

A Parallel Implementation of 4-Dimensional Haralick Texture Analysis for Disk-resident Image Datasets *

Brent Woods[†], Bradley Clymer[†], Joel Saltz[‡], Tahsin Kurc[‡]

[†] Dept. of Electrical Engineering
The Ohio State University
Columbus, OH, 43210
{woods,clymer}@bmi.osu.edu

[‡] Dept. of Biomedical Informatics
The Ohio State University
Columbus, OH, 43210
{kurc,jsaltz}@bmi.osu.edu

Abstract

Texture analysis is one possible method to detect features in biomedical images. During texture analysis, texture related information is found by examining local variations in image brightness. 4-dimensional (4D) Haralick texture analysis is a method that extracts local variations along space and time dimensions and represents them as a collection of fourteen statistical parameters. However, the application of the 4D Haralick method on large time-dependent 2D and 3D image datasets is hindered by computation and memory requirements. This paper presents a parallel implementation of 4D Haralick texture analysis on PC clusters. We present a performance evaluation of our implementation on a cluster of PCs. Our results show that good performance can be achieved for this application via combined use of task- and data-parallelism.

1 Introduction

The quality and usefulness of medical imaging is constantly evolving, leading to better patient care and more reliance on advanced imaging techniques. For example, a current method of cancer research uses dynamic contrast enhanced magnetic resonance imaging (DCE-MRI) [25, 26, 35], which is also the main motivating application for this work, for detection and monitoring of tumors. During a DCE-MRI scan, the patient is injected with a contrast medium. A series of 3D MRI scans of a region of inter-

est, such as the breast, are taken at specific time intervals. Cancerous tumors by their nature tend to receive more of the blood containing the contrast agent. Once oxygen and nutrients have been consumed, the blood and contrast agent are removed as waste from the tumor. This process is monitored by MRI scans over many time steps. In addition, follow-up studies, which acquire multiple image datasets at different dates, can be conducted to monitor the progression and response to treatment of the tumor. Extraction and analysis of features from these images over multiple time steps can be used to detect tumors, by characterizing, for instance, contrast uptake and elimination in a region, and examine their progression over time.

In medical imaging, the diagnostic problem in the region of interest can often be associated with a variation in image brightness [24]. Texture analysis is one possible method to detect and examine such variations. During texture analysis, texture related information is found by examining local variations in image brightness. Haralick texture analysis is a form of statistical texture analysis that represents local variations as a collection of up to fourteen statistical parameters, such as contrast and entropy [19].

Using texture to analyze DCE-MRI datasets has shown great potential in tumor detection. Images that have been analyzed by radiologists can be used along with the results of texture analysis to train a neural network. Once trained, the neural network becomes a convenient tool for discovering cancerous tissue given the texture analysis results. The effectiveness of using 4D Haralick-based texture analysis for cancer detection will be discussed in a future paper.

As advances in imaging technologies allow a researcher to capture higher quality images and acquire more images in a shorter period of time, the amount of data that must be stored and processed increases as well. Obtaining additional data by acquiring images over many time steps provides a more complete view of the patient's physiology, but it can also create a quantity of data that may be impossible

*This research was supported in part by the National Science Foundation under Grants #ACI-9619020 (UC Subcontract #10152408), #EIA-0121177, #ACI-0203846, #ACI-0130437, #ANI-0330612, #ACI-9982087, Lawrence Livermore National Laboratory under Grant #B517095 (UC Subcontract #10184497), NIH NIBIB BISTI #P20EB000591, Ohio Board of Regents BRTTC #BRTT02-0003.

to process on a single workstation. With increasing resolution of medical imaging devices, a dataset, which consists of many time steps, may not fit in memory. For example, a digitizing microscope can scan pathology slides at 40x magnification, resulting in images of multiple gigabytes in size. Similarly, high-resolution MRI scanners are capable of acquiring 3D volumes of 1024×1024 -pixel images over many time steps. In addition, texture analysis is a computationally intensive process. These issues may be addressed using distributed computing. Current methods of analyzing DCE-MRI datasets can be tedious and time consuming for radiologists. These methods often involve cinematic viewing of the contrast agent flow, observation of a color-coded representation of the vascular permeability characteristics, and examination of the time versus intensity plots of individual pixels [25]. Automating the DCE-MRI data analysis process using distributed computing may also allow the radiologist to have the results of the DCE-MRI procedure before the patient exits the MRI facility. The ability to evaluate the patient on the same day as DCE-MRI procedure is a major motivation for using distributed computing.

In this paper, we develop a parallel implementation of 4D Haralick texture analysis for disk-resident datasets. Our approach involves combined use of task- and data-parallelism to leverage distributed computing power and storage space on PC clusters. We perform a performance evaluation of the implementation using a cluster of PCs.

2 Related Work

Parallel image processing and visualization is a widely studied area. In this section, we describe some of the previous work in parallel visualization and image analysis.

In this paper, we target efficient use of general purpose CPUs on PC clusters by use of task- and data-parallelism; we did not assume availability of graphics processing units (GPUs) on compute hosts and did not take advantage of GPUs in carrying out Haralick texture computations. There has been an increasing interest in applying programmable GPUs to speed up parts of image processing operations and general purpose computations [8, 9, 18, 12, 22, 29]. A future extension to our work could investigate how the Haralick-based texture computations could be mapped onto GPUs; in such an implementation, we anticipate that combined use of functional decomposition and data parallelism (the approach taken in this paper) will be an efficient approach as it can enable decomposition and placement of processing operations across multiple processing units (CPUs and GPUs).

We are not aware of any parallel implementations of 4D Haralick texture analysis. Fleig et.al. [17, 27] implemented a parallel haralick texture analysis program that worked on 2D slices of a 3D volume. Each slice was treated sepa-

ately and processed by a single function in memory. Unlike [17, 27], the implementation described in this paper handles disk-resident 4D datasets and can carry out Haralick texture computations in 4-dimensions.

Chiang and Silva [11] propose methods for iso-surface extraction for datasets that cannot fit in memory. They introduce several techniques and indexing structures to efficiently search for cells through which the iso-surface passes, and to reduce I/O costs and disk space requirements. Cox and Ellsworth [14] show that relying on operating system virtual memory results in poor performance. They propose a paged system and algorithms for memory management and paging for out-of-core visualization. Their results show that *application controlled paging* can substantially improve application performance.

Ueng et. al [39] present algorithms for streamline visualization of large unstructured tetrahedral meshes. They employ an octree to partition an out-of-core dataset into smaller sets, and describe optimized techniques for scheduling operations and managing memory for streamline visualization. Arge et. al. [2] present efficient external memory algorithms for applications that make use of grid-based terrains in Geographic Information Systems. Bajaj et. al. [4] present a parallel algorithm for out-of-core isocontouring of scientific datasets for visualization. In [28], several image-space partitioning algorithms are evaluated on parallel systems for visualization of unstructured grids.

Manolakos and Funk [32] describe a Java-based tool for rapid prototyping of image processing operations. This tool uses a component-based framework, called JavaPorts, and implements a master-worker mechanism. Oberhuber [34] presents an infrastructure for remote execution of image processing applications using SGI ImageVision library, which is developed to run on SGI machines, and Net-Solve [10].

SCIRun [37] is a problem solving environment that enables implementation and execution of visualization and image processing applications from connected modules. Dv [1] is a framework for developing applications for distributed visualization of large scientific datasets. It is based on the notion of active frames, which are application level mobile objects. An active frame contains application data, called frame data, and a frame program that processes the data. Active frames are executed by active frame servers running on the machines at the client and remote sites. Hastings et. al. [21] present a toolkit for implementation and execution of image analysis applications as a network of ITK [23] and VTK [36] functions in a distributed environment.

3 Haralick Texture Analysis

The goal of texture analysis is to quantify the dependencies between neighboring pixels and patterns of variation in image brightness within a region of interest [24, 38, 13]. In texture analysis, useful information can be found by examining local variations in image brightness. Haralick texture analysis [19] is a form of a statistical texture analysis that utilizes co-occurrence matrices. It relies on the joint statistics of neighboring pixels or voxels in the dataset rather than a structure definition.

The basis behind the method is the study of second order statistics relating neighbouring pixels at various spacings and directions. A second-order joint conditional probability density function is computed, given a specific distance between pixels and a specific direction. This second-order joint conditional probability density function is referred to as a co-occurrence matrix. A co-occurrence matrix can also be thought of as a joint histogram of two random variables. The random variables are the gray level of one pixel (g_1) and the gray level of its neighboring pixel (g_2), where the neighborhood between two pixels is defined by a user-specified distance and direction. This represents a joint probability distribution function (p.d.f.), which gives the probability of neighboring pixels changing from the intensity g_1 to the intensity g_2 . The co-occurrence matrix measures the number of occurrences in which two neighboring pixels, one with gray level g_1 and the other with gray level g_2 , occur a distance d away and along a certain direction. A short description of co-occurrence matrix computation is given in the Appendix.

There are three notable properties of the co-occurrence matrix. First, the relationships between neighboring pixels occur in both the forward and backward direction. Consider a 2D case; there are 8 total directions: 0, 45, 90, 135, 180, 225, 270, and 315 degrees. However, opposite angles yield the same co-occurrence matrix. Therefore only four unique directions exist (see Figure 12 in the Appendix for the eight possible directions and the four unique vectors). Along the same idea, there is symmetry in the co-occurrence matrix because the gray level relationships between the pixels occur in both the forward and backward directions. The values along the diagonal of the co-occurrence matrix are unique; however, the values above the diagonal match the values below the diagonal in the co-occurrence matrix. The co-occurrence matrix is also a square matrix and is always $N_g \times N_g$ in size, where N_g is the total number of gray levels possible. Therefore, the size of the co-occurrence matrix is fixed by the total number of gray levels and is independent of distance and direction values.

Once a co-occurrence matrix is computed, statistical parameters can be calculated from the matrix. The fourteen textural features described by Haralick [19] provide a wide

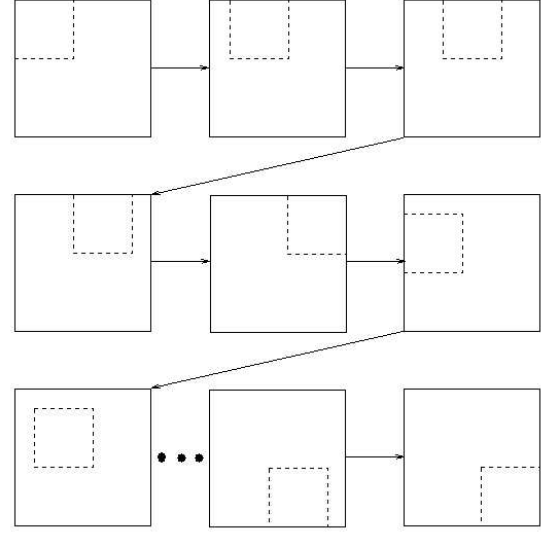


Figure 1. Raster Scanning: A ROI window scans through the image.

range of parameters that can be used in medical imaging texture analysis.

In medical images, many localized texture changes denoting tumors, capillaries, and differing tissues may exist. Thus, it is often necessary to apply a series of texture calculations with each calculation performed on a localized region of interest. This process is known as raster scanning. Raster scanning begins with a fixed, specified region of interest (ROI), where the size of the ROI depends on the size of important structures within the image. Raster scanning begins at the first pixel in the image set. Figure 1 illustrates raster scanning for 2D case. The region within the ROI is used to generate a co-occurrence matrix. One or more of the Haralick parameters is then calculated and sent to a storage buffer. The ROI window is then shifted to an adjacent voxel. Again, a co-occurrence matrix is generated based on the region around that point. Haralick parameters are calculated and sent to a storage buffer. This scanning window process continues for all points in which the ROI occurs within the boundary of the image. The series of output parameters can be used in computer aided diagnosis, stored to disk, or used to construct a graphical view of the results. A pseudo-code summarizing the 4D Haralick texture analysis algorithm is given in Figure 2.

4 Parallel 4D Haralick Texture Analysis

The 4D Haralick texture analysis application can be modeled as four major stages. The first stages is to read in the 4D raw image dataset from the storage system and pass it to texture analysis operations. The second stage is

```

 $R \leftarrow (x_l, y_l, z_l, t_l)$  – ROI lengths in each dimension.
 $F \leftarrow \{H_i, \dots, H_j\}$  – set of selected Haralick functions.

foreach  $x$  in  $(0, \dots, L_x - x_l)$  do
  foreach  $y$  in  $(0, \dots, L_y - y_l)$  do
    foreach  $z$  in  $(0, \dots, L_z - z_l)$  do
      foreach  $t$  in  $(0, \dots, L_t - t_l)$  do
         $ROI \leftarrow (x, y, z, t) \times (x + x_l - 1, y + y_l - 1, z + z_l - 1, t + t_l - 1)$ 
         $M_c \leftarrow$  Compute co-occurrence matrix for  $ROI$ .
        foreach  $f$  in  $F$  do
          Compute haralick parameter  $f$  using  $M_c$ .

```

Figure 2. Sequential 4D Haralick texture analysis algorithm. The algorithm iterates over all the pixels/voxels in each dimension, creating local regions of interests (ROIs). Note that the entire ROI must be contained within the dataset. For each ROI, a co-occurrence matrix is computed. Using the co-occurrence matrix, the selected subset of Haralick parameters is calculated.

to compute the co-occurrence matrices. The calculation of Haralick texture parameters from the 4D data is the third stage in the processing structure. The resulting output is a 4D dataset for each Haralick parameter computed. The final stage is to output the 4D Haralick texture analysis results in a user specified format. Based on this modeling of Haralick texture analysis computations, we developed a task- and data-parallel implementation. Data parallelism is achieved by distributing data across the nodes in the system for both storage and computing purposes. The task parallelism is obtained by implementation and execution of the four major stages as separate tasks using a component-based framework [6, 5, 7]. In this section, we first briefly present the underlying runtime framework used in our implementation. We then describe how data is distributed across nodes for storage, individual components implementing different stages of the texture analysis algorithm, and optimizations for data retrieval and processing.

4.1 Runtime Middleware

In this project, distributed computing is accomplished through a middleware framework, called DataCutter, designed to process large datasets [6, 5, 7]. DataCutter is based on a filter-stream programming model that represents operations of a data-intensive application as a set of filters [7]. Data are exchanged between filters using streams, which are unidirectional pipes. Streams deliver data from producer filters to consumer filters in user-defined data chunks (data buffers). To achieve distributed computing, operational tasks are divided among a series of filters. Each filter can be executed on a separate processor/machine in the environment, or filters can be co-located. When filters are executed on separate processors, data exchange is done using TCP/IP sockets. When filters are co-located, the

runtime system transfers a data buffer from a producer filter to a consumer filter by copying the pointer to the data buffer. Consumer and producer filters can run concurrently and process data chunks in a pipelined fashion.

Filters may be replicated and placed on different nodes. Each replicated filter can process data independent of the other identical filters. Data parallelism can be made possible by distributing data buffers among replicated filters on-the-fly. Partitioning data into data chunks can help to achieve load balance and reduce the memory requirements of the nodes. Either explicit or transparent copies of a filter can be instantiated and executed. If the copies of a filter are *transparent*, the DataCutter scheduler controls which of the identical filter copies receives a data buffer. The DataCutter scheduler can schedule data buffers to transparent filter copies in either round robin or demand driven sequences. In a *round robin* distribution, the scheduler assigns data to each transparent filter in turn. Thus, each transparent filter receives roughly the same amount of data to process. In a *demand driven* scheduling of data buffers, the DataCutter scheduler assigns the distribution based on the buffer consumption rate of the transparent filter copies. The goal of the demand driven approach is to send data to the transparent filter copies that can process them the fastest. *Explicit* filters are used to give the user control over which consumer filter receives which data chunk from a producer filter. Explicit filters are useful in situations where assignment of data chunks to filter copies in a user-defined way is required or can improve performance.

4.2 Data Distribution Among Storage Nodes

In MRI studies, a 4D image dataset can be composed of a series of 3D volumes. A 3D volume is made up of a number of image slices, each of which is usually small

to medium size (i.e., 256^2 to 1024^2 pixels). However, the 3D volume can consist of many image slices (e.g., 1024 slices) and image acquisition can be performed over a large number of time steps. Thus, it is possible that a 4D dataset may not fit on a single storage node and need to be distributed across multiple nodes. In addition, distributing the input dataset across multiple storage nodes has the advantage that data retrieval can be parallelized. A number of techniques have been developed for partitioning and declustering multi-dimensional datasets [15, 16, 31, 33]. Obviously, the effectiveness of a particular distribution depends on how well it matches the common data access and query patterns of the target application class. In MRI studies, common analysis queries specify entire 3D volumes over a range of time steps. In our current implementation, 2D image slices that make a 3D volume at a time step are distributed across storage nodes in round robin fashion. Each 2D image is assigned to a single storage node and stored on disk in a separate file. A simple index file is created on each storage node for the images assigned to that storage node. In this index file, each image file is associated with a $(sliceid, timestep)$ tuple. *timestep* denotes the time step the image slice belongs to and *sliceid* is the number of the image slice within the 3D volume.

4.3 Filters

We have developed three sets of filters to carry out the four stages of Haralick texture analysis operations (see Figure 3). These filter sets can be connected to form an end-to-end Haralick texture analysis chain. For a more detailed explanation of the filter sets developed and their implementation, please refer to the thesis by Woods [40]. The filter scheme also provides support for incremental development. For instance the filter developed to read in raw DCE-MRI data may be easily replaced by a filter which reads DICOM format images.

In our current implementation we do not provide a graphical user interface for composition of various filters into filter groups, since our focus has been on evaluating different strategies for parallel and distributed execution. The filters are implemented in C++ using the base classes provided by the DataCutter framework and the filter network is expressed as an XML document [21]. An extension to our current implementation would be to investigate the use of graphical tools, such as SCIRun [37] and AVS [3], which provide interfaces to compose applications from individual modules, and of higher level languages [8, 20] as front-end for creating and composing filters and filter networks.

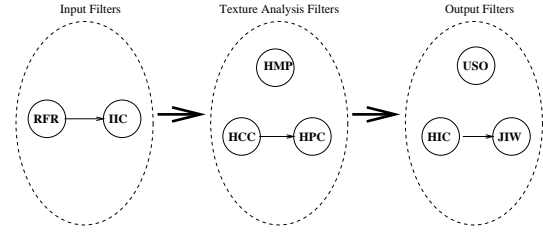


Figure 3. Three filter sets to carry out the main tasks in a Haralick texture analysis application.

4.3.1 Input Filters

RAWFileReader (RFR)

The purpose of the RFR filter is to read raw image data from disk and send them to other filters for processing. Multiple RFR filters can be executed if the image dataset is distributed across several storage nodes. In this case, a RFR filter is placed on each node containing image data. Each RFR filter extracts the local data needed to build a ROI and sends that data to the input stitch filter.

InputImageConstructor (IIC) (Input Stitch)

In order to compute a co-occurrence matrix, the complete Region-of-Interest (ROI) data are needed. If the 4D image dataset is distributed across multiple storage nodes, then a copy of the RAWFileReader filter will retrieve and send only the local data portions to other filters. The InputImageConstructor (IIC) filter reconstructs full ROIs and distributes them to the texture analysis filters. The inputs to the IIC filter are portions of the image data from the output of different RFR filters. The IIC filter places the input MRI portions into temporary buffers. After all data elements needed to build a complete ROI are received, the ROI is put into a send buffer. When the send buffer is full, the buffer is sent to the texture analysis filters.

4.3.2 Texture Analysis Filters

In our implementation, the Haralick texture analysis algorithm can be carried out in a distributed environment in various ways. The Haralick texture analysis operations for computing co-occurrence matrices and Haralick parameters can be contained in a single filter or task-distributed among two pipelined filters. Dividing the operations among two filters creates another level of task parallelism, but also introduces communication overhead between the filters that perform the operations.

HaralickMatrixProducer (HMP)

The HMP filter carries out the entire Haralick texture analysis processing. The filter receives a buffer of ROI image data from the IIC filter. For each of the ROIs in the input buffer, the co-occurrence matrix is calculated based on the image data within the ROI. The co-occurrence matrix is then used to generate any Haralick parameters that have been chosen by the user.

HaralickCoMatrixCalculator (HCC)

The HCC filter is responsible for calculating just the co-occurrence matrix from the input data. For each ROI in the input buffer, a co-occurrence matrix is calculated. The co-occurrence information is stored in an output buffer. When the output buffer becomes full or the end of an input data message is received, the data in the output buffer is passed to the HaralickParameterCalculator filter.

HaralickParameterCalculator (HPC)

The HPC filter is responsible for calculating the Haralick parameters from the co-occurrence matrices received from a HCC filter. All user-selected Haralick parameters are calculated for each matrix. Each parameter is stored in its own output buffer. When the output buffers are full or when the end of input data message is encountered, the data elements stored within the output buffers are sent to an output filter.

4.3.3 Output Filters

The user may choose to send the output portions received from texture analysis filters directly to disk. Once on disk, the data may be postprocessed for purposes of computer aided diagnosis. The user may also choose to store the Haralick parameter results in a visual way. To accomplish this, Haralick parameter output portions sent from the texture analysis filters are received at an output stitch filter. This filter reconstructs the parameter output portions into a series of 4D datasets. Each 4D dataset is the output for a single Haralick parameter. Once reconstructed, the 4D output datasets can be written to disk as a series of jpeg images.

UnstitchedOutput (USO)

The USO filter is responsible for writing the Haralick parameter information out to disk. The input to this filter is a stream of data elements for a Haralick parameter. The filter then writes the parameter data out to disk. Each input stream is assigned a unique file name. A file is opened, and the parameter values along with corresponding positional information are stored to the file. Postprocessing

applications can then use the data stored in these files for further computations.

HaralickImageConstructor (HIC)

The HIC filter is used to build the Haralick parameter information into images. This filter receives input streams consisting of Haralick parameter information. Each input stream contains a subset of the total output elements for a single Haralick parameter. This output stitch uses positional information stored in the input stream to place the parameter elements into their appropriate positions in the parameter image data structure. Once all data elements for a Haralick parameter have been correctly placed, a complete 4D dataset consisting of all elements for a parameter has been built. Once the output dataset is completely assembled, it is passed to the next filter for further processing.

JPGImageWriter (JIW)

The JIW filter receives a stream of data containing elements for a Haralick parameter that has been assembled by position. The input stream also contains the minimum and maximum values for the Haralick parameter elements. Using the minimum and maximum values, the data can be normalized. Therefore each value is assigned a value between zero and one. A zero results in a black pixel, and a one results in a white pixel. Any intermediate elements are assigned a scaled gray value. The filter then converts the 4D data into a series of 2D images that are stored in jpeg format.

Transparent filter copies of the RFR, HMP, HCC, HPC, and USO filters can be instantiated and executed in the environment. Figures 4 and 5 show two possible instantiations, referred to here as *the split HCC and HPC filter implementation* and *the HMP filter implementation*, of the 4D Haralick texture analysis application.

4.4 Data Retrieval

As stated earlier, a complete 4D ROI data is necessary to build one co-occurrence matrix. Fig. 6(a) illustrates how a 2D image can be accessed by ROIs; thus, each data packet sent to the texture analysis filters contains the ROI needed to build a co-occurrence matrix. In Fig. 6(a), ROI_x and ROI_y correspond to the dimension lengths of the ROI, which are supplied by the user. Also in Fig. 6(a) P1, P2, and P3 are the data chunks sent to the texture analysis filters. Note that most of the chunks contain overlapped data. If the input data is retrieved by ROIs, the data elements in the overlapped regions must be retrieved and sent to the texture analysis filters multiple times. Therefore, data retrieval in

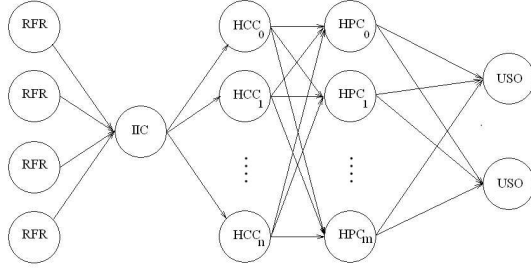


Figure 4. An example instantiation of the split HCC and HPC filter implementation. The input data is distributed among four storage nodes and ROIs are reconstructed using the IIC filter. Texture analysis is performed using n transparent copies of the HCC filter and m transparent copies of the HPC filter thereby splitting the texture analysis operations among two filters. The output Haralick parameters are stored to disk.

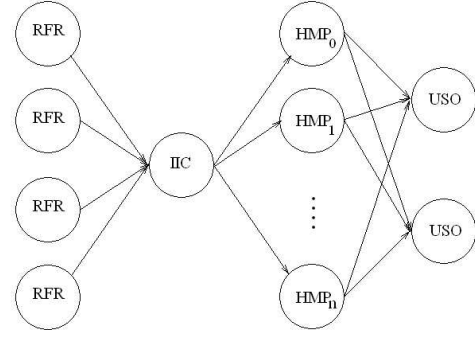


Figure 5. An example instantiation of the HMP filter implementation. The input data is distributed among four file systems and ROIs are reconstructed using the IIC filter. Texture analysis is performed using n transparent copies of the HMP filter, which combines all texture analysis operations into one filter. The output Haralick parameters are stored to disk.

terms of ROIs creates the largest volume of communication between the input filters and the texture analysis filters. In order to reduce the amount of data read from disk and communicated between RFR and IIC filters as well as IIC and texture analysis filters, the data are retrieved in 4D chunks, each of which contains a subset of ROIs. In Figure 6(b) a 2D image is partitioned into four data chunks each with the user specified dimensions $chunkx \times chunky$. The amount of overlap between two chunks in the x -direction depends on the ROI x -dimension length according to Eq. 1, and the amount of overlap between two adjacent chunks in the y -direction depends on the ROI y -dimension length according to Eq. 2.

$$overlap_x = chunkx - ROI_x + 1; \quad (1)$$

$$overlap_y = chunky - ROI_y + 1; \quad (2)$$

The current implementation has two types of chunks; an *RFR-to-IIC chunk* for data retrieved from disk and sent to the IIC filter and a *IIC-to-TEXTURE chunk* for communication between the IIC and haralick texture analysis filters. The input image data is stored as a set of image slices on disk. A RFR filter reads a 2D subsection of each image slice and puts it into a buffer, which corresponds to the I/O chunk. When the buffer is full, the RFR filter sends the buffer to the IIC filter. When the IIC filter receives buffers from RFR filters, it copies and reorganizes the contents of the buffers in a set of buffers, each of which is a 4D array and corresponds to a separate IIC-to-TEXTURE chunk. When a buffer is full, it is sent to one of the copies of the texture analysis

filters (i.e., HMP or HCC filters). An HMP or HCC filter then performs a raster scan of the chunk, received from the IIC filter, for ROIs and computes co-occurrence matrices. Having two types of chunks allow better optimization of execution time for different types of overheads. For example, a larger chunk size can be chosen for RFR-to-IIC chunk so that the number of disk seek operations for data retrieval can be reduced. With a smaller size for the IIC-to-TEXTURE chunk, pipelining between the IIC and texture analysis filters can be increased.

4.4.1 Full vs Sparse Matrix Representation

The co-occurrence matrix is a $N_g \times N_g$ matrix that relates the intensities of neighboring pixels along a certain direction. The number of gray levels (N_g) may be relatively large, such as 65536 (16-bit) intensity levels, or relatively small, such as requantized 32 (5-bit) levels. Our experiments have shown that in some cases, matrices generated using a typical $5 \times 5 \times 5 \times 5$ ROI and requantized 32 levels can have on average as little as 10.7 non-zero entries per matrix (out of $32 \times 32 = 1024$ entries, about 1% of the matrix). We note that this average takes into account matrix symmetry, and the symmetric entries are only stored once. Knowing that many of the co-occurrence matrices are relatively sparse leads to the following matrix storage schemes and optimizations.

The most obvious method of representing a co-occurrence matrix in memory is a 2D array of $N_g \times N_g$ elements. In this work, we refer to such a representation as

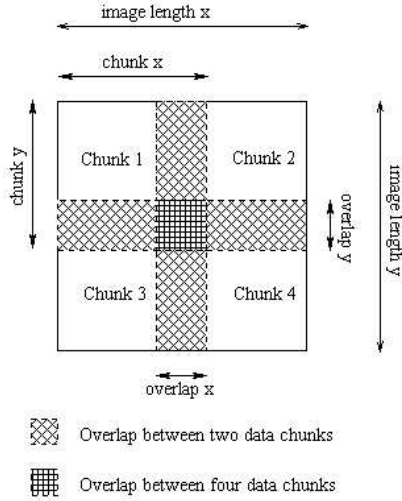
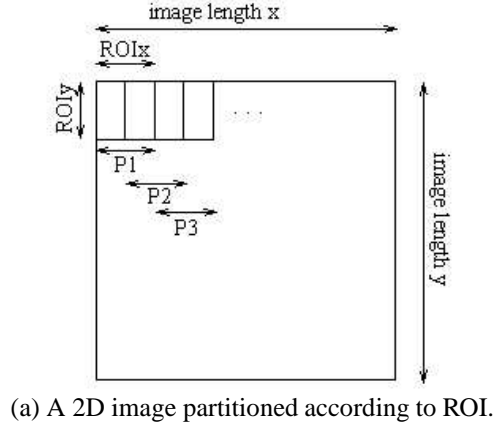


Figure 6. Two data retrieval strategies: retrieving ROIs, retrieving chunks.

a full matrix storage representation. Without optimization, all Haralick parameter calculations treat each element in the matrix the same. Therefore, zero entries in the matrix are added to running sums along with non zero entries. However, before adding an entry from the co-occurrence matrix, the entry can be tested to see if it is zero. If the entry is zero valued, then the entry is simply skipped. By first checking for zero values, we are able to reduce the time needed to process relatively sparse matrices. In fact, this optimization allowed us to process a typical MRI dataset in one-fourth the time.

The co-occurrence matrix may also be stored in a sparse matrix storage representation. Only the non zero and non duplicated (due to symmetry) entries are stored along with positional information in memory. The positional information is needed to map each non zero, non duplicated entry to its position in the co-occurrence matrix. If a matrix is in

the sparse form, then the Haralick parameter calculations do not have to check for non zero entries. Therefore, the matrix can be processed directly from the sparse form, and no conversion back to a co-occurrence array is needed. In addition, the sparse matrix representation can greatly reduce the data traffic leaving the HCC filter, if the texture analysis operations are split between the HCC and HPC filters. If the matrices are stored in sparse form, then they are also transmitted via the network in the sparse form.

5 Experimental Results

5.1 Experimental Setup

In the experiments presented in this paper, we used a dataset obtained from a DCE-MRI study. This dataset consists of 32 time steps. Each time step is made up of 32 images of 256×256 pixels each. Each pixel is 2 bytes in size¹. The region of interest (ROI) was set according to the dimension lengths $5 \times 5 \times 5 \times 5$. This ROI size was chosen because previous studies on the analysis of 2D images showed that such a ROI would be typical for an MRI application [24]. The number of gray levels, N_g , used to requantize the DCE-MRI dataset was set to 32, because in most cases values greater than 32 do not significantly improve the texture analysis results [24, 30].

Since each image slice in the input dataset is relatively small, the RFR-to-IIC chunk dimension lengths used in the experiments were set to $256 \times 256 \times 6 \times 6$. In this way, a RFR filter can read one image slice without any disk seek operations required to retrieve smaller image regions. The IIC-to-TEXTURE chunk dimension lengths used in data partitioning for distribution to texture analysis filters were set to $67 \times 67 \times 6 \times 6$ for all tests. When we conducted tests using smaller chunks, the overlap between partitions created a volume of communication that was too great. As a result, the program execution time was unacceptably large. Larger chunk sizes also produced poor results because the large data portions could not be distributed to the texture analysis filters fast enough, which left some texture analysis filters in an idle state. Therefore, we chose a chunk size that had a tolerable amount of overlap as a result of partitioning and also produced a balanced data distribution among the texture analysis filters. The HCC filters were configured to send out a packet of co-occurrence matrices whenever $\frac{1}{64}$ of a $67 \times 67 \times 6 \times 6$ chunk had been processed. Another possible packet size would be the entire chunk. However, for

¹Note that this sample dataset is small enough to fit in memory of a processor. Hence, as an optimization the dataset can be replicated on all of the nodes and read into memory as a whole in order to eliminate the need for the IIC filter. However, for large datasets it may not be possible to apply this optimization. Hence, in our experiments we assume that the sample dataset is not replicated and too big to fit in memory.

our configuration these settings result in good pipelining of data across different stages of the filter group, but do not cause excessive communication latencies.

For all tests, the following Haralick parameters were calculated: Angular Second Moment, Correlation, Sum of Squares, and Inverse Difference Moment [19] (see Appendix). We chose these parameters since they are four of the most computation-expensive parameters that can be produced from a co-occurrence matrix. We chose not to include all Haralick parameters because a typical DCE-MRI study would likely not need all parameters in order to generate a diagnosis.

5.2 Homogeneous Cluster Experiments

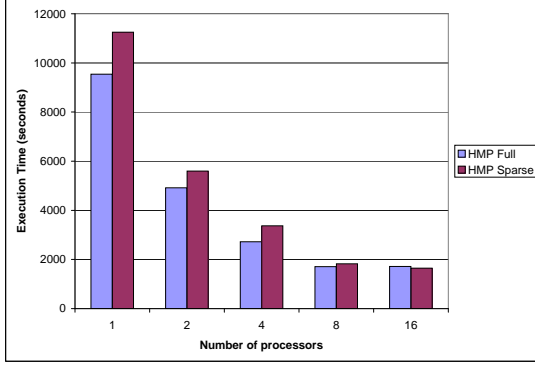
For the experiments detailed in this section, a homogeneous PC cluster was used. The cluster, referred to here as *PIII*, contains 24 nodes, each with a Pentium III processor and 512 MB of memory. All nodes are connected via a FastEthernet Switch capable of transmitting data at a rate of 100 Mbits per second.

In the first set of experiments, we investigate the impact of using full matrix representation vs sparse matrix representation and the performance of the *split HCC and HPC filter implementation* and the *HMP filter implementation* (see Figures 4 and 5). In the experiments, the input dataset was distributed across 4 I/O nodes. One of the nodes in the system was used to run the IIC filter. One USO filter was used for output. The remaining nodes were used to run the HMP filters or the HCC and HPC filters. Figure 7(a) shows the execution time when the number of nodes for HMP filters is varied from 1 to 16. In each configuration, one transparent copy of the HMP filter was placed on one node. As is seen from the figure, the implementation using sparse matrix representation performs worse than the implementation using full matrix representation. When HMP filters are used, the co-occurrence matrix computation and Haralick parameter calculation are done in the same filter, and there is no communication overhead between the two operations. Thus, the overhead introduced due to storing and accessing co-occurrence matrix in sparse representation degrades the performance. On the other hand, using sparse matrix representation achieves better performance in the split HCC and HPC filter case, as seen in Figure 7(b). This is mainly because of the fact that with sparse representation the communication overhead is reduced significantly. In this experiment, multiple transparent copies of HCC and HPC filters are created, but only one filter is executed on one node – we should note that for the one-node configuration, both HCC and HPC filter copies are executed on the same node. The number of copies for HCC and HPC filters was determined based on their relative processing times. We observed that the HCC filter was about 4 to 5 times more expensive than

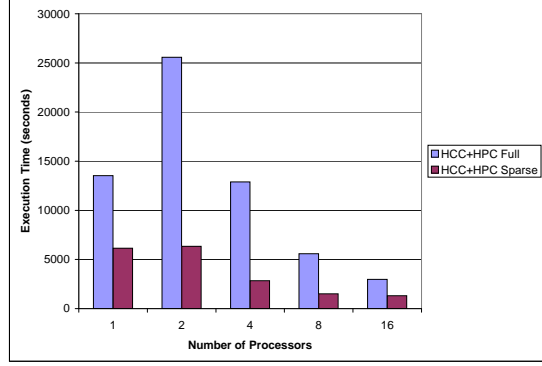
the HPC filter on average. Hence, the number of nodes in a given setup was partitioned so that a 4-to-1 ratio was maintained between HCC and HPC filters, when possible. For example, for the 16-node configuration, 13 HCC and 3 HPC filters were executed in the system.

The split HCC and HPC filter implementation provides flexibility in that HCC and HPC filters can be executed on separate nodes or run on the same node. The next set of experiments examines the performance impact of executing copies of HCC and HPC on the same node. When HCC and HPC filters are placed on the same node, the communication overhead will be reduced since buffers from the HCC filter that are delivered to the local copy of the HPC filter will incur no communication overhead (buffer exchange between two co-located filters is done via simple pointer copy operation). In addition, more copies of HCC and HPC filters can be executed in the system. However, since a node in the cluster used in these experiments has a single processor, the CPU has to multiplex between the two filters and its power has to be shared. In Figure 8, *No Overlap* denotes the case in which no two filters are co-located, whereas copies of HCC and HPC filters are executed on the same node in the case of *Overlap*. In the experiments, the HMP filter implementation used the full matrix representation and the split HCC and HPC filter implementation employed the sparse matrix representation for co-occurrence matrices. As is seen in the figure, *Overlap* achieves better performance compared to the HMP filter implementation and the *No Overlap* HCC+HPC implementation. Although the processing power is shared between two filters, the reduction in communication overhead and more copies of filters result in better performance. We also observe that in the one-node case, the split HCC and HPC filter implementation performs better than the HMP filter implementation. This result can be attributed to better pipelining that can be achieved by the split implementation; when HCC or HPC filter is waiting for send and receive operations to complete, the other filter can be doing computation.

Figure 9 shows the processing time of each filter (i.e., RFR, IIC, HCC, HPC, and USO) for the split HCC and HPC filter implementation. The read (RFR) and write (USO) overheads are negligible compared to the time taken by other filters. We observe that the execution time of the HCC and HPC filters decrease as more nodes are added. However, the IIC filter becomes a bottleneck filter in the 16-node configuration and adversely affects the scalability of the application to larger number of nodes. In order to alleviate this bottleneck, multiple *explicit* copies of the IIC filter should be instantiated. While transparent copies of the RFR, HCC, and HPC filters can be executed to take advantage of demand driven buffer scheduling, explicit copies of the IIC filter need to be created. This is because pieces of the same RFR-to-IIC chunk retrieved by multiple RFR filters must



(a)



(b)

Figure 7. The performance impact of using full matrix representation vs sparse matrix representation. (a) the HMP filter implementation. (b) The split HCC and HPC filter implementation.

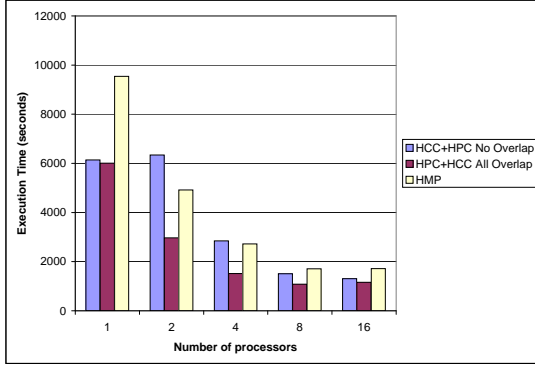


Figure 8. The performance impact of co-locating HCC and HPC filters vs running them on separate processors.

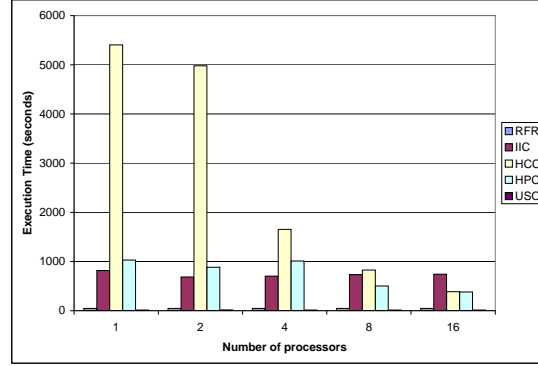


Figure 9. The processing time of each filter in the split HCC and HPC filter implementation. In this experiment, HCC and HPC filters are executed on separate nodes.

be assembled together to form complete IIC-to-TEXTURE chunks and ROIs. We examined round robin distribution of RFR-to-IIC chunks across multiple copies of the IIC filter. Our results showed that as the number of IIC filters is increased, the processing time of each IIC filter decreases almost linearly, as expected.

5.3 Heterogeneous Environment Experiments

In this set of experiments, we investigated execution of the parallel implementation in a heterogeneous environment. In addition to the PIII cluster used in the homogeneous experiments, two additional clusters were made available. The first additional cluster, referred to here as *XEON*, contains five nodes. Each node of the *XEON* cluster has dual Xeon 2.4GHz processors and 2GB of memory. The nodes on the *XEON* cluster are connected by a Gigabit Switch. The second additional cluster, referred to

here as *OPTERON*, contains six nodes. Each node of the *OPTERON* cluster has dual Opteron 1.4GHz processors and 8GB of memory. The nodes on the *OPTERON* cluster are also connected by a Gigabit Switch. PIII is connected to *XEON* and *OPTERON* through a shared 100 Mbit/s network. *XEON* and *OPTERON* are connected to each other using a Gigabit network.

Two experiments were performed to test the implementations in a heterogeneous environment. The first experiment provides a comparison of the HMP filter implementation and the split HCC and HPC filter implementation using the PIII and *XEON* clusters. In this experiment, 4 RFR filters, 4 IIC filters, and 2 USO filters were executed on the PIII cluster. The texture analysis filters were placed across the two clusters on a total of 18 nodes (13 nodes from the PIII cluster and 5 from the *XEON* cluster). For the HMP filter implementation, one transparent copy of HMP filter was in-

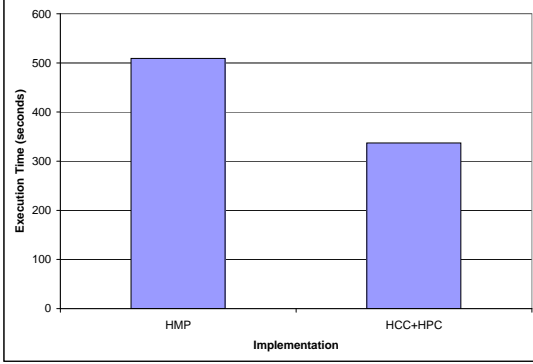


Figure 10. Performance comparison of the HMP filter implementation and the split HCC and HPC filter implementation in a heterogeneous environment.

stantiated on each processor. Since the XEON cluster has 10 processors (on 5 nodes), the total number of HMP filters was 23. For the split HCC and HPC filter implementation, one copy of HCC and one copy of HPC were co-located on each node, which resulted in 18 copies of HCC and 18 copies of HPC filters. While the HMP filter implementation aims to achieve good performance by spreading data across more HMP filters, the split HPC and HCC implementation targets a more balanced use of task- and data-parallelism by splitting the co-occurrence matrix computation and Haralick parameter calculation operations and creating multiple copies of individual filters. However, fewer copies of each filter are created in the split HCC and HPC filter implementation. As is seen in Figure 10, the split implementation achieves better performance. First, although 10 copies of HMP filter can be created on the XEON cluster, more data has to flow from the PIII cluster to this cluster across a relatively slow network to make optimal use of these copies. On the other hand, the split HCC and HPC filter implementation can take advantage of demand driven scheduling of co-occurrence matrix buffers across the filters within the same cluster, once an IIC-to-TEXTURE chunk is received. Second, better pipelining of computations and better overlap between computation and communication can be achieved with the split HCC and HPC filter implementation. Especially on the XEON cluster where the HCC and HPC filters are co-located on the same node, but run on separate processors.

In the second heterogeneous environment experiment, the XEON and OPTERON clusters were used to compare round robin and demand driven buffer scheduling policies. Four RFR filters, 1 IIC filter, 2 HPC filters, and 1 USO filter were executed on separate nodes on the OPTERON cluster. Because the HCC filter is the most computation-expensive filter, the HCC filters were used to evaluate the round robin

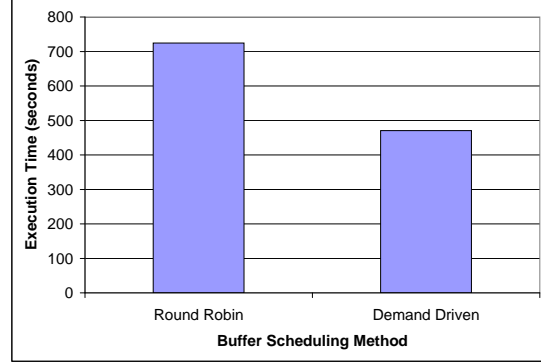


Figure 11. Performance comparison of the round robin and demand driven buffer scheduling policies.

and demand driven scheduling policies. Four HCC filters were placed on the XEON nodes and 4 HCC filters were placed on the OPTERON nodes. In this filter layout no more than one filter is assigned to any processor. When using the round robin mechanism, the DataCutter scheduler assures that all transparent filter copies receive approximately the same number of data buffers. The demand driven mechanism allows the DataCutter scheduler to assigns data buffers to the transparent copy that will likely process the data the fastest. As shown in Figure 11, the demand driven method performs better than the round robin method. Filter placement also becomes important when using the demand driven write policy. Because the OPTERON HCC filters receive more data packets in demand driven scheduling, there is less communication overhead because the HPC filters are also placed on the OPTERON nodes. In this experiment, the round robin scheduling method causes the XEON HCC filters to receive more data packets; therefore, more HCC-HPC communication overhead exists compared with the demand driven method.

These two experiments show that factors such as network bandwidth play an important role in choosing the implementation to use, filter layout, and sizes of buffers used for transferring data between two filters. For example, if network latency is high and bandwidth is low, communication overhead incurred by transmitting small buffers can outweigh the gain from more pipelining. In such a case, larger buffers might achieve better performance results. Also, filters that exchange large volumes of data can be colocated to minimize communication volume. We plan to carry out a more extensive investigation of the impact of architecture parameters on the choice of implementation in a future work.

6 Conclusions

Haralick texture analysis is a computation intensive application that involves repeated co-occurrence matrix generation and repeated computations on the co-occurrence matrices. The 4D datasets produced by time dependent imaging methods also affects the amount of computation as such datasets can be large. The storage and memory resources available on a single computer may not be sufficient to manage large datasets. We developed a parallel 4D Haralick texture analysis implementation to address these challenges. Our implementation demonstrates how operations of the texture analysis program may be data- and task-distributed to allow parallel and pipelined operation. We have evaluated different implementations and optimizations on cluster of PCs. Our results show that the split HCC and HPC filter implementation achieves good performance when sparse matrix representation is employed. The results also show that in a heterogeneous computing environment, the split HCC and HPC filter representation provides greater flexibility and improved pipelining compared to the HMP implementation.

Acknowledgment. We would like to express our gratitude to the reviewers of our paper. Their comments and suggestions helped us greatly in improving the content and presentation of the paper.

References

- [1] M. Aeschlimann, P. Dinda, J. Lopez, B. Lowekamp, L. Kallivokas, and D. O'Hallaron. Preliminary report on the design of a framework for distributed visualization. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, pages 1833–1839, Las Vegas, NV, June 1999.
- [2] L. Arge, L. Toma, and J. S. Vitter. I/o-efficient algorithms for problems on grid-based terrains. In *Proceedings of 2nd Workshop on Algorithm Engineering and Experimentation (ALENEX '00)*, 2000.
- [3] AVS: Advanced Visualization Systems. <http://www.avs.com>.
- [4] 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.
- [5] 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.
- [6] M. D. Beynon, R. Ferreira, T. Kurc, A. Sussman, and J. Saltz. DataCutter: Middleware for filtering very large scientific datasets on archival storage systems. In *Proceedings of the Eighth Goddard Conference on Mass Storage Systems and Technologies/17th IEEE Symposium on Mass Storage Systems*, pages 119–133. National Aeronautics and Space Administration, Mar. 2000. NASA/CP 2000-209888.
- [7] 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.
- [8] BrookGPU: Brook for GPUs. A compiler and runtime implementation of the Brook stream program language for graphics hardware. <http://graphics.stanford.edu/projects/brookgpu>.
- [9] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for gpus: Stream computing on graphics hardware. In *Computer Graphics (SIGGRAPH 2004 Proceedings)*, 2004.
- [10] H. Casanova and J. Dongarra. NetSolve: a network enabled server for solving computational science problems. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(3):212–223, 1997.
- [11] 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.
- [12] P. Colantoni, N. Boukala, and J. da Rugna. Fast and accurate color image processing using 3d graphics cards. In *Proceedings of 8th International Fall Workshop on Vision, Modeling, and Visualization 2003 (VMV 2003)*, Nov. 2003.
- [13] R. W. Connors and C. A. Harlow. A theoretical comparison of texture algorithms. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, PAMI-2(3):204–222, May 1980.
- [14] M. Cox and D. Ellsworth. Application-controlled demand paging for out-of-core visualization. In *Proceedings of the 8th IEEE Visualization '97 Conference*, 1997.

- [15] D. DeWitt and J. Gray. Parallel database systems: the future of high performance database systems. *Communications of the ACM*, 35(6):85–98, June 1992.
- [16] 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.
- [17] D. Fleig. DCE-MRI medical image processing using haralick texture analysis. Master’s thesis, Ohio State University, 2003.
- [18] N. K. Govindaraju, B. Lloyd, W. Wang, M. C. Lin, and D. Manocha. Fast database operations using graphics processors. In *Proceedings of ACM SIGMOD 2004*, 2004.
- [19] R. M. Haralick, K. Shanmugam, and I. Dinstein. Textural features for image classification. *IEEE Trans. on Systems, Man, and Cybernetics*, 3(6):610–621, Nov. 1973.
- [20] Haskell: A Purely Functional Language. <http://www.haskell.org>.
- [21] S. Hastings, T. Kurc, S. Langella, U. Catalyurek, T. Pan, and J. Saltz. Image processing for the grid: A toolkit for building grid-enabled image processing applications. In *CCGrid: IEEE International Symposium on Cluster Computing and the Grid*. IEEE Press, May 2003.
- [22] M. Hopf and T. Ertl. Accelerating morphological analysis with graphics hardware. In *Proceedings of Workshop on Vision, Modeling, and Visualization (VMV 2000)*, 2000.
- [23] National Library of Medicine. Insight Segmentation and Registration Toolkit (ITK). <http://www.itk.org/>.
- [24] D. C. James. Haralick texture analysis of simulated microcalcification effects in breast magnetic resonance imaging. *Master’s Thesis, Ohio State University*, 2000.
- [25] M. V. Knopp, F. Giesel, H. Marcos, H. von Tengg-Kobligk, and P. Choyke. Dynamic contrast-enhanced magnetic resonance imaging in oncology. *Topics in Magnetic Resonance Imaging*, 12(2):301–308, 2001.
- [26] M. V. Knopp, E. Weiss, H. Sinn, J. Mattern, H. Junkermann, J. Radeleff, A. Magener, G. Brix, S. Delorme, I. Zuna, and G. van Kaick. Pathophysiologic basis of contrast enhancement in breast tumors. *Journal of Magnetic Resonance Imaging*, 10:260–266, 1999.
- [27] T. Kurc, S. Hastings, U. Catalyurek, J. Saltz, J. D. Fleig, B. D. Clymer, H. von Tengg-Kobligk, K. T. Baudendistel, R. Machiraju, and M. V. Knopp. A distributed execution environment for analysis of DCE-MR image datasets. In *The Society for Computer Applications in Radiology (SCAR 2003)*, published as an Abstract, 2003.
- [28] H. Kutluca, T. Kurc, and C. Aykanat. Image-space decomposition algorithms for sort-first parallel volume rendering of unstructured grids. *The Journal of Supercomputing*, 15(1):51–93, 2000.
- [29] A. Lefohn, J. Cates, and R. Whitaker. Interactive, gpu-based level sets for 3d segmentation. In *Medical Image Computing and Computer Assisted Intervention (Miccai)*, pages 564–572, 2003.
- [30] R. A. Lerski, K. Straughan, L. R. Schad, D. Boyce, S. Bluml, and I. Zuna. MR image texture analysis - An approach to tissue characterization. *Magnetic Resonance Imaging*, 11:873–887, 1993.
- [31] D.-R. Liu and S. Shekhar. A similarity graph-based approach to declustering problems and its application towards parallelizing grid files. In *the 11th Inter. Conference on Data Engineering*, pages 373–381, Taipei, Taiwan, Mar. 1995.
- [32] E. Manolakos and A. Funk. Rapid prototyping of component-based distributed image processing applications using javaports. In *Workshop on Computer-Aided Medical Image Analysis, CenSSIS Research and Industrial Collaboration Conference*, 2002.
- [33] B. Moon and J. H. Saltz. Scalability analysis of declustering methods for multidimensional range queries. *IEEE Transactions on Knowledge and Data Engineering*, 10(2):310–327, March/April 1998.
- [34] M. Oberhuber. Distributed high-performance image processing on the internet. Master’s thesis, Technische Universitat Graz, 2002.
- [35] A. R. Padhani. Dynamic contrast-enhanced MRI in clinical oncology: Current status and future directions. *Journal of Magnetic Resonance Imaging*, 16:407–422, 2002.
- [36] W. Schroeder, K. Martin, and B. Lorensen. *The Visualization Toolkit: An Object-Oriented Approach To 3D Graphics*. Prentice Hall, 2nd edition, 1997.
- [37] SCIRun: A Scientific Computing Problem Solving Environment. Scientific Computing and Imaging Institute (SCI), <http://software.sci.utah.edu/scirun.html>.

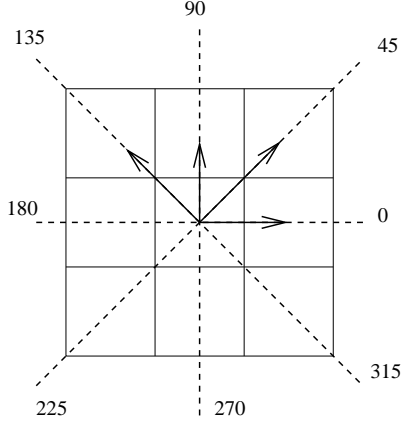


Figure 12. The directions relative to the center pixel for 2D Haralick texture analysis.

- [38] G. D. Tourassi. Journey toward computer-aided diagnosis: Role of image texture analysis. *Radiology*, 213:317–320, 1999.
- [39] 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.
- [40] B. J. Woods. 4-D haralick texture analysis of DCE-MRI datasets using distributed computing. *Undergraduate Honors Research Thesis, Ohio State University*, 2004.

Appendix

The Co-occurrence Matrix

The co-occurrence matrix measures the number of occurrences in which two neighboring pixels, one with gray level g_1 and the other with gray level g_2 , occur a distance d away and along a certain direction. Therefore, a neighboring pixel is defined by a distance and a direction from another pixel. In our implementation no interpolation was used; therefore, distance refers to the number of discrete pixels that are between two neighboring pixels. For example, pixels that neighbor at the corner, as in the 45 degree case, are still considered to be one unit of distance apart. Given a pixel with gray level g_1 , the co-occurrence matrix can be used to determine the probability that a neighboring pixel a certain distance away and along a certain direction has a gray level g_2 . For a simple 2D case, the directions can be broken down according to an angle, θ , from pixel p_1 . For 2D, these angles are 0, 45, 90, and 135 degrees (see Figure 12). The co-occurrence matrix, $p(i, j, d, \theta)$, for 0 and 135 degrees can be defined as follows.

$$\begin{aligned} p(i, j, d, 0) = & \# \{ [(k, l), (m, n)] \in (L_y \times L_x) \\ & \times (L_y \times L_x) \mid k - m = 0, |l - n| = d, \\ & I(k, l) = i, I(m, n) = j \} \end{aligned} \quad (3)$$

$$\begin{aligned} p(i, j, d, 135) = & \# \{ [(k, l), (m, n)] \in (L_y \times L_x) \\ & \times (L_y \times L_x) \mid (k - m = d, l - n = d) \\ & \text{or } (k - m = -d, l - n = -d), \\ & I(k, l) = i, I(m, n) = j \} \end{aligned} \quad (4)$$

where $\#$ denotes the number of elements in the set, $I(k, l)$ is the gray-tone value at pixel (k, l) , and $I(m, n)$ is the gray-tone value at pixel (m, n) [19].

Original work on texture analysis was only concerned with applying texture analysis to 2-D images. As a result only four unique directions need to be considered for two dimensions, and the directions were defined as angles, such as 0, 45, 90, and 135 degrees. As the dimension increases, the number of total directions also increases. In a 3-D dataset, 13 unique directions exist, and for a 4-D dataset 40 unique directions exist.

The Haralick parameter functions used in the Experiments.

Angular Second Moment

The first Haralick texture parameter used in our experiments is angular second moment. The moments of a random variable help quantify the distribution of values in the co-occurrence matrix. Given an image, angular second moment can be used to measure the amount of homogeneity in the image. Eq. 5 describes the formula used to calculate angular second moment.

$$f_1 = \sum_{i=0}^{N_g-1} \sum_{j=0}^{N_g-1} P(i, j)^2 \quad (5)$$

Correlation

The second parameter, correlation, measures the relationship between two variables using the covariance values of the two random variables. A high correlation value indicates that the pixels in an image have similar tonal values. Correlation is calculated according to Eq. 6.

$$f_3 = \frac{1}{\sigma_\alpha \sigma_\beta} \sum_{i=0}^{N_g-1} \sum_{j=0}^{N_g-1} (i - \mu_\alpha)(j - \mu_\beta) P(i, j) \quad (6)$$

Sum of Squares

Sum of squares, or variance, is important in the study of second order statistics. The sum of squares, which is also the same as second central moment, measures the spread of the distribution of the two random variables. Eq. 7 calculates the sum of squares.

$$f_4 = \sum_{i=0}^{N_g-1} \sum_{j=0}^{N_g-1} (i - \mu)^2 P(i, j), \mu = \mu_\alpha = \mu_\beta \quad (7)$$

Inverse Difference Moment

Inverse difference moment also relates to the study of second order statistics. In the calculation of this parameter, each value in the co-occurrence matrix is weighted inversely by the change in gray levels between pixels. The inverse difference moment is described by Eq. 8. Note: one is added to the denominator to avoid dividing by zero.

$$f_5 = \sum_{i=0}^{N_g-1} \sum_{j=0}^{N_g-1} \frac{P(i, j)}{1 + (i - j)^2} \quad (8)$$