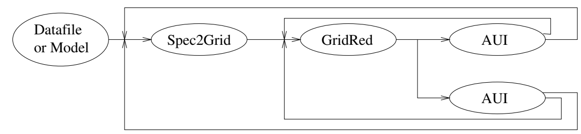

Figure 1. Sample Access Grid Application

Some of these computations are substantial, and so are their effects on the sizes of data events being transported. For instance, a typical spectral to grid transformation can be performed at the rate of 213 levels per second on a Sun Ultra 30, and the data expansion implied by this conversion increases the size of spectral data by a factor of 4.04 when producing grid data. This implies that it would be advantageous to postpone conversion until the data reaches the consumer, in order to preserve bandwidth. However, even high performance graphical rendering machines, like our OpenGL-based, feature-poor active UI running on an SGI Octane, can be overwhelmed by the processing and storage demands of a visualization that must render large data sets. This is one of the interesting problems to be addressed by the runtime methods for data stream configuration presented next.

4. ACDS: Concepts

ACDS supports the construction and adaptation of computational data streams used in scientific applications. Since stream computations may themselves be computationally intensive, they can benefit from parallelization. This motivates ACDS' 'split' and 'merge' adaptations described below. Since the amounts of data being streamed may be large, data cannot be viewed in its entirety at all times. This implies the need for data filtering and the need to change filtering at runtime in accordance with current user behavior or needs; these needs motivate ACDS' support for the runtime adaptation of parameters in single or sets of stream components. Finally, ACDS supports the runtime migration of stream components, in order to deal with dynamic variations in the node and network loads of the underlying computational and access grids.

Parameter Changes are actions that alter the behaviour of individual stream components. ACDS supports such changes with control events consumed by stream components and generated by user interfaces, by the ACDS monitoring and steering tool (MST), or by other stream components.

Task Migration. Dynamic load-balancing algorithms [14, 12] may be classified by the distribution levels of their algorithms and by the ways in which they can affect application behaviour, ranging from local knowledge and local changes to global knowledge and global changes. ACDS enables task migration based on both local or global migration methods, by supporting the movement of individual and/or of multiple stream components, and by permitting such movements to be initiated by stream components themselves or by remote sites. Our implementation of task migration assumes that stream components offer explicit operations for state saving and migration, as commonly done in restart files for scientific applications.

Task Splitting. We call the set of parallel tasks generated through splitting a *program*, while its individual components are called *tasks*. Splitting is difficult when performed for a stream component (i.e., a program) that communicates with other programs, each of which may itself consist of multiple tasks. To address these difficulties, the split operation may be used in three different modes.

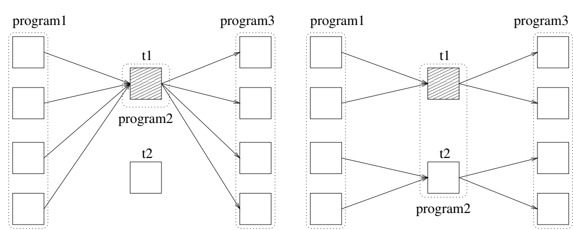

Figure 2. Start Position and Distribution Mode

Parallel Mode. Each copy of the source task performs a different job, with tasks negotiating for jobs. This mode is used to increase the level of parallelism of the stream component being split. Sample uses of this mode include executing the same code on different data (SPMD) or executing different codes on shared parts of the data (MIMD). Stream components split in this fashion must make the correct assumptions concerning the necessary synchronization at their respective inputs and outputs. Figure 2 depicts a situation in which 'Program2' is split into two tasks, with each task operating on half the data. Three alternative synchronization methods can be employed by the parallelized version of 'Program2': (1) synchronization at the inputs of 'Program2', (2) synchronization at its outputs, and (3) synchronization at both its inputs and outputs. A typical use of the split operation is one in which some particular stream component is parallelized, followed by the reassembly of results in a subsequent component. Cost models resident in the MST can provide estimates of the potential benefits of split operations, as shown with the experimental results described in Section 6.

Redundant Mode. A program 'split' in 'redundant mode' generates two parallel tasks (from one initial task) that both execute the same operations on the same data, and that send their outputs to the same target(s) as the initial task. This mode is useful when splitting is performed to improve component reliability.

Configuration Mode. A program 'split' in 'configuration mode' again results in two identical tasks. However, their outputs may be directed at different targets. This mode is useful when dynamically creating a 'branch' in a computational data stream, perhaps to process and visualize the same data as the original branch, but using different processing methods and displays. An example drawn from the sample application presented in Section 3 is one in which one stream branch extracts physical information from atmospheric data (e.g., wind velocities), whereas the second, new branch extracts chemistry information (e.g., ozone concentrations).

Merging is the inverse operation of splitting. The only difficulty with merging concerns connection reconfiguration for adjacent stream components. ACDS' buffer management and communication facilities integrated with stream components automatically deal with such reconfiguration.

Adaptation Transactions. The implementation of these operations is based on a technique called adaptation transactions. It is a distributed version of the multiprocessor mechanism first described in [1] and is based on a 2-phase transaction protocol. Additional detail appears in [6], including possible optimizations, graph analysis methods and the fashion in which cycles are removed, failure recovery, and how to deal with concurrent adaptation requests.

5. ACDS: Architecture and Implementation

Framework for Stream Components. Given a code module that implements the basic functionality of a stream component, it should be straightforward to construct a new stream component and integrate it into existing streams. In particular, component programmers should not have to be concerned about the underlying ACDS structure that monitors component behavior, supports splitting and merging, implements adaptation transactions, and handles component connections. Toward these ends, the implementation of ACDS stream components utilizes two basic C++ classes, as depicted in Figure 3: (1) the *basic stream component* class provides communication support via its *event channel interface* class and other basic utilities like monitoring. This class uses ECHo event channels for inter-component communications. (2) The *adaptable stream component* class provides everything that is necessary to carry out adaptations.

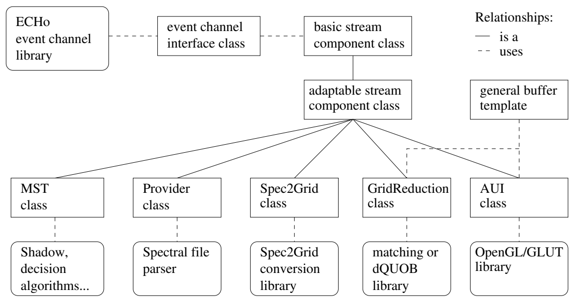

Figure 3. Class Hierarchy and Libraries

A new stream component is created by deriving a new application class from the *adaptable stream component* class. The computational code must be provided in the stubs and the internal state that should be transferred in case of component migration must be identified. Finally, information about the manner in which the new component may be split into multiple tasks (and merged) must be supplied. For the sample application described in this paper, each stream component is derived from the adaptable class, even the data provider, the AUI, and the monitoring and steering tool (MST) itself, (see Figure 3).

Internal Component Structure. The internal structure of a stream component is not visible to application programmers, but it is useful to describe it to place into context the performance results presented in Section 6.

Each stream component consists of four different threads. The monitoring thread gathers timing information about the performance of this stream component and counts events, including incoming events, computation times, and outgoing events. This information is sent over the monitoring channel to the MST. Monitoring events can also serve as acknowledgments for completed adaptations. The enactment thread carries out adaptation transactions, and it interacts with the MST and with other stream components, as necessary. Both of these threads are run periodically. Most of the actual 'work' in a stream component is performed by the network thread accepting inputs and the computation thread running the component's code and issuing output events. Since event communications are asynchronous, each stream component can take advantage of communication/computation overlap in its operation.

MST Structure. The Monitoring and Steering Tool (MST) supervises ACDS' stream operation and adaptation. Its main components are the data management system, adaptation decision algorithm, and adaptation enactment mechanism. Data management keeps track of the stream's task graph, of the node graph of available processors, and of the mapping of tasks to processors. Associated with each task is a monitoring trace window and other attributes like mapping constraints, available adaptation actions, and operating system. The MST also performs resource management, by keeping track of previously created local and remote processes. These processes act as 'containers' for newly created stream components and their tasks. Once created, such containers are 'acquired' in response to 'split' operations and 'released' when tasks are 'merged'.

At startup time, each stream component sends a registration message to the MST via the system's monitoring channel; the message contains application-specific information with which data management in the MST constructs its own 'shadow' of each stream component. Runtime component registration with the MST is coupled with the fact that the MST decides on changes like 'split' and 'merge' and guarantees the consistency of the resulting 'view' of the computational stream maintained in the MST. The MST uses its internal view of the computational stream when executing its decision algorithms to make suitable stream adaptations. A detailed description of the decision algorithms employed in the MST appears in [6]. Discussions concerning the effects of monitoring rates and detail on the performance of MST decision algorithms appear in Section 6. Currently, the main bottleneck for large data volumes is the MST as shown by experiments in [6], where we also discuss in detail our solution to this problem. Briefly, since the MST is a stream component itself, it can also be split, merged, migrated, and changed in terms of internal parameters, thereby permitting us to dynamically build a hierarchy of MSTs.

6. Evaluation

This section utilizes output data from the atmospheric simulation described in Section 3. The measurements reported here use a cluster of Sun Ultra 30 workstations (128MB RAM, 247MHz, Solaris 2.5.1). These machines are connected via switched 100MBit/s Ethernet links. Data is displayed with an OpenGL-based visualization tool running on SGI O2 machines (64MB RAM, 195MHz, Irix 6.3). The SGIs are connected to the Suns via 10 MBit/s Ethernet.

The atmospheric data used in our experiments is organized by simulation time steps and by the 3D nature of this data set. Specifically, each time step simulates 2 hours of real time; atmospheric data is comprised of 9 different species, each having 64 longitudes, 32 latitudes, and 37 level values, where each value is represented by a floating point number. This results in roughly 2.7MB of data per time step in grid format. For long term storage, this data is compressed into spectral form, with a resulting constant compression rate of 4.04, thereby reducing data size for one time step to roughly 675KB.

A 'debugging' model run simulates at least 6 weeks of real-time and generates a total of about 340MB of spectral data. A run used for interesting scientific inquiries might simulate 1-2 years of real-time and produces about 1.5 to 3GB of spectral data, which translates to about 12GB in grid format. Compared to other scientific applications, these data amounts are still small. Today's large data sets can easily reach the order of several TB and are continually growing. The atmospheric data file used in our experiments resides on the local disk of one of the Sun machines on which the provider runs.

ACDS' utility for high performance data streams has two sources: (1) its ability to react to changes in the availability of underlying computing resources and (2) its ability to react to changes in end user needs. This differentiates ACDS from traditional research in load balancing and migration.

Migration experiments. The main purpose of this experiment is to show that ACDS is able to move a system from a bad initial situation to a better one. However, this initial configuration can occur during normal operation due to dynamic changes in system load or end user requirements. The 'Initial Configuration' shown in Figure 4 does not exhibit good performance because the AUI and two stream transformation and filtering components are all mapped to a single machine, and because this machine is not 'well-connected' to the computational cluster generating data.

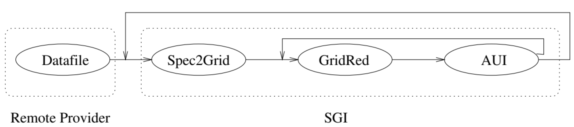

Figure 4. Initial Configuration

The MST discovers these facts by monitoring stream performance. In response, it first migrates the Spec2Grid stream component to the Sun cluster, followed by the migration of the GridRed component, the latter being the stream component that reduces network bandwidth needs by filtering the stream in response to changes in end user behavior seen by the AUI.

Configuration	Time in s
Middleware on Sun	109
Middleware on SGI	437
Migration with MST	118

Table 1. Improved Performance – Migration

Table 1 presents the results for a 'debugging' model run. The first two rows represent the best and the worst cases without the MST enabled and the stream configured by hand. The times shown are the total stream execution times for both cases. The third row depicts total stream execution time when using ACDS' stream monitoring and adaptation and the decision algorithm currently embedded in the MST.

These results are encouraging, since performance with MST is only 8.3% worse than the best possible performance attained by manual component placement. Specifically, these results demonstrate that the current delays and overheads due to MST usage are acceptable for the computational data streams addressed by the ACDS system.

Configuration	Time in s
No load	109
Load on GridReduction, no migration	337
Load on Spec2Grid, no migration	328
Load on Spec2Grid node, migration	134
Load on GridReduction node, migration	151

Table 2. External Load

The previous experiment demonstrated MST's ability to deal with heterogeneity in the underlying computing infrastructure. In comparison, the experimental results depicted in Table 2 concern performance improvements derived from ACDS and the MST in response to runtime changes in system loads. Specifically, in these experiments, we impose large additional loads on the respective computational engines. With a small delay due to monitoring and steering overheads, the stream components are migrated to idle nodes, and the stream asymptotically reaches its optimal performance.

Dynamic stream behavior. Changes in machine loads and in user requirements are two causes of stream adaptations. A third cause are runtime variations of the execution times or the communication bandwidths due to the dynamic behavior of stream components themselves. Such dynamic behaviors are common in complex components with many internal branches taken in response to the data values received as inputs.

The experiment described next simulates such component behavior, by varying the computation time of the 'Spec2Grid' component in relation to the types of atmospheric species being transformed. For experiment purposes, we assume that the most 'expensive' species requires 30 times the execution time of the 'normal' species. Runtime changes in computation times are due to users' dynamic selections of the species being viewed.

This experiment demonstrates the utility of the 'split' operation on stream components, where a user's switch from the normal to the expensive species results in a component split and therefore, in the reduction of stream component execution time due to parallelization. Figure 5 depicts the situation after the Spec2Grid element has been split once and when the expensive species is being transformed.

The MST tool currently determines suitable stream con-

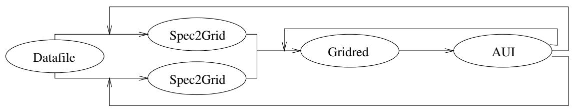

Figure 5. Situation after Splitting Spec2Grid

figurations by trial and error. The resulting stream performance for one sample data run is depicted in Table 3. In this run, the computationally expensive species is requested for 100 time steps. Four different scenarios are measured. In the first scenario, the stream is not adapted at all, so that a single instance of Spec2Grid is used throughout. In the second scenario, the stream always uses two instances of Spec2Grid. Due to workload imbalances, using three instances of Spec2Grid (shown in the third row) leads to a performance drop. Finally, performance for the same run with MST enabled appears in the fourth row. Throughout this run, the level of parallelism for Spec2Grid varies from 2 to 3, resulting in repeated split and merge operations. Methods to avoid this thrashing in general are left out for brevity.

Configuration	Time in s
Single instance of Spec2Grid	127
Two instances of Spec2Grid	84
Three instances of Spec2Grid	157
Adaptations turned on	115

Table 3. Improved Performance – Splitting

Enactment Costs. The enactment of adaptation decisions, that is, the execution of adaptation transactions, typically takes only a few seconds, thereby making it feasible to adapt computational data streams with delays suitable for end users operating user interfaces via a keyboard or a mouse. We characterize these costs in detail in [6].

7. Future Work

Future research should address the scalability of systems like ACDS to the large-scale, wide area 'access grid' computations and 'portals' now being envisioned by HPC researchers. Specific topics include the hierarchical structuring of system monitoring and steering methods and tools, additional support for system reliability, and the integration and use of multiple system and network monitoring tools, including MOSS [3] or OMIS [10] for individual stream components, and ReMoS for network monitoring.

Acknowledgments. Thanks to Greg Eisenhauer, Vernard Martin, and Beth Plale for their ideas and their support for this project. Thanks to Arndt Bode and Thomas Ludwig from the TU Munich for their support of this work at Georgia Tech.

References

[1] T. E. Bihari and K. Schwan. Dynamic adaptation of real-time software. *ACM Transaction on Computer Systems*, 9(2):143–174, 1991.

[2] G. Eisenhauer. The echo event delivery system. Technical Report GIT-CC-99-08, College of Computing, Georgia Institute of Technology, Atlanta, GA 30322-0280, 1999.

[3] G. Eisenhauer and K. Schwan. An object-based infrastructure for program monitoring and steering. In *Proceedings of the 2nd SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT'98)*, pages 10–20, August 1998.

[4] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *Intl J. Supercomputer Applications*, 11(2):115–128, 1997.

[5] A. S. Grimsaw and W. A. Wulf. Legion – a view from 50,000 feet. In *Proceedings of the Fith IEEE International Symposium on High Performance Distributed Computing*, August 1996.

[6] C. Isert. A framework and adaptive methods for improving the performance of dynamic parallel programs. Master's thesis, Technische Universität München and Georgia Institute of Technology, 1999.

[7] C. Isert, D. King, K. Schwan, B. Plale, and G. Eisenhauer. Steering data streams in distributed computational laboratories. In *High Performance Distributed Computing (HPDC8)*, 1999. full report available as GIT-CC-99-12.

[8] T. Kindler, K. Schwan, D. Silva, M. Trauner, and F. Alyea. A parallel spectral model for atmospheric transport processes. *Concurrency: Practice and Experience*, 8(9):639–666, November 1996.

[9] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and migration of unix processes in the condor distributed processing system. Technical Report 1346, University of Wisconsin, April 1997.

[10] T. Ludwig, R. Wismüller, V. Sunderam, and A. Bode. *OMIS — On-Line Monitoring Interface Specification (Version 2.0)*, volume 9 of *Research Report Series, Technische Universität München*. Shaker, Aachen, 1997.

[11] V. Martin and K. Schwan. ILI: An adaptive infrastructure for dynamic interactive distributed applications. In *Proceedings of the Fourth International Conference on Configurable Distributed Systems*, pages 224 – 231. IEEE, IEEE Computer Society, May 1998.

[12] R. P. R. Diekmann, B. Monien. Load balancing strategies for distributed memory machines. In Karsch/Monien/Satz, editor, *Multi-Scale Phenomena and their Simulation*, pages 255–266. World Scientific, 1997.

[13] R. Stevens et al. Plans for creating access grid technology for the national machine room. TeamB/C Working Group Meeting Minutes, Argonne National Laboratories, Chicago, February 1999.

[14] M. J. Zaki, W. Li, and S. Parthasarathy. Customized dynamic load balancing for a network of worksations. Technical Report 602, The University of Rochester, December 1995.