A Component-based Implementation of Iso-surface Rendering for Visualizing Large Datasets *
(Extended Abstract)

Michael D. Beynon[†], Tahsin Kurc[‡], Umit Catalyurek[‡], Alan Sussman[†], Joel Saltz[†‡]

† Institute for Advanced Computer Studies
and
Dept. of Computer Science
University of Maryland
College Park, MD 20742

‡ Dept. of Pathology
Johns Hopkins Medical Institutions
Baltimore, MD 21287

{beynon,kurc,umit,als,saltz}@cs.umd.edu

1 Introduction

Isosurface rendering is a technique for extracting and visualizing surfaces within a 3D volume. It is a widely used visualization method in many application areas. For instance, in a typical analysis of datasets generated by environmental simulations, a scientist examines the transport of one or more chemicals in the region being studied. Iso-surface rendering is a method that is well-suited for visualizing the density distributions of chemicals in the region. Figure 1 shows a rendering of the output from a reactive transport simulation.

A number of research projects [2, 5, 6, 11] have examined algorithms and methods for the visualization of large, out-of-core datasets on workstations and parallel machines. Recent research on programming models for developing applications in a distributed environment has focused on the use of component-based models [8], in which an application is composed of multiple interacting computational objects. In this paper, we describe a component-based implementation of isosurface rendering for visualizing very large datasets in a distributed, heterogeneous environment. We use DataCutter [3, 4], a component framework that supports subsetting and user-defined processing of large multi-dimensional datasets in a distributed environment. We present experimental results on a heterogeneous collection of multiprocessor machines.

2 DataCutter

DataCutter provides a framework, called filter-stream programming, for developing data-intensive applications that execute in a distributed, heterogeneous environment. In this framework, the application processing structure is implemented as a set of components, called filters. Data exchange between filters is performed through a stream abstraction.

*This research was supported by the National Science Foundation under Grants #ACI-9619020 (UC Subcontract #10152408) and #ACI-9982087, the Office of Naval Research under Grant #N6600197C8534, Lawrence Livermore National Laboratory under Grant #B500288 (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).

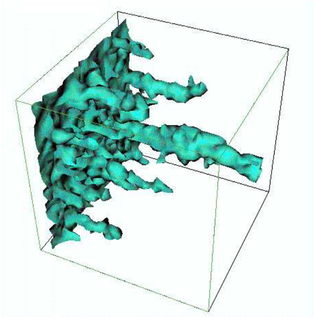
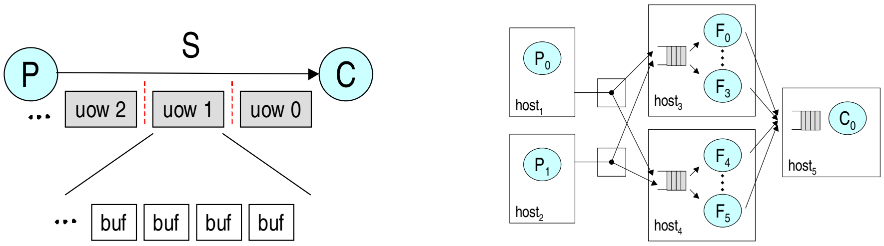


Figure 1: Isosurface rendering of chemical densities in a reactive transport simulation.



(a) Data buffers and end-of-work markers on a stream.

(b) P,F,C filter group instantiated using transparent copies.

Figure 2: DataCutter stream abstraction and support for copies.

2.1 Filters and Streams

A *filter* is a user-defined object that performs application-specific processing on data. Currently, filter code is expressed using a C++ language binding by sub-classing a filter base class. The well-defined interface for filters consists of init, process, and finalize callback functions. A *stream* is an abstraction used for all filter communication, and specifies how filters are logically connected. A stream provides the means of data flow between two filters. All transfers to and from streams are through a provided buffer abstraction. Streams transfer data in fixed size buffers. The current prototype implementation uses TCP for point-to-point stream communication.

Filter operations progress as a sequence of cycles, with each cycle handling a single application-defined *unit-of-work* (UOW). An example of a UOW would be rendering of a simulation dataset from a particular viewing direction. A work cycle starts when the filtering service calls the filter **init** function, which is where any required resources such as memory can be pre-allocated. Next the **process** function is called to read from any input streams, work on data buffers received, and write to any output streams. A special marker is sent by the runtime system after the last buffer to mark the end for the current UOW (see Figure 2(a)). The **finalize** function is called after all processing is finished for the current UOW, to allow release of allocated resources such as scratch space. The interface functions *may* be called again to process another UOW.

2.2 Support for Parallelism: Transparent Filter Copies

Once the application processing structure has been decomposed into a set of filters, it is possible to use multiple filters for implementing a pipeline of processing on data as it progresses from data sources to clients. The choice of filter placement is a primary degree of freedom in affecting application performance. Good performance can be obtained by placing filters with affinity to data sources near the sources, minimizing communication volume on slow links, and placing filters to deal with heterogeneity [3]. Note that pipelining works well when all stages are balanced, both in terms of relative processing time of the stages, as well as the time of each stage compared to the communication cost between stages. Oftentimes, the processing of filter-based applications is not well balanced, which results in bottlenecks that cause other filters before and after a bottleneck filter to become idle. This imbalance and resulting performance penalty can be addressed using another degree of freedom, parallelism, by executing multiple copies of a single filter across a set of host machines. The runtime performance optimizations target the combined use of ensembles of distributed-memory systems and multiprocessor machines. In DataCutter, we provide support for *transparent copies*, in which the filter is unaware of the concurrent filter replication. We define a *copy set* to be all transparent copies of a given filter that are executing on a single host.

The filter runtime system maintains the illusion of a single logical point-to-point stream for communication between a logical producer filter and a logical consumer filter. When the logical producer or logical consumer is transparently copied, the system must decide for each producer which copy to send a stream buffer to. For example, in Figure 2(b), if copy P_1 issues a buffer write operation to the logical stream that connects filter P to filter F, the buffer can be sent to the copy set on $host_3$ or $host_4$. Each copy set shares a single buffer queue which provides demand-based balance within a single host. For distribution between copy sets (different hosts), we consider several policies: (1) Round Robin (RR) distribution of buffers among copy sets, (2) Weighted Round Robin (WRR) among copy sets based on the number of copies on that host, (3) a Demand Driven (DD) sliding window mechanism based on buffer consumption rate.

Not all filters will operate correctly in parallel as transparent copies, because of internal filter state. For example, a filter that attempts to compute the average size of all buffers processed for a unit of work will not arrive at the correct answer, because only a subset of all buffers for the unit-of-work were seen at any one copy, hence the internal sum of buffer sizes is less than the true total. Such cases can be annotated to prevent the runtime system from utilizing transparent copies, or an additional application-specific *combine* filter can be appended to merge partial results into the final output.

3 Isosurface Rendering

In isosurface rendering, we are given a three-dimensional grid with scalar values at grid points and a user-defined scalar value, called the iso-surface value. The visualization algorithm has two main steps. The first step extracts the surface on which the scalar value is equal to the iso-surface value. The second step renders the extracted surface (iso-surface) to generate an image. In the component architecture, each of these steps is implemented as a filter. The filter-based implementation consists of a total of four filters (Figure 3). 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 to its output stream. A **raster** (**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. During processing, multiple transparent copies of the raster filter can be executed. In that case, an image containing a portion of the rendered triangles will be generated by each copy of the raster filter. Therefore a **merge** (**M**) filter is used to composite the partial results to form the final output image. The merge filter also sends the final image to the client for display.

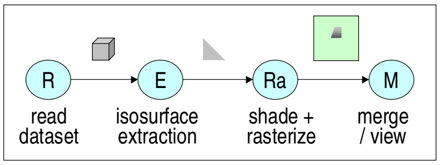


Figure 3: Isosurface rendering application modeled as filters.

3.1 Extract Filter

Our implementation uses the marching cubes algorithm [10], which is a commonly used algorithm for extracting iso-surfaces from a rectilinear mesh. In this algorithm, volume elements (voxels) are visited one by one. At each voxel, the scalar values at the corners of the voxel are compared to the iso-surface value. If the scalar values are all less than or all greater than the iso-surface value, no surface passes through the voxel. Otherwise, using the scalar values at the corners, the algorithm generates a set of triangles that approximate the surface passing through the voxel. The triangles from the current voxel are added into a list of triangles.

The extract filter receives data from the read filter in fixed size buffers. A buffer contains a subset of voxels in the dataset. Note that each voxel is processed independently in the marching cubes algorithm. Therefore, the extract filter can carry out iso-surface extraction on the voxels in a buffer as soon as the buffer is read from its input stream. Moreover, multiple copies of the extract filter can be instantiated and executed concurrently. The data transfer between the extract filter and the raster filter is also performed using fixed-size buffers. The extract filter writes the triangles extracted from each voxel in the current input buffer 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.

3.2 Raster Filter

The raster filter performs rendering of the triangles received in fixed size buffers from the extract filter. First, the triangles in the current input buffer are transformed from world coordinates to viewing coordinates (with respect to the viewing parameters). The transformed triangles are projected onto a 2-dimensional image plane (the screen), and clipped to the screen boundaries. Then, the filter carries out hidden-surface removal and shading of triangles to produce a realistic image. Our implementation used the Gouraud shading method [12]. Hidden-surface removal determines which polygon is visible at a pixel location on the image plane. We now discuss two hidden-surface removal methods: *z-buffer* rendering [12] and *active pixel* rendering [9]. For the following discussion, a pixel location (x,y) on the image plane is said to be *active* if at least one pixel is generated for that location. Otherwise, it is called an *inactive* pixel location.

3.2.1 Z-buffer Rendering

In this method, a 2-dimensional array, called a *z-buffer*, is used for hidden-surface removal. Each z-buffer entry corresponds to a pixel in the image plane, and stores a color value (an RGB value) and a z-value. The (z-value,color) pair stores the distance and color of the foremost polygon at a pixel location. A full z-buffer is allocated and initialized in the **init** method of each instantiated copy of the raster filter.

The z-buffer algorithm initially inserts triangles from the input buffer into a *y-bucket structure*, which is an array (of size N, where N is the y-resolution of the screen) of linked lists. A polygon is inserted into the linked list at the y-bucket location corresponding to the lowest scanline that intersects the on-screen projection of the polygon. A scanline corresponds to a row of pixels on the screen. After this initialization step, hidden-surface removal is performed in scanline-order starting from the lowest numbered scanline. The triangles in the y-bucket of the current scanline and the triangles whose y-extent overlaps the scanline are processed. The intersection of a triangle with the current scanline creates line segments, called *polygon spans*, which are rasterized one-by-one generating pixels for the scanline. Each new pixel is compared with the pixel stored in the corresponding z-buffer location. If the distance to the screen of the new pixel is less than the pixel stored in the z-buffer, the new pixel replaces the pixel in the z-buffer.

Since hidden-surface removal operations are order-independent, i.e., the result does not depend on the order triangles are processed, one z-buffer is sufficient to store the partial image. As the raster filter reads data buffers from its input stream, the contents of each buffer is rendered to the same z-buffer. When the

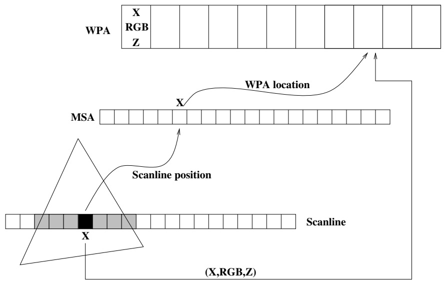


Figure 4: Rasterizing a polygon span in the active pixel rendering algorithm.

end-of-work marker is received from the input stream, the raster filter sends the contents of the entire z-buffer to the merge filter in fixed size buffers. The merge filter also allocates and initializes a z-buffer when it is instantiated. As data is received from the raster filter, the received z-buffer values are merged with the values stored in the local z-buffer. Results are merged by comparing each pixel to the local z-buffer, and choosing the one with the smallest distance to the screen. After all buffers are received and processed, the merge filter extracts color values from the z-buffer and generates an image to send to the client for display.

We can view the z-buffer rendering algorithm as having two phases: (1) a local rendering phase, in which input buffers are processed in the raster filter, and (2) a pixel merging phase, in which z-buffers are merged in the merge filter. The end-of-work marker behaves as a synchronization point between the two phases and among all copies of the raster filter, since all the copies wait until receiving this marker to start sending data to the merge filter. In the z-buffer rendering algorithm, the system memory is not efficiently utilized, because each copy of the raster filter allocates a full z-buffer. Moreover, z-buffer rendering may introduce large communication overhead in the pixel merging phase because pixel information for inactive pixel locations is also sent. Memory usage, and communication and synchronization overheads can be reduced by not storing or sending inactive pixels in each copy of the raster filter. This algorithm is referred to as *active pixel* rendering.

3.2.2 Active Pixel Rendering

This algorithm utilizes a modified scanline z-buffer scheme to store foremost pixels in consecutive memory locations efficiently. Essentially, active pixel uses a spare representation of the dense z-buffer. It also avoids allocation and initialization of a full z-buffer. Two arrays are used for hidden-surface removal. The first array is the *Winning Pixel Array* (WPA), which is used to store the foremost (winning) pixels. Each entry in this array contains the position on the screen, distance to the screen, and color (RGB) value of a foremost pixel. Since hidden-surface removal is carried out in scanline order, the pixels in the WPA are in scanline order and pixels in a scanline are stored in consecutive locations. Hence, only the x value of a pixel at (x,y) position on the screen is stored in the location field. In our implementation, fixed size buffers for the output stream of the raster filter are used for the WPA. The second array, the *Modified Scanline Array* (MSA) (of size M, where M is the x-resolution of the screen), is an integer array and serves as an index into the WPA for the scanline being processed.

As in the z-buffer rendering algorithm, the input buffer triangles are processed in scanline order. At the beginning of each scanline, the negative value of the scanline number is stored in the location field of the first available WPA entry. That entry is used as a marker to indicate the beginning of a new scanline. Then, the index of the next available position in the WPA is stored into an integer, named *start*. During the processing of a polygon span, when a pixel is generated for a position x in the current scanline, the location x in the MSA is used to access the WPA (see Figure 4); the value stored in the MSA entry (MSA[x]) is used as an index into the WPA. If MSA[x] has a value greater than *start*, it means that the WPA entry at location MSA[x] is updated by a span belonging to the current scanline. In that case, the generated pixel is used to update the entry in the WPA. Otherwise, the pixel generated for the current scanline is directly appended to the WPA, and the index of the the pixel's position in the WPA is stored in MSA[x]. This indexing scheme avoids re-initialization of the MSA at each scanline. The WPA is sent to the merge filter when full or when all triangles in the current input buffer are processed. In this way, processing of triangles in the raster filter is overlapped with merging operations in the merge filter. Also, unlike the z-buffer algorithm, there is no synchronization point between the local rendering phase and pixel merging phase, allowing pipelining of all filters including the merge.

4 Experimental Results

In this section, we experimentally demonstrate the implications of structuring the isosurface rendering application in different ways, and show how DataCutter provides appropriate support for efficient execution in a shared, heterogeneous environment.

Experiments were run on a dedicated cluster of SMP nodes for repeatability (8 2-processor Pentium II 450MHz nodes, 256MB memory, one 18GB disk, RedHat 6.2 Linux distribution; one 8-processor Pentium III 550MHz node, 4GB memory, RedHat 7.1beta Linux). The 2-processor nodes have a Gigabit Ethernet interconnect; the 8-processor node is connected to the 2-processor nodes over 100 Mbit Ethernet.

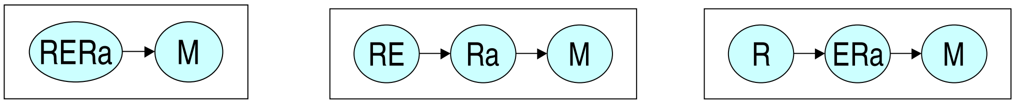
We experimented using a dataset generated by the parallel environmental simulator ParSSim [1], developed at the Texas Institute for Computational and Applied Mathematics (TICAM) at the University of Texas. The dataset is 1.5GB in size and contains the simulation output for fluid flow and the transport of four chemical species over 10 time steps on a rectilinear grid of $302 \times 125 \times 125$ points. The grid at each time step was partitioned into 1536 equal sub-volumes in three dimensions, and was distributed to 64 data files using a Hilbert curve-based declustering algorithm [7]. The data files were distributed to disks attached to the 2-processor nodes in round-robin order for each configuration. The number of read filters for a given experiment indicates the degree of dataset partitioning. For all experiments, we rendered a single iso-surface into a 2048x2048 RGB image. The timing values presented are the average of five repeated runs.

We first present a baseline experiment where each of the four filters are isolated and executed on a separate host in pipeline fashion. The first table shows the number of buffers and total volume sent between the pairs of filters. The active pixel version sends many more buffers, but they have a smaller total size. The second table shows the processing times of each of the base filters, the sum of these processing times, and the actual response time. Raster is by far the most expensive filter.

			Active Pixel	*Z-buffer*
	R→E	E→Ra	Ra→M	Ra→M
number	443	470	16	469
volume (MB)	38.6	11.8	32.0	28.5

	R		*E*		*Ra*		*M*		*Sum*	*RespTime*
z-buffer	0.68	5.3%	1.65	13.0%	9.43	74.5%	0.90	7.1%	12.66	11.22
active pixel	0.64	4.3%	1.64	11.2%	11.67	79.5%	0.73	5.0%	14.68	12.65

One of the degrees of freedom in a component-based implementation is to decide how to decompose the application processing structure into components. We have experimented with three different configurations of the iso-surface application filters described in Section 3, where multiple **R**ead, **E**xtract, and **Ra**ster filters are combined into a single filter (Figure 5). In all configurations, The **M**erge filter is always a separate filter, and only one copy of **M**erge is executed on one of the nodes in all experiments.



(a) **RERa–M** configuration (b) **RE–Ra–M** configuration (c) **R–ERa–M** configuration

Figure 5: Experimental configurations with different combinations of **R**ead, **E**xtract, and **Ra**ster as filters.

4.1 Background Load

In this experiment, we examine the performance of the three filter configurations and the DataCutter policies for buffer distribution among multiple transparent filter copies (see Section 2.2), when there is computational load imbalance among the nodes in the system due to other user processes. One copy of each filter (excluding **M**erge) was executed on each of eight 2-processor nodes. Background load was added to 4 of the nodes by executing a user level job that consumes CPU time, at the same priority as the filter code. The node running the **M**erge filter does not have a background job. Since we are using LinuxThreads, which execute threads as kernel processes, the actual filter thread slowdown depends on the number of application filters and the number of background jobs, plus one DataCutter system thread on each node.

		Active Pixel Rendering			*Z-buffer Rendering*		
Configuration	*# Bg Jobs*	*RR*	*WRR*	*DD*	*RR*	*WRR*	*DD*
RERa–M	1	3.13	3.16	3.16	11.45	11.49	11.54
RE–Ra–M	1	3.07	3.03	3.06	11.56	11.58	11.64
R–ERa–M	1	3.20	3.21	3.25	11.73	11.86	11.87
RERa–M	2	4.13	4.22	4.16	12.31	12.01	11.79
RE–Ra–M	2	3.95	3.72	3.71	12.57	12.77	12.10
R–ERa–M	2	4.07	4.08	3.87	13.12	13.52	12.31
RERa–M	4	7.10	6.89	6.71	13.39	13.77	13.82
RE–Ra–M	4	5.89	5.82	5.42	15.29	15.19	14.59
R–ERa–M	4	6.36	6.30	5.75	15.98	16.09	14.31
RERa–M	8	12.74	12.17	12.20	19.85	20.24	19.50
RE–Ra–M	8	10.63	9.90	9.58	22.81	22.98	20.43
R–ERa–M	8	11.16	11.01	9.65	22.87	23.63	20.35

Table 1: Comparison of execution time (seconds) for filter configurations with background jobs. 8 nodes total are used, with 7 nodes executing 1 copy of each of the filters except the merge filter (and background jobs on 4 of those nodes), and 1 node executes 1 copy of all filters including the merge filter. Load balancing policies are round robin (RR), weighted round robin (WRR) and demand driven (DD).

As shown in Table 1, increasing the number of background jobs on a subset of the 8 nodes increases the execution time as expected. For the round-robin policy (RR), the combined filter that contains **R**ead sends one buffer to every consuming filter. Weighted Round Robin (WRR) is expected to have the same effect as RR, since the number of copies per node of any given filter is one in this experiment. Since only half the nodes are actually experiencing the background load, there is load imbalance among the two sets of nodes. For the Demand Driven (DD) policy, when a consumer filter processes a data buffer received from a producer, it sends back a acknowledgment message (indicating that the buffer has been consumed) to the producer. The producer chooses the consumer filter with the least number of unacknowledged buffers to send a data buffer. As is seen from the table, the DD policy deals best with the load imbalance, by dynamically scheduling buffers among the filters. The main tradeoff with DD is the overhead of the acknowledgment messages for each buffer versus the benefits of hiding load imbalance. Our results show that in this machine configuration the overhead is relatively small, because the performance of DD is close to that of RR and WRR when background load is low. As the load imbalance among nodes increases, DD achieves better performance over RR and WRR, especially for the active pixel algorithm.

For the z-buffer algorithm, all three filter configurations achieve similar performance. Since full z-buffer arrays are combined in the merge filter, the communication overhead dominates the execution time and the merging of partial images becomes a bottleneck. For the active pixel algorithm, the performance of the **RE**–**Ra**–**M** and **R**–**ERa**–**M** filter configurations is better than the **ERa**–**M** configuration, especially using the DD policy. The **RERa**–**M** configuration never shows improvement using DD, because a single combined filter allows no demand-driven distribution of buffers among copies, and the output for each filter goes to the single **M**erge copy. In most cases, the **RE**–**Ra**–**M** configuration shows the best performance, since the raster filter is the most compute intensive filter among all the filters, and the volume of data transfer between **RE** and **Ra** is less than that of data transfer between **R** and **ERa** in the experiments. Thus, we limit the next experiment to the **RE**–**Ra**–**M** configuration.

4.2 Varying the Number of Nodes

This experiment compares the performance of the active pixel algorithm against that of the z-buffer algorithm when the number of nodes is varied. We also look at the performance implications of running two copies of the raster filter on each node and of colocating some of the copies of the raster filter with the merge filter. The execution times are shown in Table 2. In this experiment, no background jobs are run on the nodes, and a single **RE** filter was executed on each node.

		Active Pixel Rendering				*Z-buffer Rendering*			
Configuration	# **Ra**	*1 node*	*2 nodes*	*4 nodes*	*8 nodes*	*1 node*	*2 nodes*	*4 nodes*	*8 nodes*
RE–**Ra**–**M**	1 (0)	*n/a*	12.70	4.81	2.89	*n/a*	11.32	7.65	10.65
	1 (1)	12.18	7.32	4.17	3.00	10.74	7.92	7.89	11.48
RE–**Ra**–**M**	2 (0)	*n/a*	7.73	3.22	2.60	*n/a*	9.27	10.52	19.04
	2 (2)	8.16	5.70	3.88	3.24	8.57	8.85	11.71	20.78

Table 2: Comparison of execution time (seconds) while varying number of nodes. In the second column, the value in the parentheses is the number of raster filters colocated with the merge filter when there are more than 2 nodes. For the 1-node configuration, the raster and merge filters must be colocated.

As seen in Table 2, the active pixel algorithm performs better than the z-buffer algorithm as the number of nodes is increased because of the pipelining and overlapped processing of triangles and merge operations. Moreover, the z-buffer algorithm incurs much higher communication overhead than the active pixel algorithm, causing the merging of partial results to become bottleneck and flooding the interconnect between the nodes. This effect is seen from the table by the increase in the execution time of the z-buffer algorithm when the number of nodes is increased from 4 to 8.

Increasing the number of the raster filters improves performance for the active pixel algorithm. However, the increase in not linear. The amount of data processed by each copy decreases as the number of nodes is increased, while the total volume of data sent to the merge filter remains constant. **M**erge can be thought of as a quasi-sequential portion of the application; for the active pixel algorithm, merging operations are somewhat pipelined, but for the z-buffer algorithm there is a barrier before the merge. Therefore, as the number of nodes and the number of the raster filters increase, **M**erge becomes a bottleneck. This situation is more visible for the z-buffer algorithm; performance degrades when the number of the raster filters is increased from one to two on 8 nodes. Colocating some copies of the raster filter with **M**erge helps improve performance on smaller numbers of nodes by reducing the volume of communication. However, execution time may increase when multiple copies of the raster filter are colocated with **M**erge on larger number of nodes. All the filters on the same node are multiplexed onto the CPU by the OS. Thus, **M**erge does not get adequate CPU time on the node where it executes, resulting in increased execution time for the entire application.

4.3 Using a Heterogeneous Collection of Nodes

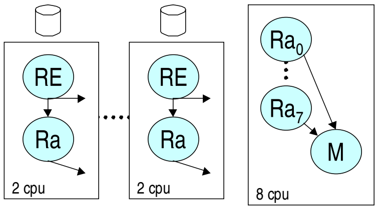


Figure 6: Heterogeneous collection of nodes experimental setup, varying number of 2-processor nodes.

In this experiment, we use the 8-processor machine as a compute node and vary the number of 2-processor nodes (Figure 6). We look at the performance of different filter configurations using the active pixel algorithm. **M**erge is executed on the 8-processor node along with 7 **ERa** or **Ra** filters. One copy of each filter except **M**erge is run on every 2-processor node for each configuration. The dataset is partitioned among disks on the 2-processor nodes. Note that while interprocessor communication is done over Gigabit Ethernet among the 2-processor nodes, all communication to the 8-processor node goes over slower 100Mbit Ethernet.

	1 data node			*2 data nodes*			*4 data nodes*			*8 data nodes*		
Configuration	*RR*	*WRR*	*DD*	*RR*	*WRR*	*DD*	*RR*	*WRR*	*DD*	*RR*	*WRR*	*DD*
RE–**Ra**–**M**	7.26	3.02	3.04	5.69	2.62	2.99	4.37	2.98	3.50	3.75	2.89	3.83
R–**ERa**–**M**	8.22	3.97	4.22	6.47	4.08	3.99	5.10	4.07	4.11	4.00	3.69	4.20

Table 3: Comparison of execution time (seconds) for filter configurations using the active pixel algorithm while varying the number of nodes storing the dataset. The 8-processor node is a compute node.

In all cases, the **RE**–**Ra**–**M** configuration performs better than **R**–**ERa**–**M** because of lower volume of communication. When the number of 2-processor nodes is high we don't obtain performance benefits from using the 8-processor node. The main reason is the slow network connection to and from the 8-processor node, which causes too much communication overhead. Nevertheless, use of the 8-processor node improves performance when data is partitioned on a small number of nodes. Decomposing the application into an appropriate set of filters allows us to move part of the computation to the 8-processor node, thus resulting in better parallelism. Our results also show that WRR achieves the best performance among all the DataCutter policies in this experiment. Note that there are no background jobs running on the nodes. Hence, DD should behave similarly to WRR, but acknowledgment messages from filters running on the 8-processor node incur high overhead in DD because of the slow network connection.

5 Conclusions

The component-based isosurface rendering allows a flexible implementation for achieving good performance in a heterogeneous environment and when the load on system resources change. Partitioning the application into filters and executing multiple copies of bottleneck filters results in parallelism and better utilization of resources. The active pixel algorithm is a better alternative to the z-buffer algorithm in the component-based implementation of isosurface rendering. It makes better use of system memory, and allows the rasterization and merging operations to be pipelined and overlapped.

A demand driven buffer distribution scheme among multiple filter copies achieves better performance than other policies when the bandwidth of the interconnect is reasonably high and the system load dynamically changes. However, extra communication required for acknowledgment messages introduces too much overhead when the network is slow. We plan to further investigate methods to reduce the communication overhead in DD and look at other dynamic strategies for buffer distribution.

Merging of partial images by a single filter is a performance bottleneck in the current implementation. As the number of copies of other filters or the number of nodes increases, this filter becomes a bottleneck. This is expected, and we are investigating other implementations for merging to alleviate this problem, such as a hierarchy of merging filters. The current approach replicates the image space across the raster filters. Alternatively, we could partition the image space into subregions among the raster filters, thus eliminating the merge filter. However, this will cause load imbalance among raster filters if the amount of data for each subregion is not the same, or if it varied over time. We also plan to investigate a hybrid strategy that combines image-partitioning and image-replication.

References

[1] T. Arbogast, S. Bryant, C. Dawson, and M. F. Wheeler. Parssim: The parallel subsurface simulator, single phase. *http://www.ticam.utexas.edu/∼arbogast/parssim*.

[2] C. L. Bajaj, V. Pascucci, D. Thompson, and X. Y. Zhang. Parallel accelerated isocontouring for out-of-core visualization. In *Proceedings of the 1999 IEEE Symposium on Parallel Visualization and Graphics*, pages 97–104, San Francisco, CA, USA, Oct 1999.

[3] 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.

[4] M. D. Beynon, T. Kurc, A. Sussman, and J. Saltz. Optimizing execution of component-based applications using group instances. In *Proceedings of CCGrid2001: IEEE International Symposium on Cluster Computing and the Grid*. IEEE Computer Society Press, May 2001. To appear.

[5] Y.-J. Chiang and C. Silva. External memory techniques for isosurface extraction in scientific visualization. In J. Abello and J. Vitter, editors, *External Memory Algorithms and Visualization*, volume 50, pages 247–277. DIMACS Book Series, American Mathematical Society, 1999.

[6] M. Cox and D. Ellsworth. Application-controlled demand paging for out-of-core visualization. In *Proceedings of the 8th IEEE Visualization'97 Conference*, 1997.

[7] C. Faloutsos and P. Bhagwat. Declustering using fractals. In *Proceedings of the 2nd International Conference on Parallel and Distributed Information Systems*, pages 18–25, Jan. 1993.

[8] Global Grid Forum. *http://www.gridforum.org*.

[9] T. Kurc, C. Aykanat, and B. Ozguc. Object-space parallel polygon rendering on hypercubes. *Computers & Graphics*, 22(4):487–503, 1998.

[10] W. Lorensen and H. Cline. Marching cubes: a high resolution 3d surface reconstruction algorithm. *Computer Graphics*, 21(4):163–169, 1987.

[11] S.-K. Ueng, K. Sikorski, and K.-L. Ma. Out-of-core streamline visualization on large unstructured meshes. *IEEE Transactions on Visualization and Computer Graphics*, 3(4):370–380, Dec. 1997.

[12] A. Watt. *Fundamentals of three-dimensional computer graphics*. Addison Wesley, 1989.