

Executing Multiple Pipelined Data Analysis Operations in the Grid *

Matthew Spencer[‡], Renato Ferreira[‡], Michael Beynon[†], Tahsin Kurc[‡],
Umit Catalyurek[‡], Alan Sussman[†], Joel Saltz[‡]

[‡] Dept. of Biomedical Informatics [†] Dept. of Computer Science
The Ohio State University University of Maryland
Columbus, OH, 43210 College Park, MD 20742
{spencer.4,ferreira.18,kurc.1,catalyurek.1,saltz.3}@osu.edu
{als,beynon}@cs.umd.edu

Abstract

Processing of data in many data analysis applications can be represented as an acyclic, coarse grain data flow, from data sources to the client. This paper is concerned with scheduling of multiple data analysis operations, each of which is represented as a pipelined chain of processing on data. We define the scheduling problem for effectively placing components onto Grid resources, and propose two scheduling algorithms. Experimental results are presented using a visualization application.

1 Introduction

Processing of data in many applications that query and manipulate scientific datasets can be represented as an acyclic, coarse grain data flow, from one or more data sources (e.g., one or more datasets distributed across storage systems) to the client. For a given client query, the data of interest is retrieved from the corresponding datasets. The data is then processed via a *pipelined* sequence of operations, while it progresses from the data sources to the client. In this work we address the scheduling of multiple queries, represented as pipelined chain of operations on data, in a Grid environment.

Our approach focuses on component-based frameworks, where an application is developed from a set of interacting software components [13, 18, 20, 28, 27]. In a component-based framework, the placement of components onto computational resources represents an important degree of flexibility in optimizing application performance. Network and computation overheads can be decreased by efficiently placing components to deal with computational heterogeneity in both the application and the available resources [7, 8]. Parallelism is another method for increasing performance, by executing multiple copies of a single component on a single host machine or across a set of hosts [10]. In this paper we define the scheduling problem for effectively placing components in a pipelined data processing chain onto Grid computational and network resources. Two scheduling algorithms are described to efficiently execute client queries in a dynamic Grid environment when multiple queries are presented to the runtime system. We present preliminary experimental results from a visualization application, implemented using a component-based framework.

*This research was supported by the National Science Foundation under Grants #EIA-0121161, #EIA-0121177, #ACI-9619020 (UC Subcontract #10152408), #ACI-0130437, and #ACI-9982087, Lawrence Livermore National Laboratory under Grant #B500288 and #517095 (UC Subcontract #10184497), and the Department of Defense, Advanced Research Projects Agency, USAF, AFMC through Science Applications International Corporation under Grant #F30602-00-C-0009 (SAIC Subcontract #4400025559).

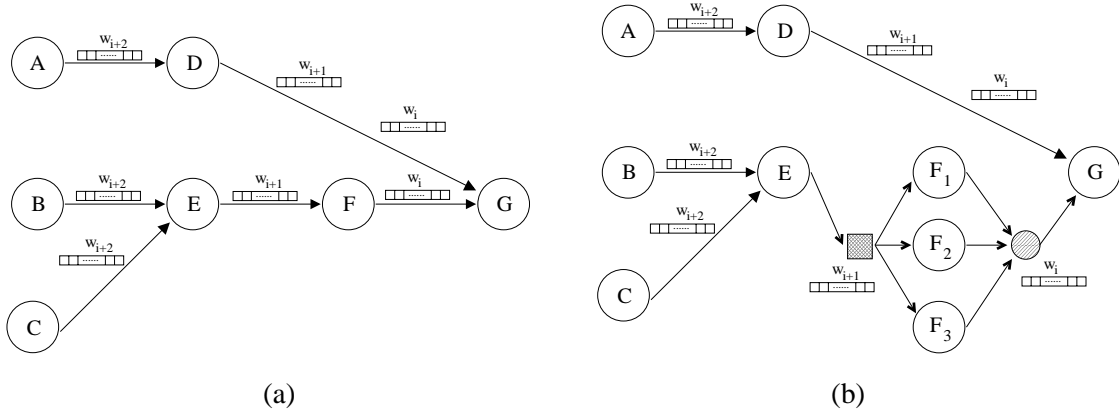


Figure 1: (a) A sample filter group with 7 filters. (b) The filter group with three transparent copies of filter F.

2 Programming Model and Runtime Environment

The programming model, called *filter-stream programming* [8], is a component-based model. Each component performs a portion of the application-specific processing, and interactions between the components are realized by flow of data and control information. The interface for a component, referred to as a *filter*, consists of three functions: (1) an initialization function, in which any required resources such as memory for data structures are allocated and initialized, (2) a processing function, in which user-defined operations are applied on data elements, and (3) a finalization function, in which the resources allocated are released. Filters are connected via *logical pipes*. A *logical pipe* denotes a uni-directional data flow from one filter (i.e., the producer) to another (i.e., the consumer). A filter is required to read data from its input pipes and write data to its output pipes only. We define a *data stream* as a labeled sequence of data transferred from one filter to another. A data stream is terminated by a special *end of stream* element.

The overall processing structure of an application is realized by a *filter group* (or *filter network*), which is a set of filters connected through logical pipes. A *unit of work* assigned to a filter group is evaluated in a sequence of *wavefronts*. Figure 1(a) displays a sample filter group with 7 filters and the flow of three wavefronts w , $w + 1$ and $w + 2$ through the filter network. Each data stream is marked with a unit of work id and wavefront number. Wavefronts impose a partial ordering on processing of data elements, because the semantics of wavefronts is that a filter has to finish processing a data stream marked with wavefront w before it can initiate the processing for a data stream associated with wavefront $w + 1$. When a filter finishes processing data from wavefront w , it sends a logical *end of wavefront* signal to downstream filters (consumers) through each of its logical output pipes.

The programming model provides several abstractions to facilitate runtime scheduling and performance optimizations. A *transparent filter copy* is a copy of a filter in a filter group. The filter copy is transparent in the sense that it shares the same *logical* input and output pipes of the original filter. A transparent copy of a filter can be made if the semantics of the filter network are not affected. That is, the output of a unit of work should be the same, regardless of the number of transparent copies. Figure 1(b) shows an illustration of the filter group in Figure 1(a) with three transparent copies of filter F. Two types of transparent copies are allowed; *replicated filter copy* and *partitioned filter copy*. If a replicated copy of a filter is created, the internal state structures of the filter are replicated. The runtime system is responsible for scheduling elements (or buffers) in a data stream among the transparent copies. Various situations arise when it is useful to partition a data structure between a collection of filter copies. This kind of filter copy is called a *transparent partitioned filter copy*.

We have developed a component framework, called DataCutter [7, 8], based on the filter-stream pro-

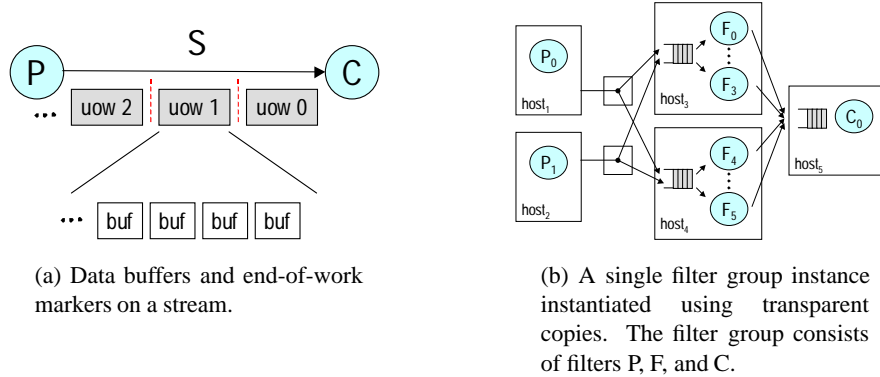


Figure 2: DataCutter stream abstraction and support for copies.

gramming model. DataCutter provides support for developing applications that execute *generalized reduction operations*, which are common in the data processing kernel of many data analysis applications [6, 14]. This type of processing consists of retrieving the data of interest and performing user-defined transformation, mapping, and aggregation operations on the data. The transformation and aggregation functions in generalized reduction operations are *associative* and *commutative*. That is, the result of the operation is independent of the order input data items are processed. An intermediate data structure, referred to as an *accumulator*, can be used to hold intermediate results during processing.

In DataCutter, filter code is expressed using a C++ language binding by sub-classing a filter base class, and the logical pipe abstraction of filter-stream programming is referred to as a *stream*, which is used for all filter communication and specifies how filters are logically connected. All transfers to and from streams are through a provided buffer abstraction (Figure 2(a)). A buffer represents a contiguous memory region containing useful data. The runtime system supports distributed and multi-threaded execution. If two or more filters are placed on the same host, each filter is executed by a separate thread. The stream communication between two co-located producer and consumer filters is carried out by passing the buffer pointers, while TCP/IP sockets are used for point-to-point stream communication between two filters on different machines. Since generalized reduction operations are order independent, a unit-of-work (UOW) is executed in a single wavefront. An example of a UOW would be rendering of a dataset from a particular viewing direction.

DataCutter provides support for *transparent replicated copies* and *filter group instances* (Figure 2(b)). Multiple filter group instances can be instantiated and executed concurrently. A UOW can be assigned to any instance. When the producer or consumer of a filter is transparently copied, the system must decide for each producer which copy to send a stream buffer to. For distribution of buffers between transparent copies, we have implemented several distribution policies: (1) Round Robin (RR), (2) Weighted Round Robin (WRR) among copies based on the number of copies on that host, and (3) a Demand Driven (DD) mechanism based on buffer consumption rate. The DD policy attempts to send buffers to the filter that will process them fastest.

A filter may maintain data structures to keep intermediate results. If copies are generated for such filters, *partial* results from the copies should be combined. This requires an *application-specific combine* filter be implemented and instantiated. The output from the combine filter goes to the next stage(s) in the data flow network of filters.

3 Scheduling of Multiple Pipelined Operations

Our goal is to create multiple filter group instances and schedule each group instance on the machines in the environment to improve the system performance, when multiple units of work (UOWs) are submitted to the system.

3.1 The Scheduling Problem

As was stated in Section 2, transparent copies and placement of filters are two optimizations that can improve the execution of a single UOW. If we consider a pipelined chain of operations on data, each filter in the chain reads a buffer from its input stream, performs some amount of processing on the buffer, and writes a buffer to its output stream. Each filter can be modeled as an entity that can produce and consume data at a certain rate. The pipeline behavior that achieves the best performance occurs if all stages in the chain are balanced with respect to each other and the communication overhead between the stages is minimized. That is, the data production rates of the producer filters is matched by the data consumption rates of the consumer filters. This requires that the right number of copies of the bottleneck filters be created and efficiently placed on the machines in the environment. Thus, a scheduler should be able to create and remove copies of filters dynamically, effectively modifying the dataflow graph. We define the scheduling of a UOW as follows.

We are given a filter network G_f , a topology graph G_t , and a set of *pinned* application filters¹. The vertices in G_f are the application filters (including the pinned filters) and the edges represent the logical pipes between producer and consumer filters. In the topology graph, the vertices are the machines in the system and the edges are the communication links between the machines. *The goal is to create as many copies of unpinned filters as necessary and map the copies onto the vertices of G_t so that all the stages in the network are balanced.*

3.2 Scheduling a UOW

In scheduling a single UOW, we use the following approach: (1) perform a controlled tuning run of the application and collect various statistics about the application behavior, (2) use the statistics in cost models for predicting behavior, and (3) apply the cost models in the scheduling algorithm.

It is important to know how much computation is performed by each filter, the volume of input data and output data for each filter, and how frequently the stream read and write operations occur. The goal of the controlled tuning run is to eliminate all outside effects such as cpu contention, memory contention, network contention, background jobs, etc., and get a clear picture of filter behavior. The controlled tuning run is performed by choosing a placement where all hosts are dedicated, homogeneous, and share a common fast network. The application appends one or more UOWs to the single filter group instance, and exits upon completion of the work. The exact statistics collected and reported during the tuning run are presented in Table 1 for a Rasterization filter in the isosurface application described in Section 4.4 [9].

3.3 Scheduling Multiple UOWs

Scheduling of multiple UOWs submitted to the application frontend is carried out as follows. The UOWs to be scheduled for execution are considered one by one. It is not our objective in this work to develop methods and tools to address the issues associated with replica creation and replica management [2, 37, 15, 18, 19]. However, we take into account the replicas when scheduling UOWs that access the replicated portions of datasets. For a UOW, we consider all the replicas of the dataset. For each replica, the filter copies and placement of copies are determined by the scheduling algorithm and an estimated execution time is output. The replica and the corresponding scheduling of filters that result in the minimum estimated execution time are chosen for executing the UOW. Once a UOW has been scheduled for execution, the application frontend calls

¹Some of the application filters will be pinned to a subset of hosts in the system by the application. The number of the copies of such filters also will be pre-determined by the application prior to carrying out the scheduling of the filter network. For example, read filters, which constitute the first stage of the pipeline in most cases, are in general executed on the hosts where the datasets are stored.

| Filter Statistics | | |
|-------------------------------------|--------------------|--|
| <i>Name</i> | <i>Sample (Ra)</i> | <i>Description</i> |
| <i>name</i> | Ra | symbolic name of filter |
| <i>t_{total}</i> | 53.2312s | total wall clock time |
| <i>t_{workq}</i> | 0.0009s | time blocked waiting for work to be appended |
| <i>t_{init}</i> | 0.0172s | time for entire init() application callback |
| <i>t_{process}</i> | 53.2118s | time for entire process() application callback |
| <i>t_{read}</i> | 34.3982s | within process(), time blocked in stream read() calls |
| <i>t_{write}</i> | 0.4498s | within process(), time blocked in stream write() calls |
| <i>t_{readfirst}</i> | 0.2700s | within process(), time for first read() call |
| <i>t_{compute}</i> | 18.0936s | within process(), time performing computation |
| <i>t_{writefirst}</i> | 0.2757s | within process(), time of first write() call |
| <i>t_{finalize}</i> | 0.0005s | time for entire finalize() application callback |
| Statistics per Input Stream | | |
| <i>Name</i> | <i>Sample (Ra)</i> | <i>Description</i> |
| <i>insj.name</i> | E-Ra | symbolic name of input stream |
| <i>insj.num</i> | 2477 | number of buffers read from the stream |
| <i>insj.vol</i> | 66411KB | total size of all buffers read from the stream |
| <i>insj.read</i> | 34.3982s | time blocked in read() calls from the stream |
| Statistics per Output Stream | | |
| <i>Name</i> | <i>Sample (Ra)</i> | <i>Description</i> |
| <i>outs_k.name</i> | Ra-M | symbolic name of output stream |
| <i>outs_k.num</i> | 2471 | number of buffers written to the stream |
| <i>outs_k.vol</i> | 25514KB | total size of all buffers written to the stream |
| <i>outs_k.write</i> | 0.4498s | time blocked in write() calls to the stream |

Table 1: Statistics collected during the controlled tuning run of an application. The Sample Value (Ra) column is the actual values recorded for a tuning run of the rasterization filter in the isosurface rendering application.

the DataCutter runtime system functions to instantiate the filter group (including the copies of the filters) and submit the client request to the filter group.

Before the next UOW is scheduled for execution, the state of the system (i.e., resource availability) is updated. The scheduler and the runtime system could keep track of information about previous schedules (i.e., the placement of filters, the amount of resources used by each filter group that has been scheduled) and update the availability of resources according to this information. An alternative approach would be to use a resource monitor, which provides information about resource usage. In this paper, we use a Grid infrastructure service, called the Network Weather Service [39]. NWS monitors the nodes and networks in the system and provides on-the-fly information about the availability of CPU on a machine and the network bandwidth and latency between any two machines in the system.

4 Filter Copy Pipeline (FCP) Algorithm

In this section, we present an algorithm for scheduling a UOW. This algorithm performs a single sweep over the filter network and aims to balance the data production rate of producer filters and the data consumption rate of consumers at each stage of a pipeline. It uses a detailed compute cycle mode, but assumes point-to-point network connections between the nodes in the environment. If two or more filters are placed on the same single processor node, they share the CPU of the node. For multiple CPUs on a single SMP node, we attempt to schedule a filter copy on each CPU in turn. The CPU that can achieve the earliest finish time is

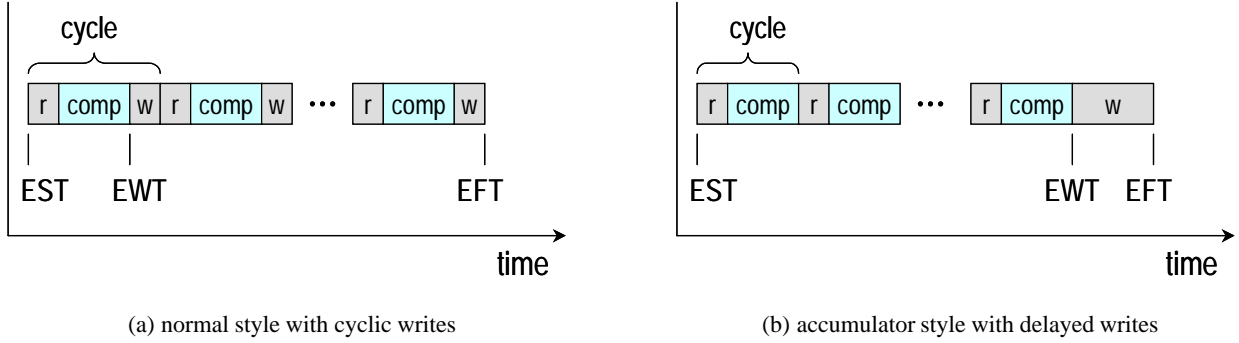


Figure 3: Diagram: Filter model with repeated filter cycles of r(ead)-comp(ute)-w(rite). EST is the earliest start time of the filter, EWT is the earliest write time, and EFT is the earliest finish time.

marked as the host of the filter copy. The idea is to approximate what a modern cpu scheduler would do on a real machine, which is to complete every job as soon as possible.

4.1 Filter Model

In processing a UOW, a filter reads some data from its input stream(s), computes for some time to generate some sort of output product, and then writes to its output stream(s). This is referred to as a *filter cycle*. Figure 3(a) illustrates this behavior. If a filter generates data by reading from disk, then the compute phase of the filter cycle would include the disk I/O calls. Some filters do not operate in this strict manner, and the read and write operations are not symmetric. For example, consider a filter that reads buffers containing data objects and renders the objects into an internal image. This can repeat until there are no more buffers to read, at which time the filter sends the final output to the next filter. This style of operation is illustrated in Figure 3(b). The filter model represents this behavior using Earliest Write Time (EWT), which denotes the write delay before write operations become part of the filter cycle. Given the amount of data read or written per filter cycle, and the frequency of filter cycles, we can compute the input and output rates of a filter.

4.2 Scheduling Algorithm

The scheduling algorithm, referred to as *filter copy pipeline* (FCP), is similar in some respects to the classic static scheduling technique called list scheduling [23], where the approach is to choose a mapping for each application filter in some order based on a ranking of the filters. For a chain of processing on data, the filters in the chain are ranked such that the first filter is given the highest rank, and the last filter has the lowest rank.

The FCP scheduling algorithm proceeds from the first filter to the last filter in order, as seen in Figure 4, and both chooses the number of filters and dynamically mutates the filter graph, then maps the copies onto hosts. In line 1, the modeled resources (i.e., CPU power and network bandwidth) are initialized using current system load estimates². The main outer loop processes filters in order, ensuring that when filter f_i is being scheduled, all filters that write to f_i 's input streams have already been scheduled. The number of copies required to satisfy the rate requirements without pipeline stalls is computed and added to the filter graph G_f in lines 3-4. The next loop iterates over all copies of filter f_i , and attempts to schedule each in turn. The loop over hosts finds the mapping that will result in the minimum earliest finish time (EFT) of filter copy $f_{i,c}$. Once chosen, the host $Map()$ function is used to consume resources on the host and network. Finally, the actual scheduled earliest start time (EST), EFT , and earliest write time (EWT) values are computed and stored for use by any successors of f_i in the next stages of the network.

²In the current implementation, this information is obtained from the Network Weather Service [39].

Input: filter network G_f , tuning run statistics, topology graph G_t
Output: G'_f with added copies and host mappings, G'_t resulting resource assignments

FCP-Scheduler(G_f, G_t):

- 1: call InitResourcesFromCurrentLoad(G_t)
- 2: **for each** filter $f_i \in G_f$ in rank order **do**
- 3: Compute $ncopies$
- 4: $G_f.Add(ncopies - 1$ copies of f_i)
- 5: **for each** copy $f_{i,c}$ of f_i **do**
- 6: **if** pinned **then**
- 7: $minhost \leftarrow$ “the specified host”
- 8: **else**
- 9: {choose host for placement}
- 10: **for each** $host_i \in cluster$ **do**
- 11: Compute $tmpEFT$
- 12: **if** $minEFT > tmpEFT$ **then**
- 13: $minEFT \leftarrow tmpEFT$
- 14: $minhost \leftarrow host_i$
- 15: **end if**
- 16: **end for**
- 17: **end if**
- 18: $minhost.Map(f_i)$
- 19: Compute $f_{i,c}.(EST, EFT, EWT)$
- 20: **end for**
- 21: **end for**

Figure 4: Algorithm: FCP-Schedule filters onto the given cluster topology.

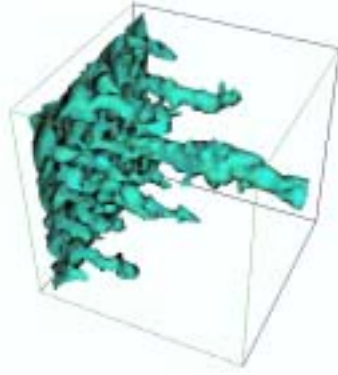
4.3 Cost Model

To determine a good placement, we must decide if one particular organization is superior to another. To answer this question, we must quantitatively evaluate an organization and determine some measure of how good it is. In this work, balancing the pipeline stages is the goal. To compute an estimate of the execution time of an application, we need to compute several quantities for each filter locally, and in summary for the entire filter group. The network between two hosts is modeled by latency (`latency`) and bandwidth (`bw`). If there are multiple hops between two hosts, the bandwidth of the slowest hop and the sum of the latency are used in the cost model.

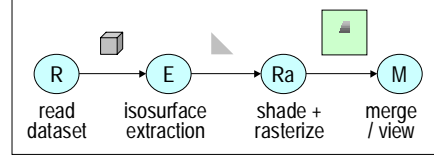
The number of copies required for a filter depends on the production (output) rate of its predecessors that write to the filter’s input streams. The output rate is the number of filter cycles per second the current filter will execute if no stalls occur.

$$rate = \frac{ncy}{EFT - EST} \quad (1)$$

Here, ncy denotes the total number of filter cycles of the given filter. Considering the unscheduled current filter, we compute the aggregate rate from all its predecessors. Then, the number of copies of the current filter ($ncopies$ in Figure 4) is calculated so as to balance the aggregate production rate from all the predecessors of the filter with the consumption rate of the filter using the tuning run statistics. This is needed because we do not know the exact consumption rate and thus the number of copies needed until the current filter is scheduled, and we cannot schedule the current filter until we know the number of copies.



(a) Isosurface rendering of chemical densities in a reactive transport simulation.



(b) Isosurface rendering application modeled as filters.

Figure 5: Isosurface rendering and filter implementation.

Before any filter can start execution, its predecessor constraints need to be satisfied. More precisely, sufficient data from each input stream must be available to start the first local filter cycle for the current filter. The time when enough data has arrived to start a filter cycle is equal to EST . Based on our list-scheduling approach, we know that all predecessors (that the current filter will read from) have already been scheduled, and that we are considering a particular host (H_{dst}) for the current filter. Thus, the EST of a filter is equal to the maximum of the write time of its producer filters. The write time of a producer filter is the EWT of the producer filter plus the time for sending $vcyc$ bytes of data from the producer to the consumer, where $vcyc$ is the volume of data per filter cycle written to the output stream.

Given the filter's EST , the algorithm predicts the EFT value on the host, which will depend on previously assigned load. Let EFT' be the optimal value, in the sense that no stalls are assumed to occur, and EFT'' be the time when the last input data will arrive. Overall EFT will depend on which of those two values is greater. If $EFT' < EFT''$ then we have a read imbalance, and the current filter will stall, hence EFT'' will be used. Otherwise, EFT' is used for EFT . The time when the first piece of data is written dictates the potential amount of pipelining available. The EWT is computed as the actual start time of a mapped filter plus the fractional total time scaled from the tuning run statistics. $STATS.t_{write\ first}$ in the following equation is the time of first write and $STATS.t_{process}$ is the data processing time of the filter, measured during the tuning run. Note that tuning run values are scaled according to the relative power of the hosts in the system.

$$EWT = EST + \left((EFT - EST) * \left(\frac{STATS.t_{write\ first}}{STATS.t_{process}} \right) \right) \quad (2)$$

4.4 Case Study Application: Isosurface Rendering

In this section we briefly described the implementation of the isosurface rendering application [10] used for the evaluation of the scheduling algorithm. Isosurface rendering is a technique for extracting and visualizing surfaces within a 3D volume [24]. Figure 5(a) shows a rendering of the output from a reactive transport simulation. Our implementation consists of a total of four filters (Figure 5(b)).

- A **read (R)** filter reads the volume data from local disk, and writes the 3D voxel elements to its output stream.

- An **extract (E)** filter reads voxels, produces a list of triangles from the voxels, and writes the triangles (the isosurface) to its output stream. We use the marching cubes algorithm [24], which is commonly used for extracting isosurfaces from a rectilinear mesh. The extract filter receives data from the read filter in buffers. A buffer contains a subset of voxels in the dataset. The triangles extracted from each voxel in the current input buffer are written to the output buffer. When the output buffer is full or the entire input buffer has been processed, the output buffer is sent to the raster filter. This organization allows the processing of voxels to be pipelined. Multiple copies of the extract filter can be instantiated and executed concurrently.
- A **rasterize (Ra)** filter reads triangles from its input stream, renders the triangles into a two-dimensional image from a particular viewpoint, and writes the image to its output stream. We employ a hidden-surface removal method, called *active pixel* rendering [22, 10]. This algorithm utilizes a modified z-buffer scheme to store foremost pixels in the output buffer efficiently. Processing of triangles is pipelined and multiple copies of the raster filter can be executed.
- A **merge (M)** filter is used to composite the partial results from multiple rasterize filters to form the final output image. The merge filter also sends the final image to the client for display.

4.5 Experimental Results

In this section we present an experimental evaluation of the scheduling approach using the isosurface rendering application. The hardware configuration used for the experiments consists of a collection of three Linux PC clusters. The first cluster, referred to here as **ROGUE**, is a disk-based storage cluster at the University of Maryland. This cluster consists of 50 nodes, each of which has Pentium III 650MHz processors, 786MB memory, and two 75GB IDE disks. The nodes are interconnected via Switched Fast Ethernet. We had access to 14 nodes of this cluster during the experiments. The second cluster, referred to here as **DC**, is located at Biomedical Informatics Department at Ohio State University. The DC cluster is composed of 5 Pentium 4 1.8GHz processors. Each node has 512MB memory and 160GB disk space and is connected to other nodes by Switched Fast Ethernet. The third cluster, referred to here as **OSUMED**, is made up of 24 Linux PCs and hosted at the Ohio Supercomputer Center. Each node has 512MB main memory and three 100GB IDE disks. The nodes are inter-connected via Switched Fast Ethernet. We used 18 nodes of this cluster for the experiments. All clusters are connected to each other over wide-area networks.

The FCP scheduler does not model WAN connections as shared resources, and for this reason it can prematurely attempt to place application filters on either side of the WAN link leading to bad and spurious performance results. In light of this, we have restricted the scheduler to placing filter groups completely within clusters for the experiments. In the application frontend, a UOW is scheduled (i.e., the number of transparent copies and the placement of filters are computed) to clusters one by one, using the filter group layout, the topology of the cluster, and the resource availability information.

We utilized the Network Weather Service (NWS) [39] to monitor the resources (i.e., network bandwidth and CPU availability) on all the clusters. Our first setup consisted of a single NWS server, with sensors activated on all the nodes in the system. The application frontend queried the NWS server for the CPU availability of each node and the bandwidth and latency between every pair of nodes in each cluster, submitting separate requests to the NWS server for each node and each pair of nodes. We observed that with this setup it took a large amount of time to gather information from the NWS, and the time to query the NWS varied significantly. To alleviate the large variance in the NWS query times and to reduce the querying time, we ran a separate NWS server on the frontend node of each cluster, with sensors reporting to the local NWS server only. The NWS sensors were instantiated to monitor the resources with periods of 7 and 20 seconds for the CPU and network, respectively. The measurements were used to determine the available CPU on any



Figure 6: Single query execution time with and without copies. EET is the estimated execution time by the scheduler.

given node, as well as the TCP bandwidth and latency between any two nodes in a given cluster. We colocated a multi-threaded proxy with every NWS server. The proxy queried the local NWS server at regular intervals (for the experiments, we set this value to 10 seconds) and gathered information about the resource availability in the corresponding cluster. In this setting, the application frontend sent the resource availability requests to the proxies colocated with each NWS server. With this setup, it took 0.15 seconds on average for the application frontend to inquire the availability of the resources in the environment.

For the isosurface application, we employed a 23GB dataset, consisting of simulation output over 10 time steps, created by a reactive chemical transport simulator, called ParSSim³, developed at the Texas Institute for Computational and Applied Mathematics at the University of Texas. The dataset was partitioned into 64 files. These files were replicated on all the clusters in the system and were distributed in a round-robin fashion across four, two and eight nodes in the ROGUE, DC and OSUMED clusters, respectively. Application queries were generated so that each query required the visualization of 5 isosurfaces in a single time step and a 2048x2048 output image was produced. The queries were generated at a uniform rate within an 8-minute period. That is, with 18 queries, a query was generated for execution at every 28.2 seconds, whereas with 60 queries, a query was generated at every 8.1 seconds.

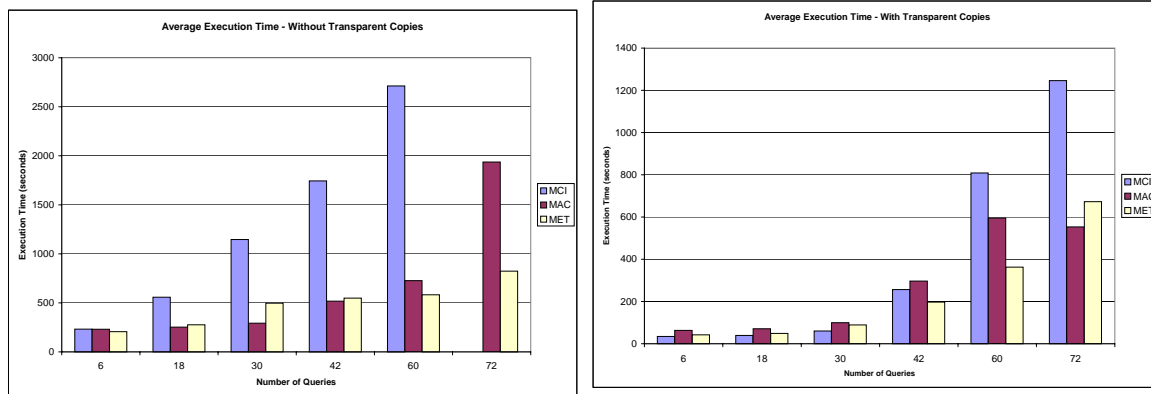
Figure 6 displays the Estimated Execution Time (EET) computed by the scheduler and the actual query execution time on each cluster. As seen in the figure, the EET estimate is very close to the actual execution time on each cluster, with and without transparent copies. We also note that using transparent copies significantly improves performance. The demand-driven mechanism was used for scheduling of buffers among the copies of a filter (see Section 2). On the ROGUE and OSUMED clusters transparent copies speeds up the single query execution approximately 5 times. The speedup on the DC cluster is about 2.5, because the DC cluster is much smaller compared to the ROGUE and OSUMED clusters.

Figure 7 shows the performance numbers for average query execution time, when the number of queries submitted to the system is varied. We implemented three different strategies for scheduling multiple queries (units of work).

Minimum Concurrent Instances (MCI). In this strategy, a query is executed within each cluster using a fixed filter placement. The scheduler keeps track of the number of active filter group instances in each cluster. The cluster with the fewest active filter group instances is chosen to execute the query.

Maximum Available Cluster (MAC). The cluster with the maximum average CPU availability is selected. The maximum CPU availability is computed as the average CPU availability of all the nodes

³<http://www.ticam.utexas.edu/~arbogast/parssim>



(a) Without Transparent Copies

(b) With Transparent Copies

Figure 7: Average query execution time. (a) Without transparent copies (b) With transparent copies.

in the cluster. A dynamic schedule (the number of copies and the placement of copies) based on the CPU availability per node and network bandwidth between the nodes in the cluster is generated by the scheduler.

Minimum Estimated Execution Time (MET). In this strategy, a dynamic schedule is generated for each cluster and the corresponding EET is computed by the scheduler. The cluster with the minimum EET is selected to run the query.

In all of these strategies, a new filter group instance is instantiated and executed with the given placement information for each query. The demand-driven mechanism is used for scheduling buffers among the copies of a filter.

As is seen from Figure 7(a), both MAC and MET perform much better than MCI, when the queries are scheduled with no transparent copies. When no transparent copies are allowed, only the placement of filters is dynamically computed in MAC and MET. Although MCI achieves comparable performance to that of the two other strategies for small number of queries, its performance becomes worse quickly and the performance gap between MCI and the other two schemes increases as the number of queries is increased. The main reason is the fact that MCI suffers from the fixed placement of filters, i.e., the same filters in different filter group instances are placed on the same nodes. As more queries are generated, the nodes are likely overloaded – we were not able to run the MCI scheme with 72 queries because of this.

Figure 7(b) shows the performance figures when queries are scheduled with transparent copies. In this experiment, the fixed placement for MCI was created manually for each cluster in the environment, based on our knowledge of the application and the tuning run results. As is seen from the figure, the performance of MCI is similar to that of MAC and MET for up to 60 queries. We attribute this behavior to two factors. First, as noted earlier the demand-driven buffer scheduling mechanism of DataCutter is employed. This mechanism sends more data buffers to filter copies on less loaded hosts, thus achieving a load balance among the copies of a filter. The fixed placement uses the same set of nodes for each filter group instance. Hence, some nodes may be more loaded than the other nodes depending on the query. However, the demand-driven mechanism compensates for this variation by dynamically distributing buffers to the filter copies on less loaded nodes. Second, queries execute faster with transparent copies. As a result, the number of active filter groups on a cluster at any given time is fewer than when no copies are allowed. However, we observe that as the number of queries is increased beyond 42, the MCI cannot cope with overloading of the system. In that case, MAC and MET performs better. On the average, MET achieves the best performance among all three strategies.

Input: filter network G_f , tuning run statistics, topology graph G_t
Output: G'_f with added copies and host mappings, G'_t resulting resource assignments

BFC-Scheduler(G_f, G_t):

```

1: Compute  $m_i$ , maximum allowed number of copies, for each filter  $f_i \in G_f$ 
2: call InitResourcesFromCurrentLoad( $G_t$ )
3: for each filter  $f_i \in G_f$  in rank order  $i$  do
4:   Compute production rate  $P_{i-1}$ 
5:   while  $P_{i-1} > 0$  and  $n_i < m_i$  do
6:      $rate \leftarrow 0$ 
7:     for each node  $h \in G_t$  do
8:       Compute scaled consumption rate  $scr$  of filter  $f_i$  on  $h$ 
9:       Compute aggregated network bandwidth  $bw$  to  $h$  from all producing filters  $f_{i-1}$ 
10:      Compute consumption rate  $cr \leftarrow \min(scr, bw)$ 
11:      if  $cr > rate$  then
12:         $rate \leftarrow cr$ 
13:         $selected\_node \leftarrow h$ 
14:      end if
15:    end for
16:    Create and place a copy of the filter  $f_i$  to  $selected\_node$ 
17:     $n_i \leftarrow n_i + 1$ 
18:    Remove  $selected\_node$  from the free list
19:    Update Network and CPU usage
20:     $P_{i-1} \leftarrow P_{i-1} - rate$ 
21:  end while
22: end for

```

Figure 8: Pseudo-code for the BFC scheduling algorithm.

5 Inter-Cluster Scheduling: Balanced Filter Copies (BFC) Algorithm

Although the FCP algorithm is designed for pipelined filter operations, it does not account for the sharing of wide area networks. To address this issue and to highlight the importance of modeling the shared WAN link, we have implemented a second algorithm, called Balanced Filter Copies (BFC).

5.1 Balanced Filter Copies Algorithm

This algorithm is similar to the FCP algorithm in that it performs a single sweep over the filter group (filter network). As was described in Section 4, FCP uses a detailed compute cycle model for pipelined filter computations. BFC, on the other hand, simply tries to balance the consumption and production rates of consumer and producer filters, but it takes into account shared network resources.

The algorithm starts from the first filter in the chain and chooses a mapping for possibly multiple transparent copies of the filter and continues choosing mappings until it reaches the last filter. In deciding how many transparent copies will be created, it tries to balance production rate of the previous filter and the consumption rate of the current filter. Pseudo code for the BFC algorithm is displayed in Figure 8.

In order to model shared network resources, the algorithm tracks the network usage of each stage of the filter network, and by reducing the available bandwidth on all network segments as it is consumed. Thus

attempting to place multiple consumer filter on a remote cluster over a slow WAN link is discouraged in this algorithm. At any step of scheduling, we stop creating and placing the copies of a filter either when we have consumed all the production rate from the previous stage, or when we reach a maximum allowable threshold for the current stage. This threshold is discussed in greater detail below. Once a filter has been placed on a specified processor, that processor is marked unavailable for use by subsequent filters in the current filter group. We should note that this is another difference of BFC from FCP, which allows multiple filters of a filter group to be placed on the same processor. In this respect, the two algorithms are complementary; one models shared CPU, while the other models shared network resources.

In the BFC algorithm, we calculate the total CPU power available in all clusters. We then divide the total available power among the filters to be placed based on the computation time taken by each filter in a tuning run – note that the tuning run has a single copy of each filter placed on separate nodes on a single cluster. For example, if a particular filter accounted for 10% of the total computation time in the tuning run, it would be assigned a maximum of 10% of the available CPU power in subsequent placements by the scheduler. This then sets a maximum allowed power that may be assigned to any individual filter in the layout during scheduling. When this power threshold is reached the scheduler stops placing the copies of the filter and moves onto the next stage in the pipeline. This approach has several consequences. First, it makes the algorithm greedy. Almost all available processors will be utilized for scheduling, if necessary. In our experiments, however, we seldom saw this occur. Second, it allows us to perform scheduling in a single pass, since we are guaranteed that by the time we reach the last filter stage, we have sufficient remaining processing power to place at least a limited number of copies of the last filter. Third, if a particular stage of the schedule has not consumed all of the output from a previous stage, and the maximum power for the current stage has been reached, the algorithm will stop placing additional copies and move onto the next stage. In this way, the BCP algorithm lacks the ability to add copies where they will most help. In the experiments, we observed that this does indeed prevent us from achieving near optimal placements. We intend to address this issue by refining the algorithm in future. Finally, since the algorithm lacks any look-ahead or iterative methodologies, it can occasionally be forced to schedule a filter on a cluster in such a way that produces sub-optimal results. For instance, if we reach the final stage, and we find that two producer filters from the previous stage have been placed on two separate clusters across a slow WAN link, we may be forced to send data across that link which may result in a high amount of contention. In this case, one would ideally want to go back and accept an inferior placement at an earlier step of scheduling in order to achieve better overall performance. Again this is an issue we will look to address in a future work. Despite some known limitations of this algorithm, the algorithm is very simple to apply, and it succeeds in certain cases where the FCP algorithm fails, as will be shown in the experimental results.

5.2 Experimental Results

In this section we present preliminary results for cross-cluster (inter-cluster) scheduling using the BFC algorithm. We used the same visualization application, with the same datasets, as described in Sections 4.4 and 4.5. Currently, in order to perform cross-cluster filter placement the DataCutter runtime system requires that all nodes within a cluster are fully accessible from all external clusters. At this time the OSUMED nodes have only internal IP addresses, and they are only accessible from a front-end node. For this reason we restricted our experiments to the ROGUE and DC clusters. In the experiments, we used a single NWS server, but started three groups of NWS sensors to measure the availability of resources. The first group of sensors were placed on the ROGUE cluster and monitored the intra-cluster network bandwidth and CPU availability. The second group were executed on the DC cluster and measured the resource availability within the cluster. The third group consisted of two network sensors, each placed on one of the nodes of the ROGUE and DC clusters. These sensors were used to monitor cluster-to-cluster WAN network bandwidth. A proxy was colocated with

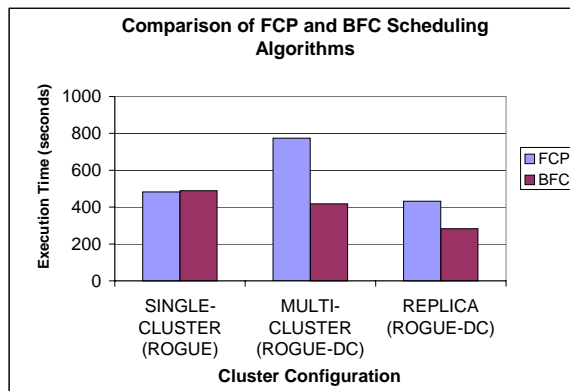


Figure 9: Performance comparison of FCP and BFC algorithms. Five queries are scheduled using a single cluster and two clusters (ROGUE and DC) with and without dataset replicas. SINGLE CLUSTER shows the average execution time of the five queries when only the ROGUE cluster is used for running the filters, i.e., no cross-cluster scheduling of filters is allowed. MULTI-CLUSTER is the configuration with one dataset copy on the ROGUE cluster, but filters can be placed on both clusters. REPLICA denotes the configuration in which the dataset is replicated on both clusters and both algorithms can schedule filters across the clusters.

the NWS server.

In our experiments, we observed that both schedulers avoided WAN links unless the availability of resources in the local cluster was low, i.e., until the cost of traversing the WAN link was approximately equal to the cost of staying in the local cluster, as expected. In the case of the link between ROGUE and DC, NWS measured the available bandwidth as approximately 5-7Mbps, and all internal bandwidths were effectively 90Mbps. In this configuration, we have to put a significant load on a cluster before the schedulers will attempt to cross the WAN link. To emulate such a situation, we ran background jobs on the ROGUE cluster specifically designed to reduce the available network bandwidths between the nodes until they were lower than that of the WAN link. Our experiments thus target the situation in which a system is shared and loaded by other applications, or use of local available resources is restricted (e.g., through a system policy). We partitioned the isosurface dataset into four disjoint sets on four separate ROGUE nodes, and into two disjoint sets on two DC nodes. We experimented with three different configurations. The first configuration allows the scheduler to use a single cluster only (SINGLE-CLUSTER in Figure 9). The second configuration restricts the scheduler to only use the ROGUE dataset, but allow filters to be placed on both ROGUE and DC nodes, if necessary (MULTI-CLUSTER). The third configuration allows the scheduler to choose between the dataset replicas on ROGUE and DC, and to schedule the filters using all nodes on both clusters (REPLICA). In each experiment, five queries were submitted to the system at a uniform rate within 100 seconds. We repeated each experiment six times, and the timing results in Figure 9 are average execution times over six runs.

As seen in Figure 9, both algorithms perform approximately equally when restricted to execute within a single cluster. However, when presented with multiple clusters and a single replica of the dataset on a low-resource cluster, the FCP algorithm tends to cross the WAN link in order to take advantage of the idle processors on a remote cluster. In that case, it typically places multiple Extract filters on the DC cluster, assuming that each of the Read-to-Extract (R-E) streams use a dedicated point-to-point link. In fact each producer-consumer stream shares the WAN link. The BFC scheduler recognizes that multiple R-E streams will have to share the WAN link, and it typically avoids this pitfall.

In the experiment where we allow the schedulers to choose the data replica to use, and also allow them to cross the WAN link, we see that the new BFC scheduler outperforms the FCP scheduler, and also that

the BFC scheduler produces the overall best result in this case. This is because of two reasons: (i) the DC cluster starts in an idle state relative to the ROGUE cluster, and (ii) once again the FCP algorithm attempts to prematurely utilize nodes on remote clusters, thus paying a penalty for over-using the WAN link.

These experiments show that under certain circumstances we want to allow schedulers to do filter placements that cross otherwise expensive WAN links to take advantage of remote grid resources (typically when the local resources are unavailable or overloaded), and that when we do so we must be careful to account for the shared nature of the WAN link. In addition, the overall best situation arises when one can utilize replicated copies of a source dataset, as well as available grid resources.

6 Related Work

A large part of classic work on scheduling for parallel machines is derived from Sarkar [31], and later work such as Yang and Gerasoulis [40]. The goal on distributed memory parallel machines is to trade-off parallelism with communication. When we introduce heterogeneity, many of these existing scheduling techniques break down due to implicit homogeneity assumptions. Many techniques have been developed for scheduling in heterogeneous computing systems [16, 36, 38, 25, 11, 21]. Some deal with a single application structured as a DAG, while others apply to globally scheduling many independent tasks. Our approach for scheduling a UOW is loosely based on the list-scheduling heuristic approach [23] that DAG scheduling uses, which can result in sub-optimal mappings. Several groups have investigated scheduling methods encountered when using network enabled servers in multi-client, multi-server scenarios [4, 12, 29, 35, 34, 41]. While network enabled servers can be composed, typically applications consist of a large number of more or less independent tasks [1, 3, 17, 26, 30, 32, 33].

One major difference of our approach is that the application processing structure is implemented as a set of interacting components. Hence, although units of work received from different clients are handled independently, each unit of work is a data flow network that creates intra-unit of work dependencies. Another difference of our work is that our task graphs are mutable, i.e., the scheduler is allowed to modify the filter network dynamically by adding and deleting copies during the scheduling process. In the standard scheduling problem formulation, the computation graph is a predetermined, fixed graph and is not modified by the scheduler. Moreover, most existing methods do not target pipelined computations on data.

7 Conclusions and Future Work

We have presented an approach for scheduling multiple queries, of which the processing structure can be represented as a pipelined chain of operations on data, in a Grid-based cluster environment. Our preliminary results show that the proposed scheduling algorithms achieve good performance, especially when the number of queries is large. We plan to further examine the performance of the strategies presented in this paper using different query workloads and other applications.

We plan to improve on the scheduling algorithms presented in this paper to address several issues in future: 1) We plan to investigate methods to account for I/O contention effects in the model, which becomes increasingly important as queries overlap. 2) In the current implementation, queries are scheduled one after another. In a multi-client environment, clients may submit requests in batches or the server may receive multiple queries simultaneously and may need to serve them in a batch. For scheduling a batch of queries, several different strategies can be employed. For example, a single multi-pipeline graph from the set of filter groups can be created and the scheduling algorithm can be applied on this graph. 3) When a new query is to be scheduled for execution, the corresponding filter group can be scheduled onto all available machines in the system. However, the main disadvantage of this approach is that an expensive query may be scheduled

to consume most of the resources to evaluate it as fast as possible. If we have a priori knowledge of potential future queries, it may be beneficial to limit the scheduling of filter groups.

During these studies it also became clear that for large numbers of queries submitted in a short time frame, it is important to have up-to-date resource information for dynamic scheduling. When the resource measurements become stale in a time that is significant relative to the rate of query submission, the scheduler is prone to make poor decisions. This is particularly true if some clusters are significantly less powerful than others in a given resource. It may be correct to place a query on a slow cluster, however if the resource measurements are not updated fast enough, placing more queries there may lead to dramatic slow-down due to contention for sparse resources.

Acknowledgments. We would like to express our gratitude to the anonymous reviewers of our paper. Their comments helped us improve the content and presentation of the paper significantly.

References

- [1] D. Abramson, M. Cope, and R. McKenzie. Modeling photochemical pollution using parallel and distributed computing platforms. In *Proceedings of PARLE-94*, pages 478–489, 1994.
- [2] B. Allcock, J. Bester, J. Bresnahan, A. L. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel, and S. Tuecke. Secure, efficient data transport and replica management for high-performance data-intensive computing. In *Proceedings of IEEE Mass Storage Conference*, April 2001.
- [3] J. Basney, M. Livny, and P. Mazzanti. Harnessing the capacity of computational grids for high-energy physics. In *Conference on Computing in High Energy and Nuclear Physics*, 2000.
- [4] F. Berman and R. Wolski. The AppLeS project: A status report. In *Proceedings of the 8th NEC Research Symposium*, Berlin, Germany, May 1997.
- [5] M. Beynon. *Supporting Data Intensive Applications in a Heterogeneous Environment*. PhD thesis, University of Maryland, 2001.
- [6] M. Beynon, C. Chang, U. Catalyurek, T. Kurc, A. Sussman, H. Andrade, R. Ferreira, and J. Saltz. Processing large-scale multidimensional data in parallel and distributed environments. *Parallel Computing*, 2002. To appear in special issue on Data Intensive Computing.
- [7] M. Beynon, T. Kurc, A. Sussman, and J. Saltz. Design of a framework for data-intensive wide-area applications. In *Proceedings of the 9th Heterogeneous Computing Workshop (HCW2000)*, pages 116–130. IEEE Computer Society Press, May 2000.
- [8] M. D. Beynon, T. Kurc, U. Catalyurek, C. Chang, A. Sussman, and J. Saltz. Distributed processing of very large datasets with DataCutter. *Parallel Computing*, 27(11):1457–1478, Oct. 2001.
- [9] M. D. Beynon, T. Kurc, U. Catalyurek, A. Sussman, and J. Saltz. A component-based implementation of iso-surface rendering for visualizing large datasets. Technical Report CS-TR-4249 and UMIACS-TR-2001-34, University of Maryland, Department of Computer Science and UMIACS, May 2001.
- [10] M. D. Beynon, T. Kurc, U. Catalyurek, A. Sussman, and J. Saltz. Efficient manipulation of large datasets on heterogeneous storage systems. In *Proceedings of the 11th Heterogeneous Computing Workshop (HCW2002)*. IEEE Computer Society Press, April 2002.
- [11] T. D. Braun, H. J. Siegel, N. Beck, L. L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. Hensgen, and R. F. Freund. A comparison study of static mapping heuristics for a class of meta- tasks on heterogeneous computing systems. *Journal of Parallel and Distributed Computing*, 61:810–837, 2001.
- [12] R. Buyya, D. Abramson, and J. Giddy. An economy driven resource management architecture for global computational power grids. In *Proceedings of PDPTA 2000*, June 2000.
- [13] Common Component Architecture Forum. <http://www.cca-forum.org>.

- [14] C. Chang, T. Kurc, A. Sussman, and J. Saltz. Optimizing retrieval and processing of multi-dimensional scientific datasets. In *Proceedings of the Third Merged IPPS/SPDP (14th International Parallel Processing Symposium & 11th Symposium on Parallel and Distributed Processing)*. IEEE Computer Society Press, May 2000.
- [15] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. Grid information services for distributed resource sharing, 2001.
- [16] M. M. Eshaghian and Y. C. Wu. Mapping heterogeneous task graphs onto heterogeneous system graphs. In *Proceedings of the 6th Heterogeneous Computing Workshop*, pages 147–160, Geneva, Switzerland, April 1997. IEEE Computer Society Press.
- [17] A. Gallant and G. Tauchen. SNP: A program for nonparametric time series analysis. Technical report, Duke Economics Working Paper No:95-26, Duke University, 1997.
- [18] Global Grid Forum. <http://www.gridforum.org>.
- [19] The Globus Project. <http://www.globus.org>.
- [20] C. Isert and K. Schwan. ACDS: Adapting computational data streams for high performance. In *14th International Parallel & Distributed Processing Symposium (IPDPS 2000)*, pages 641–646, Cancun, Mexico, May 2000. IEEE Computer Society Press.
- [21] M. Iverson and F. Ozguner. Dynamic, competitive scheduling of multiple dags in a distributed heterogeneous environment. In *Proceedings of the 7th Heterogeneous Computing Workshop*, Orlando, FL, March 1998. IEEE Computer Society Press.
- [22] T. Kurc, C. Aykanat, and B. Ozguc. Object-space parallel polygon rendering on hypercubes. *Computers & Graphics*, 22(4):487–503, 1998.
- [23] Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs. *ACM Computing Surveys*, 31(4):406–471, Dec. 1999.
- [24] W. Lorensen and H. Cline. Marching cubes: a high resolution 3d surface reconstruction algorithm. *Computer Graphics*, 21(4):163–169, 1987.
- [25] M. Maheswaran and H. J. Siegel. A dynamic matching and scheduling algorithm for heterogeneous computing systems. In *Proceedings of the 7th Heterogeneous Computing Workshop*, Orlando, FL, March 1998. IEEE Computer Society Press.
- [26] A. Majumdar. Parallel performance study of monte carlo photon transport code on shared-, distributed-, and distributed-shared-memory architectures. In *Proceedings of the 14th Parallel and Distributed Processing Symposium (IPDPS'00)*, pages 93–99, 2000.
- [27] R. Oldfield and D. Kotz. Armada: A parallel file system for computational. In *Proceedings of CCGrid2001: IEEE International Symposium on Cluster Computing and the Grid*, Brisbane, Australia, May 2001. IEEE Computer Society Press.
- [28] B. Plale and K. Schwan. dQUOB: Managing large data flows using dynamic embedded queries. In *IEEE International High Performance Distributed Computing (HPDC)*, August 2000.
- [29] J. Plank, R. Wolski, J. Brevik, and T. Bryan. G-Commerce: The study and building of computational economies for the computational grid. In *Workshop on Clusters and Computational Grids for Scientific Computing*, 2000.
- [30] A. Rogers, M. C. Carlisle, J. H. Reppy, and L. Hendren. Supporting dynamic data structures on distributed-memory machines. *ACM Transactions on Programming Languages and Systems*, 17(2):233–263, Mar. 1995.
- [31] V. Sarkar. Determining average program execution times and their variance. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 298–312. ACM Press, June 1989.
- [32] S. Sciutto. AIRES users guide and reference manual, version 2.0.0. Technical Report GAP-99-020, Auger Project, 1999.

- [33] J. Stiles, T. Bartol, E. Salpeter, and M. Salpeter. Monte Carlo simulation of neuromuscular transmitter release using MCell, a general simulator of cellular physiological processes. *Computational Neuroscience*, pages 279–284, 1998.
- [34] A. Takefusa. Bricks: A performance evaluation system for scheduling algorithms on the grids. In *JSPS Workshop on Applied Information Technology for Science (JWAITS 2001)*, 2001.
- [35] A. Takefusa, H. Casanova, S. Matsuoka, and F. Berman. A study of deadline scheduling for client-server systems on the computational grid. In *Proceedings of 10th IEEE International Symposium on High Performance Distributed Computing (HPDC-10)*, pages 406–415, 2001.
- [36] H. Topcuoglu, S. Hariri, and M.-Y. Wu. Task scheduling algorithms for heterogeneous processors. In *Proceedings of the 8th Heterogeneous Computing Workshop*, pages 3–14, San Juan, Puerto Rico, April 1999. IEEE Computer Society Press.
- [37] S. Vazhkudai, S. Tuecke, and I. Foster. Replica selection in the globus data grid, 2001.
- [38] J. B. Weissman and X. Zhao. Run-time support for scheduling parallel applications in heterogeneous nodes. In *Proceedings of the 6th High Performance Distributed Computing Conference*, August 1997.
- [39] R. Wolski, N. Spring, and J. Hayes. The network weather service: A distributed resource performance forecasting service for metacomputing. *Journal of Future Generation Computing Systems*, 15(5-6):757–768, 1999.
- [40] S. Yang, D. Gannon, S. Srinivas, and F. Bodin. High Performance Fortran interface to the Parallel C++. In *Proceedings of the Scalable High Performance Computing Conference (SHPCC-94)*, pages 301–308. IEEE Computer Society Press, May 1994.
- [41] T. Zhao and V. Karamcheti. Expressing and enforcing distributed storage sharing agreements. In *Proceedings of SC2000*, 2000.