Large Image Correction and Warping in a Cluster Environment

Vijay S. Kumar, Benjamin Rutt, Tahsin Kurc,
Umit Catalyurek, Joel Saltz
Department of Biomedical Informatics
The Ohio State University

Sunny Chow, Stephan Lamont, Maryann Martone
National Center for Microscopy
and Imaging Research
University of California San Diego

Abstract

This paper is concerned with efficient execution of a pipeline of data processing operations on very large images obtained from confocal microscopy instruments. We describe parallel, out-of-core algorithms for each operation in this pipeline. One of the challenging steps in the pipeline is the warping operation using inverse mapping based methods. We propose and investigate a set of algorithms to handle the warping computations on storage clusters. Our experimental results show that the proposed approaches are scalable both in terms of number of processors and the size of images.

Keywords: imaging, digital microscopy, parallel computation, PC clusters, out-of-core, warping

1 Introduction

Biomedical imaging has proven to be one of the key clinical components in diagnosis and staging of complex diseases and in the assessment of effectiveness of treatment regimens. Advances in imaging technologies such as digital confocal microscopy and digital high power light microscopy are making a dramatic impact on our ability to characterize disease at the cellular and microscopic levels. Scanner technologies for digitizing tissue samples and microscopy slides have rapidly advanced in the past decade. Advanced scanners have been developed that are capable of capturing high resolution images rapidly[1]. As a result, digital imaging of pathology slides and analysis of digitized microscopy images have gained an increasing interest in many fields of biomedicine. However, a significant challenge is the efficient storage, retrieval, and processing of the very large volumes of data required to represent a slide and a large collection of slides. High resolution scanners can capture images of single slides at $150K \times 150K$ pixels (66 Gigabytes). Processing of digital microscopy images involve a range of operations including simple 2D browsing of images, calculation of signal (color value) distribution within a sub-region of the image, extraction of features through segmentation of different cell types, and 3D reconstruction of images from multiple slides, each of which represents a different focal plane. The memory and processing requirements of large volumes of image data make analysis of digitized microscopy images good candidates for execution on parallel machines.

In an ongoing project, we are developing parallel runtime support for a pipeline of image processing operations on very large microscopy images. The purpose of this pipeline is to pre-process digitized images obtained from mouse brain tissue samples using confocal microscopy instruments. The output from this pipeline can be queried and analyzed using additional analysis methods. The pipeline consists of two main stages; *correctional tasks* and *preprocessing tasks* (see Figure 1). In our earlier work, we developed task- and data-parallel out-of-core algorithms for the pre-processing tasks stage [Rutt et al. 2005].

In this paper, we describe algorithms to efficiently execute the *correctional tasks* stage of the pipeline on parallel machines and on very large images. This stage is composed of a network of tasks (see Figure 2). We have developed parallel, out-of-core algorithms for each task. The most challenging task in this stage of the pipeline is *warping*. We employ a warping technique based on the *inverse mapping* approach. In this technique, pixels in the output image are traversed in a pre-determined order; for each output pixel the corresponding input pixel is obtained (using an inverse mapping function); and the color value of the output pixel is updated using the color value of the respective input pixel. For very large images and on parallel machines, this type of warping algorithm can result in high I/O and communication costs, because of the irregular mappings from the output image space to the input image space. We propose several algorithms to execute the warping task efficiently. We evaluate our algorithms using a PC cluster and multi-gigabyte large images. We also show that our approach is scalable to very large datasets using a synthetically generated 1-Terabyte image. Image warping can be employed in other imaging applications such as remote sensing and satellite imagery. Our proposed algorithms can also be applied in those applications.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SC2006 November 2006, Tampa, Florida, USA
0-7695-2700-0/06 $20.00 ©2006 IEEE

[1] Such scanners are being commercially produced

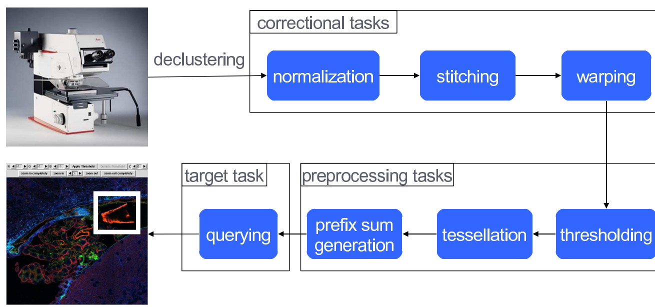


Figure 1: Overview of Larger Pipeline

2 Overview of Correctional Tasks Pipeline

An overview of the entire processing pipeline of the image analysis application is shown in Figure 1. During the image acquisition phase, the confocal microscopy instrument moves the imaging sensor in X and Y directions in fixed steps. At each step, multiple images at different focal lengths are captured. Each such image corresponds to a tile in the entire image. The movement of the sensor is such that neighbor tiles overlap each other to some extent. At the end of the image capture process, an image consists of many tiles stored in a single image file, usually with multiple Z slices. Each Z slice corresponds to a different focal plane captured by the microscope.

The *correctional tasks* stage of the pipeline is covered in more detail in Figure 2. This stage consists of a series of processing steps on images. On a cluster machine, the first step (step 1 in the figure) is to decluster the image across the nodes of the machine, yielding a Distributed IMage (.dim) metadata file. The metadata file consists of a list of host names and file locations where the declustered tiles can be found. Note that the declustering step is necessary only on a cluster machine. The remaining steps of the pipeline are as follows: a Z projection is performed on the image (step 2 in the figure). This is followed by a manual, user-interactive alignment step (step 3) in order to give an initial "seed" as to how the overlapping tiles should be stitched together to form the image. Step 4 of the correctional tasks stage is a normalization operation, which corrects for lighting variances in separate tiles. As is seen from the figure, the inputs to this step are the Z projected image and the original image. The next step is the automatic alignment operation (step 5), which determines the finalized offset of each tile in the final image by sampling points along the borders of tiles and choosing the best alignment. The outputs from the normalization and the automatic alignment steps are processed in the stitching step (step 6). This step takes the finalized offsets as input and creates the final image, and cleans up any temporary image files. Finally (step 7), the image is warped using an inverse mapping method to conform to a standard appearance (i.e., an atlas) of the subject type under study – for example, the images from mouse studies are warped so that they are aligned with a mouse brain atlas.

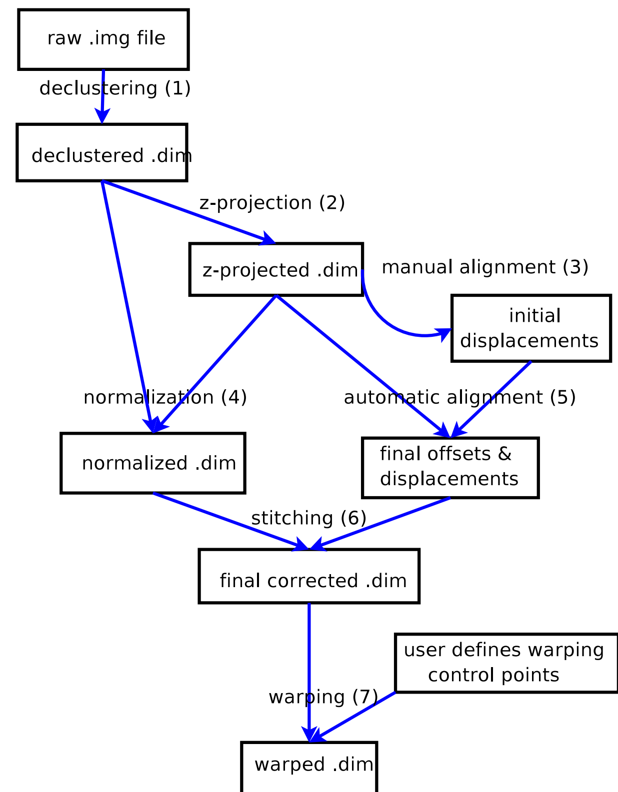


Figure 2: Tasks covered in this paper

3 Related Work

The computing and memory requirements of analysis algorithms have always been a challenge in image analysis and scientific visualization [Chiang and Silva 1999; Cox and Ellsworth 1997; Ueng et al. 1997; Arge et al. 2000; Bajaj et al. 1999]. Digital microscopy is a relatively new technology. There are relatively few projects that aim to develop server tools and runtime environments to facilitate access to and processing of digital microscopy images. The design and implementation of a complete software system was described in [Afework et al. 1998] that implements a realistic digital emulation of a high power light microscope, through a client/server hardware and software architecture. The implementation emulates the usual behavior of a physical microscope, including continuously moving the stage and changing magnification and focus. The Open Microscopy Environment (http://openmicroscopy.org) project develops a database-driven system for analysis of biological images. The system consists of a relational database that stores image data and metadata. The metadata is modeled using XML schemas and data exchange between multiple systems is accomplished through XML files that contain data and meta-

data. Images in the database can be processed using a series of modular programs. These programs are connected to the back end database; a module in the processing sequence reads its input data from the database and writes its output back to the database so that the next module in the sequence can work on it. While this approach gives a flexible implementation, the back end database can become a performance bottleneck when large volumes of data need to be processed.

Image warping, one of the steps in our pipeline, is an expensive operation in image processing. Wolberg [Wolberg 1994] describes different image warping approaches like forward mapping and inverse mapping. Wittenbrink and Somani [Wittenbrink and Somani 1993] proposed one of the earliest approaches to address the problem of parallel inverse image warping. They evaluated the algorithm on an SIMD architecture. Marcato [Marcato 1998] describes two optimizations for inverse warping: a hierarchical approach and a clipping approach, but does not address the problem from the parallelization point of view. Contassot-Vivier and Miguet [Contassot-Vivier and Miguet 1999] proposed the idea of partitioning an image into sub-regions and then performing warping on these sub-regions in parallel. Their algorithm, however, works only for forward mapping and supports only limited transformations. Jiang et.al. [huang Jiang et al. 2004] proposed a parallel load-balanced image warping algorithm using forward and inverse mapping that can support many complex warping transformations. Unlike our work, they assume the presence of a forward mapping function to begin with. They also assume that an image in its entirety, can fit on disk on a designated "manager" node when performing aggregation-based operations. This may not be possible for extremely large images. Our algorithm works in a completely decentralized fashion and has no such assumptions. Unlike all of the above related work, we propose out-of-core algorithms that place minimum restrictions on the size of the images.

4 Distributed Processing Algorithms

In this section, we describe the distributed processing algorithms for disk-resident image data for each step of the correctional tasks stage. We target as our hardware platform distributed-memory cluster machines, in which the nodes are connected to each other through a switch and each node has local disks.

4.1 Middleware System

We implemented our application using DataCutter [Beynon et al. 2001], a component-based middleware framework. The DataCutter framework provides a coarse-grained data flow system and allows combined use of task- and data-parallelism. In DataCutter, application processing structure is implemented as a set of components, referred to as *filters*, that exchange data through a *stream* abstraction. A *stream* denotes a uni-directional data flow from one filter (i.e., the producer) to another (i.e., the consumer). The DataCutter runtime system supports execution of filters on heterogeneous collections of storage and compute clusters in a distributed environment. Multiple copies of a filter can be created and executed, resulting in data parallelism. The runtime system performs all steps necessary to instantiate filters on the desired hosts, to connect all logical endpoints, and to invoke the filter's processing function. Each filter executes within a separate thread, allowing for CPU, I/O and communication overlap. Data exchange between two filters on the same host is carried out by pointer hand-off operations for languages that support pointers (C++), while message passing is used for communication between filters on different hosts. For this application, we used a version of DataCutter which employs MPI as the message passing substrate. Individual filters may be written in C++, Java or Python, and can be mixed freely within the same workflow.

4.2 Image Storage and Data Declustering

An image is a raw 3-channel "BGR planar" (Blue-Green-Red planar) image. Each image consists of a set of *tiles*. A tile contains pixels in a rectangular subregion of the image. It is associated with a bounding box, defined by a $[(x_{min}, y_{min}), (x_{max}, y_{max})]$ tuple, in X and Y dimensions of the image and a Z value. The Z value indicates which Z slice the tile belongs to. The on-disk representation of a tile is one where all the pixels for the B channel are laid out in row major fashion, followed by all pixels for the G channel, and then the R channel. The rationale for using this BGR planar format is twofold: (i) The original image files are in a proprietary format called "img", where pixels are stored in the BGR planar format. Instead of using other formats, we retained this data layout in our distributed image representation. In this way, we avoid the cost of performing multiple seeks within the image file during the declustering stage. (ii) In our earlier work [Rutt et al. 2005], vector operations were used for low-level parallelism whenever Intel MMX/SSE instructions are supported. For instance, in the thresholding step, we apply the same operation to each color channel, but with possibly different parameters per channel. The MMX instructions performed the operations on multiple neighboring data at the same time. To benefit from these vector operations, we needed to ensure that data (pixels) of the same channel are laid out contiguously. So, we used the BGR planar format in the *preprocessing tasks* stage of our pipeline. For the purpose of uniformity in data layout in both the *correctional tasks* and the *preprocessing tasks* stages of the pipeline and in the absence of clear reasons that justify the use of any alternative format, we retained the BGR planar

format for this paper.

The tiles are then stored in multiple files or multiple contiguous regions of one or more files on disk. A tile represents the unit of I/O. On a cluster system, tiles are declustered across files stored on different compute nodes. A compute node "owns" a particular set of tiles, if the compute node has the tile data available on local disk.

The first step in our pipeline is declustering; that is, converting a single image file into a distributed image file consisting of tiles spread across compute nodes. Some of the operations such as the automated alignment step in the pipeline need access to neighbor tiles during execution. In order to minimize inter-processor communication in such steps, the neighbor tiles are grouped to form blocks. A block consists of a subset of tiles that are neighbors in X and Y dimensions of the image. All the tiles that have the same bounding box, but different Z values are assigned to the same block. That is, the raw image captured by the microscope is partitioned in X and Y dimensions, but not in Z dimension. A tile is assigned to a single block, and a block is stored on only one node. The blocks are declustered across the nodes in the system in round-robin fashion. A node may store multiple blocks.

4.3 Common Set of Filters

For each step of the processing pipeline, we have implemented filters that are specific to processing applied in that step. There are also a set of filters that are common to all steps and provide support for reading and writing distributed images. The *mediator* filter provides a common mechanism to read a tile from an input image, regardless of whether it is on local or remote disk. It also provides a means to write out a new tile on local disk for an image currently being created, and a mechanism to finalize all written tiles at once into their final output directories, such that total success or total failure in writing a new image transpires. The *mediator* filter receives requests from client filters and works with other *mediator* filters on other compute nodes to carry out input tile requests. The actual I/O to a local filesystem is delegated to a *mediator_reader* filter, which receives requests from the *mediator* on the same compute node. The *rangefetcher* filter is used to hide data retrieval latency. It issues a series of requests on behalf of its client filter to fetch a number of tiles in a sequence. This way, the *rangefetcher* can work slightly ahead of the client filter, minimizing tile retrieval (either from local disk or from a remote node) latency for the client filter. That is, when a client filter is working on tile T_1, the *rangefetcher* filter is reading tile T_2, such that when the client needs T_2, it is ready and waiting in memory, reducing read latency. These filters are shown in Figure 3.

4.4 Z Projection

The goal of the Z projection phase is to aggregate multiple Z planes of an input image into a single output image, using the *max* aggregate operator. That is, each (x,y) pixel in the output image will contain the brightest or maximum corresponding (x,y) pixel value across all Z planes in the input. This stage is completely parallelizable since in the previous step we divide the image among compute nodes in the X or Y direction, not in the Z direction. Each compute node can Z project the data housed on its own local filesystem, independent from any other compute node. A single *Z Projection* filter per node does the job, along with the help of the mediator.

4.5 Normalization

The goal of the normalization phase is to correct for the variances in illumination that may exist between distinct tiles. These variances hinder efforts at the automatic alignment phase to follow. As a result, there may be unsightly gradient-like seams in the final output image. Normalization is critical to creating a seamless mosaic of tiles. One of the steps in normalization is to compute the average intensity tile for the Z projected plane. Given a Z plane, the average tile for that plane is one where the pixel value at a position (x,y) within the tile is the average of the pixel values at position (x,y) within all tiles in that plane. So, to compute this average tile, we need data from the entire Z plane.

The idea of using this average pixel based approach assumes that the illumination gradient is uniform for each tile in the Z projected plane. The act of computing an average tile for a plane then reinforces what is common across the data and minimizes image specific features thereby giving us an approximation of the illumination gradient. In addition to computing the average tile, an additional offset tile is computed by taking a minimum projection across all the data in the plane. The contributions of the approximated illumination gradient and the pixel offsets to each tile are then removed to give us a corrected dataset. Further details on the normalization technique can be found in [Chow et al. 2006].

Each compute node needs the average tile for normalization. However, each node owns only a part of the Z projected plane. In our implementation, we have each node initially compute the average tile based on the portions of the Z plane it owns locally. We then partition this tile uniformly into P parts, where P is the number of nodes. The nodes then communicate in a ring fashion, where each node sends its part of this tile to its neighbor. On receiving a part, each node adjusts its average tile to reflect the average of all parts received up to that point. After two passes of P-1 communications each, all nodes will have the finalized average tile. Our approach requires just $2 \times C$ amount of data transfer. Here, C is the size of a tile. The same procedure is used to compute the offset tile. Using the average and offset tiles, each

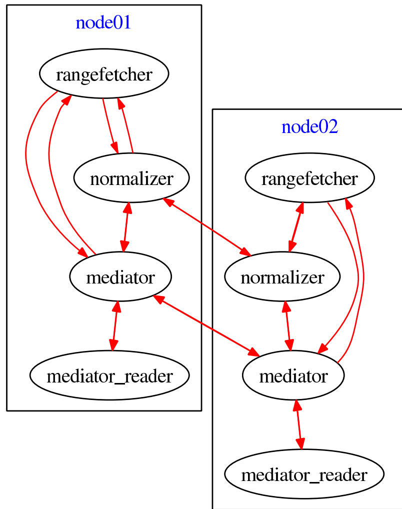


Figure 3: Low-Level Normalization Workflow

node normalizes the portions of the image it owns. This is completely parallelizable. Normalization is first performed on the Z projected plane and then on each individual Z slice of the original declustered image.

As an example of the DataCutter workflow layout we use in a typical phase of the larger pipeline, refer to the normalization diagram of Figure 3. Each oval represents a running filter or thread on a given compute node, and each arrow represents a communication line. A single normalization filter per compute node does the job, along with the help of the mediator.

4.6 Automatic Alignment

The goal of the automatic alignment phase is to determine how the distinct partially overlapping tiles should converge upon one another to produce a properly aligned image, eliminating all the overlapping regions. The output of this phase is a list of finalized offsets (into the output image) for each and every tile. That is, the upper left corner position of each tile in the output image will be known. The phase after alignment, which we refer to as stitching, does the actual job of producing the final image, given the output from the automatic alignment phase.

The automatic alignment algorithm first computes candidate alignment displacements for each pair of adjacent tiles within a single Z plane. Horizontal and vertical alignments are processed separately. Each alignment takes two adjacent (overlapping) tiles as input, and performs feature detection without scanning the overlapping region in its entirety. Feature detection is performed by filtering each tile with a Sobel filter [Gonzalez and Woods 2001] to compute the gradient of the tile. The highest values from the gradient are then regarded as features. Features are then matched across two adjacent tiles by performing a normalized cross correlation of a window centered on a tile feature from one tile and a region centered on the expected location of the tile feature (found in the first tile) on the other tile. Each horizontal and vertical displacement is given a score. The score used to integrate the horizontal and vertical displacements is the number of feature correspondences found for that particular displacement; the higher the score the better the alignment.

For an image with c columns and r rows, there will be $(c-1) \times r$ horizontal alignment displacements, and $c \times (r-1)$ vertical alignment displacements. Reinterpreting each tile as a graph node and each displacement as a graph edge, the maximum spanning tree of the graph (maximizing the displacement score) will compute the best alignment. That is, the displacements with the highest score will be used to align the tiles. When this phase is complete, the finalized offsets of every tile are saved and passed to the next phase. Further details on the automatic alignment step can found in [Chow et al. 2006].

The decomposition of work across compute nodes is as follows. Each compute node is responsible for computing alignments between any tiles it owns and the tiles immediately to the right and below in X and Y dimensions. Most of the time, this means that any pair of alignments require reading two tiles from local disk and computing their alignment. However, on the borders between tiles owned by one compute node and another compute node, some network communication is necessary to process the alignments. We have observed in our experiments that when the images are large relative to the number of compute nodes, this border communication cost becomes insignificant. That is, for multi-gigabyte size images, the image dimensions consist of tens of thousand of pixels which is much greater than the number of nodes used (16-32). For automatic alignment, we use an alignment filter per compute node and a single maximum spanning tree filter placed on one node. When all alignments have been made and scored, the maximum spanning tree filter chooses the best alignments and saves the result to disk.

4.7 Stitching

The goal of the stitching phase is to combine the partially overlapping tiles into non-overlapping tiles, producing a final non-overlapping image on disk. As input, this phase takes in the finalized offsets produced by the automatic alignment step. In practice, these offsets place the output tiles in their natural position after getting rid of the overlap, plus or minus a small number of pixels in the X and/or Y direction. Since the output image is guaranteed to be smaller than the input image (due to the fact that neighboring tiles in the input will overlap), this stage must do a mapping. In order that we preserve as much locality as possible, we map in such a way that the number of tiles and which compute node they reside on are preserved. The decomposition of work across compute nodes in the stitching step is as follows. Each compute node is responsible for stitching any tiles it owns in the out-

put space. We make a one-to-one correspondence between ownership regions in the input and output space. That is, if a compute node owned the first two rows of tiles in the input space, it would own the first two rows of tiles in the (smaller) output space. In the majority of cases, the image data in the output space derives from image data in the input space on the same compute node. However, on the borders between tiles owned by one compute node, and another compute node, some network communication will be necessary to merge the final result. A single stitching filter per compute node executes this step.

4.8 Warping

The goal of image warping [Wolberg 1994] is to geometrically transform an image; for our purposes, this means conforming our stitched image (from the previous phase) to a standard appearance, i.e., to a pre-defined 2D atlas. Having standard image templates facilitates comparison between multiple images of the same subject type. Image warping involves the application of a *mapping function* that defines the spatial correspondence between points in the input and output images. Examples of such functions include affine, bilinear or polynomial transformations [Wolberg 1994]. In 2D, mapping functions f_1 and f_2 map a point (x,y) in one image onto a point (u,v) in the other image as follows: $[u,v] = [f_1(x,y), f_2(x,y)]$. In some warping algorithms, a *forward mapping* function is defined and applied to each input pixel to produce its output pixel location. In other warping algorithms, an *inverse mapping* function is defined and applied to each output pixel location to determine which input pixel to draw from to produce the output pixel. We employ inverse mapping instead of *forward mapping*, because it ensures that all output pixels are computed, and prevents "holes" from occurring in the warped image.

It must be noted that we do not perform a 3D warp (warping across multiple Z planes), and that the 2D warping of each Z plane is independent of the other planes. Each Z plane is just a 2D image. So, view interpolation techniques like triangle-based warping[Fu et al. 1998] that help in visualizing depth-related information are not applicable in our case.

In our case, the user interactively defines a set of *control points* over the image using a graphical interface called Jibber[2], i.e. for each control point, the user identifies its position in the input image, called "start" and its corresponding position in the output image, called "end". The start and end of each control point indicates the user's requested change in the image, that pixels at or near the start in the original image should move to a corresponding position at or near the end in the warped image. These control points as a set describe how the image should morph. Figure 4 shows how the control points can be specified to warp an image of a portion of the brain so that it fits a standard brain atlas. On the left we see the unwarped image (colored region) overlay the atlas. On the right, we see the same image warped so as to fit the atlas dimensions. For a fairly large dynamic set of images, it may be possible to automate this initial step. For example, a feature extraction algorithm could probably find outlines and void areas in the input image and match them against the contours from the output image (i.e. the atlas). Such an approach may not scale well to very large images. So, as image size grows, we could use a low-resolution version of the image in the automated control point generation process. We used a manual approach because it is more general. Partial sections and sections of the image that are mangled in some way can still be registered with the atlas.

[2] http://ncmir.ucsd.edu/distr/Jibber/Jibber.html

Given a set of control points, we generate *approximate* inverse mapping functions to characterize the spatial correspondence and map the remaining points in the image. That is, given any pixel p_o's location in the output space, we can determine which candidate input pixel p_i will contribute to pixel p_o with minimum error. This is the first phase of inverse warping, which we refer to as *computation*. In the second phase, which is called *mapping*, we assign the RGB color values at p_i to p_o. Thus, the warping algorithm must iterate over the output space and determine which areas of the input space to read from. This correspondence is modeled using low-order polynomials. We use the *Weighted Least-Squares with Orthogonal Polynomials* technique [Wolberg 1994] to determine the inverse mapping transformation (of some order M) for each output pixel. This technique is very compute intensive. Other faster techniques apply a single global transformation to all pixels. However, this weighted technique takes pixel - control point locality information into account to generate different transformations for each pixel, and thus provides better results[3].

The computational workload is divided among the compute nodes at the granularity of a tile. If a compute node owns a given input tile $IS_{x,y,z}$, then it will eventually own the output tile $OS_{x,y,z}$ after the warp; that is, the tiling divisions do not change, nor does the ownership of the corresponding tiles. Each node performs computation on the output tiles that it owns. In case the nodes are SMP machines, we have multiple filters on these nodes, where each filter performs the warping computation in parallel on different tiles. Thus, the initial declustering of the image dictates load balance.

The bigger challenge we face in warping is the mapping of input pixel RGB values to an output pixel in a distributed environment. Here, an output pixel location within one tile may have its input pixel lie within the same or different (neighboring or non-neighboring) tile. In either case, we need to read the input tile to do the mapping. Reading input tiles on a per-pixel basis will be costly due to seek and network overheads. We do not know up front which input tiles will be needed

[3] We should note that our implementation is such that a user can easily plug-in any warping technique to perform many kinds of warping transformations.

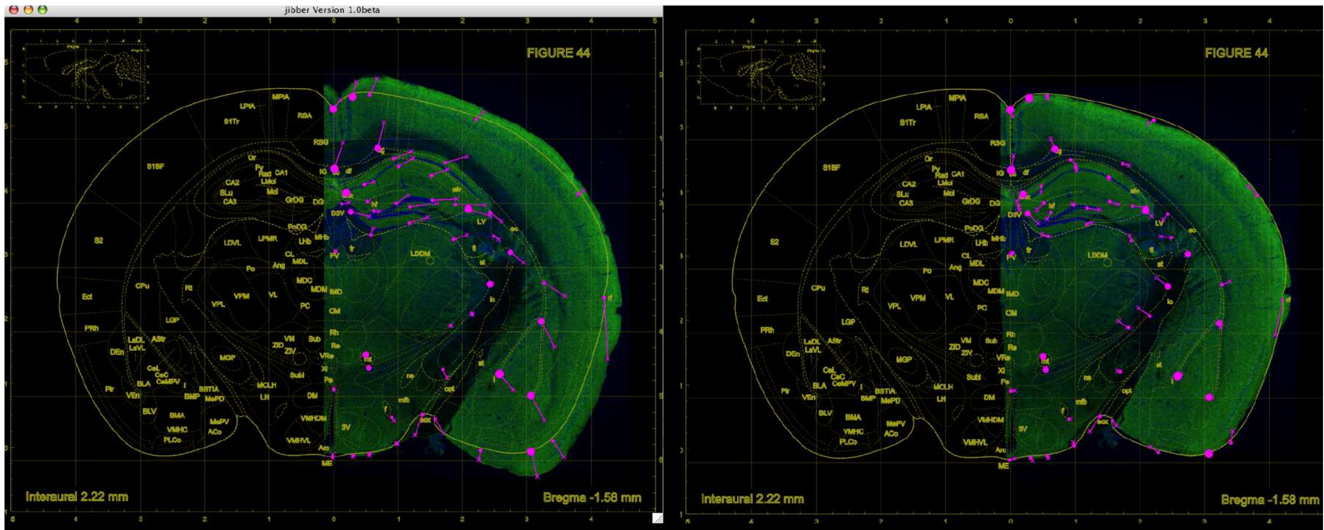


Figure 4: Warping of a brain mosaic to a standard brain atlas

to map a particular output tile, without applying the inverse mapping function over the entire output tile. Each compute node iterates over its output space in some order, one tile after another. As soon as it has finished warping a tile, that tile is written to local disk. In this way, each output tile is written exactly once. Our overall goal is to reduce overall time taken to warp the image, and to reduce consumption of system resources; to achieve these goals, we attempt to minimize computation and I/O overheads.

We now present several algorithms for parallel image warping acting on distributed, disk-resident image data when only an inverse mapping function exists.

4.8.1 Serializable Mappings

The *serializable mappings* (abbreviated as *SM*) algorithm makes an initial pass over the output image, where each compute node, for any output data it owns, will up front compute all mappings from output pixel positions to input pixel positions. These mappings are then communicated as requests to the nodes that own the corresponding input pixels. Next, each node will gather the mappings involving input pixels it owns and store them sequentially in a local disk *log* file ($logfile_a$), using one log file per input tile.

Once all mappings are built, each compute node will then iterate over the requested input tile log files one at a time, read the corresponding input tile and satisfy all pixel requests in the log file, communicate each RGB pixel value to the compute node that owns the corresponding output pixel, and purge the log file. Upon receiving an RGB pixel value targeting an output pixel location, each compute node will store the pixels sequentially in another local disk log file ($logfile_b$), again organized with one log file per output tile. Finally, when all instances of $logfile_b$ are complete, each compute node will, one at a time, replay its $logfile_b$ files, finalize its output tiles, write it out to permanent disk, and purge the log file.

In addition to the above, we use one optimization: as we are iterating over output tiles to compute the mappings, we also read the corresponding (at the same tile position) input tile, in the hopes that some of the output pixels will draw from the corresponding input tile. For any pixels that do draw from the same tile, we look up the RGB value from the input tile, bypass $logfile_a$ and directly create $logfile_b$ entries.

In effect, we create a global schedule of data movement that eliminates all communication of whole tiles between compute nodes. Instead, only the mapping requests and values are communicated. Each node reads only the tiles it owns and does so at most twice (once for the optimization mentioned above and once during $logfile_a$ replay). However, the storage overhead of this algorithm is high. Each entry in our $logfile_a$ files is 40 bytes while each entry in our $logfile_b$ files is 19 bytes as compared to a 3 byte pixel. In the worst case, we could have a $logfile_a$ entry for each pixel, in which case the collective storage overhead over all compute nodes would amount to $\frac{40}{3}$ times the image size. This is not feasible when we deal with large images. Also, larger the log files, greater the file handling overhead.

4.8.2 On Demand Mapper

Our next warping algorithm is called *on-demand mapper* (abbreviated as *ODM*). Its approach is to fetch input tiles on demand to satisfy the needs of a small set of "active" output tiles. The formal algorithm is shown in Figure 5. In the algorithm, inverse pixel mappings for a set of N output tiles, which a processor owns, are computed before any input tiles are retrieved. During this step the algorithm computes the set of input tiles which are required to create the pixel values for the active output tiles. Since the input image is

partitioned regularly into rectangular tiles, the complexity of finding the input tile for a given input pixel is $O(1)$. After this pre-computation step, each needed input tile is retrieved (either read from local or remote disk) into memory one at a time. All needed pixels from the input tile are applied to whichever of the N output tiles need them, before the input tile is evicted from memory. Once all the input tiles have been retrieved and processed, the output tiles have been created. They can be written to disk, and the algorithm moves on to the next set of N output tiles owned by the node.

The memory requirement of this algorithm is larger than that of the other algorithms. If N defines the number of output tiles to be processed at a time and S is the size in bytes of an average tile, then the total amount of main memory required during processing is equal to the sum of (1) $N \times S$ for the output tiles, (2) $N \times S \times \frac{32}{3}$ for the stored mappings and (3) S for the one input tile being read at a time. The constant in (2) refers to the fact that each pixel in a tile takes up 3 bytes, but the mapping M in Figure 5 takes up 32 bytes per pixel (8 for each of x and y location, 8 bytes for a pointer, and another 8 bytes for the size of a tile).

The ODM algorithm reduces the number of times an input tile must be read. If there are T_n output tiles, then in the worst case, each input tile must be read $\frac{T_n}{N}$ times. In addition, within a "run" of handling N output tiles, this algorithm is in effect a perfect scheduler; it schedules all accesses for a given input tile such that they all fall sequentially.

Input:
Input image I, with corresponding array of tiles T_I
Number of x, y tiles as $xmax, ymax$
Number of output tiles N to process together
Algorithm:
01 From I, implicitly derive output image O, with corresponding set of tiles T_O.
02 $M \leftarrow \{\}$ (dictionary mapping input tiles to an array of [input pixel location, output pointer] tuples)
03 $A \leftarrow []$ (array of active output tiles)
04 $n \leftarrow 0$ (size of A)
05 for $y = 0$ to $ymax - 1$
06 for $x = 0$ to $xmax - 1$
07 $t_I \leftarrow T_I(x, y)$
08 if on_local_disk(t_I)
09 $t_O \leftarrow$ "allocate new output tile"
10 append t_O to A
11 for every pixel p_O in t_O
12 compute input pixel p_I that maps to p_O
13 $ptr \leftarrow p_O$'s array location within t_O
14 append mapping $t_I \rightarrow [p_I, ptr]$ to M
15 $n \leftarrow n + 1$
16 if $n = N$ or ($x = xmax - 1$ and $y = ymax - 1$)
17 for every t_I in M
18 read t_I into memory from local/remote disk
19 for every $[p_I, ptr]$ associated with t_I
20 set ptr to RGB value at p_I
21 for every t_O in A
22 write t_O to local disk
23 deallocate t_O
24 $M \leftarrow \{\}, A \leftarrow [], n \leftarrow 0$

Figure 5: ODM Algorithm

4.8.3 Asymmetric Traveling Salesman Problem Scheduler

The *Asymmetric Traveling Salesman Problem scheduler* (abbreviated as *ATSP*) seeks to eliminate the cost of additional storage from the *SM* algorithm and the memory requirements of the *ODM* algorithm. This however comes with a price, of having to make two passes over the output space and thus, perform two rounds of computation.

In the first pass, during computation, each node *implicitly* partitions each output tile it owns into as few disjoint subtiles as possible (where each subtile contains one or more neighboring points in row-major order in the tile), such that the set of all input tiles required to map all points in each subtile can fit in memory on that node. We denote the set of input tiles corresponding to a subtile as the "input-set" of that subtile. A subtile could range in size from a single point to one or more partial or full rows of the tile to the entire tile itself. Not all subtiles necessarily contain the same number of points. The input-sets could also be disjoint, but for most *well-behaved* warping transformations, for a given output tile, there is a fair amount of overlap, because points in different subtiles could draw from the same tile. Each node maintains an in-memory dictionary mapping each subtile to its input-set. In the worst-case, where each subtile is a single point, this dictionary has an entry for every pixel, and so the memory requirements of this algorithm will be as large as those of *ODM*, if not larger. But in practice these subtiles are much larger, so the memory needs will be less than that of *ODM*.

We now focus on minimizing the volume of data transfer. An input tile must be read once for every input-set that contains it. But since the input-sets overlap with each other, an input tile once read, could be cached and reused for multiple input-sets. Each node determines a local schedule of data movement that minimizes the number of input tile reads. That is, when servicing a subtile P_1, if the immediate next subtile P_2's input-set overlaps with that of P_1, then we cache the common input tiles in memory when they are initially read as part of P_1's input-set and thereby service (a subtile or all of) P_2's requests, without having to re-read these tiles.

In order to minimize the data retrieval overhead, each compute node must traverse the subtiles in its output space in such an order that results in high overlap of the input-sets between successive subtiles. The problem of determining such an order is analogous to the well known Traveling Salesman Problem (TSP)[Lawler et al. 1985]. Here, for each compute node, we interpret each subtile as a vertex of a graph. The

set difference between the input-sets of any two subtiles is the weight of the edge between the corresponding vertices in the graph. Hence, the cost of traveling from one vertex to another is the amount of input tile data that needs to be read to service all requests in the destination's input-set. Since set difference is not always commutative, the resultant graph is asymmetric. So we have an instance of an asymmetric TSP (hence the name *ATSP scheduler*). Each compute node, then determines a least cost path that starts at any subtile and traverses every subtile exactly once [4]. There are many heuristic algorithms to solve the ATSP. For our work, we use the effective LKH [Helsgaun 2000] implementation of the Lin-Kernighan heuristic [Lin and Kernighan 1973]. We observed that LKH, like other TSP-solvers, was rather slow for large graphs. For a large number of subtiles, solving for the TSP was contributing significantly to the overall warping time, thereby undoing the benefits of reducing the subtile reads. Thus, to handle a large number of subtiles effectively, we break down the ATSP problem into smaller instances, where each instance handles all subtiles within a single output tile. The number of subtiles within each output tile tends to be small when compared to the total number of subtiles. A compute node solves each instance, i.e. it determines the order of traversal of the subtiles within each output tile it owns, such that the number of input tile reads for each output tile is minimized.

A disadvantage of the above approach is that we lose out on potential reuse of input tiles across different output tiles. So, as a current extension, we are trying to view our original problem as a hierarchical TSP: At the higher level, each compute node needs to determine an order of traversal of output tiles, so that the overall number of input tile reads is minimized. One approach to solve this is to model it as a specific instance of the Generalized Traveling Salesman Problem (GTSP) [Lawler et al. 1985]. The GTSP consists of determining a least cost tour passing through each of several clusters of vertices of a graph. Each output tile can be interpreted as one such cluster and the subtiles within each tile as the vertices within that cluster. Our notion of edge weight is the same as before. Each compute node needs to determine a least cost path that starts at any cluster, traverses every cluster exactly once and passes through exactly one vertex within each cluster. To solve a GTSP, we need to transform it into an instance of ATSP of equivalent or larger size[Ben-Arieh et al. 2003]. Here again, we expect the resulting problem to be so large that LKH will consume a large amount of time. We are currently looking at different approaches to solve this problem. At the lower level, each node solves a small ATSP instance within each output tile. This way, each node determines an optimal path that covers the entire output space.

Once the least cost path for output space traversal is determined, each node makes its second pass over the output tiles it owns; only, this time it services input tile requests in the order determined by the ATSP scheduler. For a subtile within an output tile, we read its entire input-set at once (by definition, this is guaranteed to fit in memory). We re-compute the input pixel for each point within the subtile. This input pixel is already in memory, and so we assign its RGB color values to the output point. When this is repeated for every subtile, the entire output tile is finalized and can be written to local disk.

[4] Actually, the problem is one of finding the shortest Hamiltonian path in an asymmetric graph, but this can easily be reduced to an instance of ATSP.

5 Experimental Results

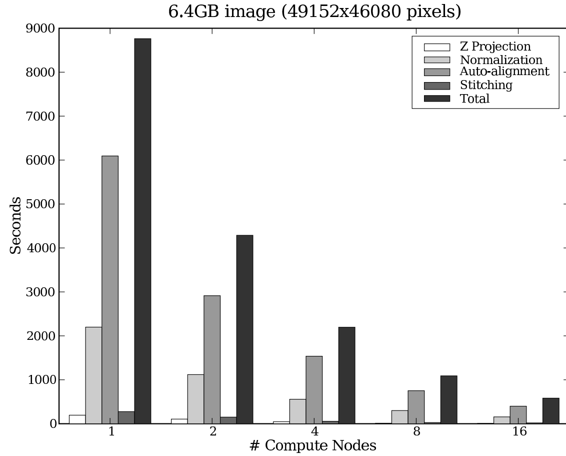
Our experiments were carried out using a cluster of 32 compute nodes. The cluster, which was provided through an NSF Research Infrastructure grant, consists of dual-processor nodes equipped with 2.4 GHz AMD Opteron processors and 8 GB of memory, interconnected by both an Infiniband and 1Gbps Ethernet network. The storage system consists of 2x250GB SATA disks installed locally on each compute node, joined into a 437GB RAID0 volume with a RAID block size of 256KB. The maximum disk bandwidth available per node is around 35 MB/sec for sequential reads and 55 MB/sec for sequential writes.

For all of our experiments, we maintained more-or-less square images to emulate what would be expected in real applications. To scale up images for large-scale experiments, we would generate replicas of smaller images originating from a confocal microscope, reproducing the tiles to the right, below, and the lower right. This enabled an easy scale-up by a factor of 4.

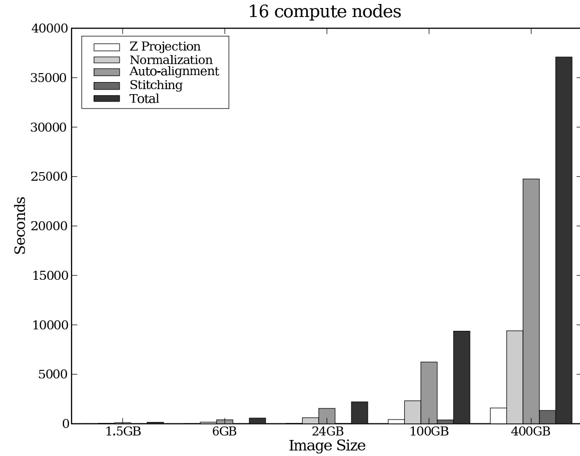
5.1 Z Projection, Normalization, Autoalignment, and Stitching Steps

In these experiments we evaluate the scalability and relative cost of each step of the image processing pipeline. In Figure 6(a), we apply Z Projection, Normalization, Autoalignment and Stitching to a 6GB input image (49152x46080x1 pixels), while varying the number of nodes from 1 to 16. This figure shows the relative performance of each stage. We observe that autoalignment is the most expensive in all cases. We also conclude that as the number of compute nodes are doubled, the total runtime is roughly cut in half, thus almost linear speedup is achieved in each of these steps.

The next set of experiments look at the performance of each step when the size of the dataset is scaled. In Figure 6(b), we apply the same processing steps, using 16 compute nodes, to a 1.5GB, 6GB, 25GB, 100GB and 400GB image. In each step, the image size is increased by a factor of 4, so we should expect the runtime to increase by a factor of 4, if we are scaling linearly. Our results confirm linear scalability, since the total runtime increases by no more than a factor of 4 with each dataset size increase.



(a) Corrective Pipeline, Scaling # of Compute Nodes



(b) Corrective Pipeline, Scaling Image Size

Figure 6: Z Projection, Normalization, Autoalignment and Stitching

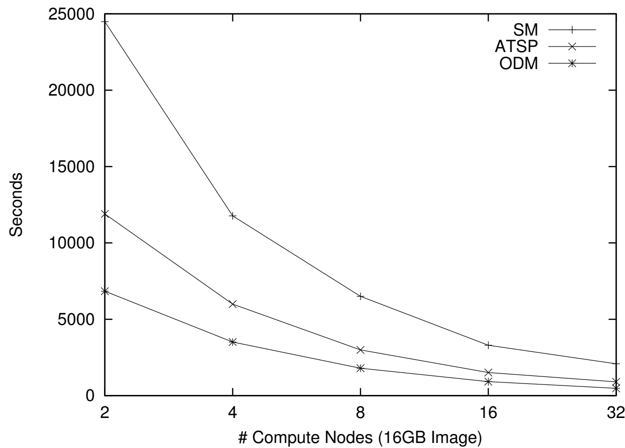


Figure 7: Scaling Compute Nodes While Image Size is Fixed

5.2 Warping

In this set of experiments, we evaluate the performance of the different algorithms proposed for the parallel execution of the warping step. In the experiments, the control points (which define the inverse warping function) were chosen such that the image rotated by 10-20 degrees.

In Figure 7, we observe performance implications of increasing the number of compute nodes, acting on an image of size 16GB, for each of the three warping algorithms. The algorithms all scale close to linearly; the total time taken is cut in half as the number of compute nodes are doubled. The ODM algorithm clearly performs best in all cases. It should be noted that the SM algorithm used a peak temporary file storage of 193 GB, or 11.91 times the size of the original image; so the worst case of 13x storage for SM was nearly hit. The total percentage of the time spent in computation (computing pixel mappings and maintaining in-memory data structures) compared to total time (averaged across all runs), was 39% for SM, 89% for ATSP and 80% for ODM. These results indicate that I/O overhead is high in SM; it spends most of its time in creating and playing back log files. Not surprisingly, the ATSP algorithm spent the largest percentage of its time in computation, with its two passes over each pixel. These results also show that the computation time for this type of warping is significant.

In Figure 8, we fixed the number of compute nodes at 16 and increased the image size from 1GB to 256GB; the total cluster memory for 16 compute nodes combined is 128GB. As seen from the figure, the algorithms scale close to linearly; the total time taken by each algorithm increases proportionally to the increases in image size. The ODM algorithm clearly performs best in all cases. This can be attributed to the better I/O performance of ODM compared to SM and its lower computational overhead compared to ATSP.

Table 1 shows the total number of input tile reads for a 64GB image for the ODM and ATSP algorithms. The number of input tile reads is smaller in the ATSP algorithm than in the ODM algorithm. The order in which sub-tiles are processed, as computed by ATSP, achieves better use of input tile cache in our implementation. However, as is seen in Figure 8, ODM has better overall performance. The computational overhead in ATSP due to traversal of the output space twice is larger than the performance gain obtained due to reduction in I/O overhead on our hardware configuration and for the inverse mapping functions used in our implementation.

In Figure 9, we observe performance implications of increasing the size of each tile, while keeping the number of tiles

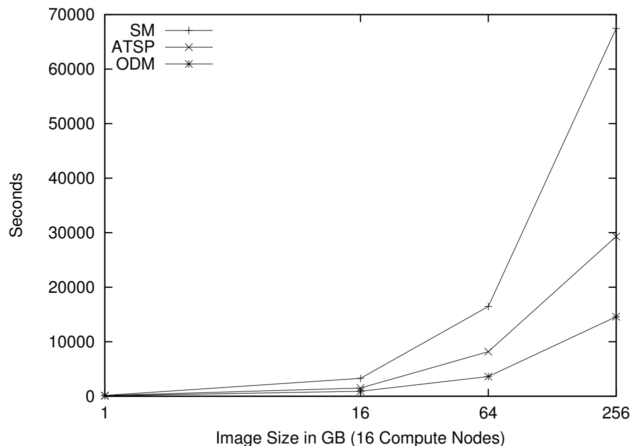


Figure 8: Scaling Image Size (16 compute nodes)

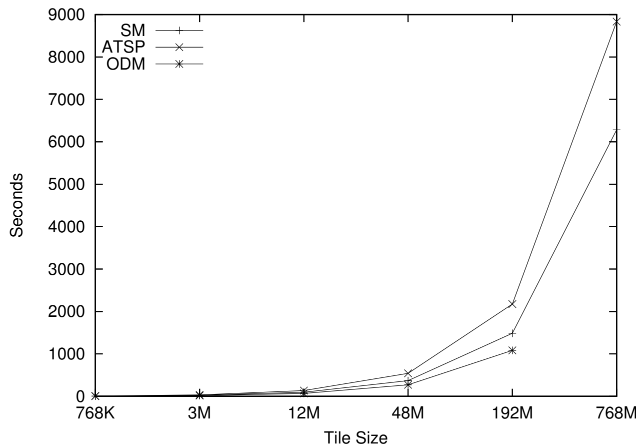


Figure 9: Scaling Tile Size (4 compute nodes, 16 tiles total, 4 per compute node)

Algorithm	**Input Tile Reads**		
	Total	Local	Remote
ATSP	171736	10713	161023
ODM	206182	12834	193348

Table 1: Input Tile Reads for 64GB Image in ATSP and ODM.

in X and Y fixed at 4, and keeping the number of compute nodes fixed at 4. The ODM algorithm performs best for most of the sizes; however, with a tile size of 768MB, the ODM algorithm ran out of memory (hence the missing data point). This is expected because of the memory requirements of ODM as discussed in Section 4.8.2. The memory requirements of ODM can be reduced by partitioning the output tile into smaller sub-tiles. Such a partitioning, however, may still result in increased I/O overhead because input tiles may be retrieved multiple times if they are required by different sub-tiles. Alternatively, the mappings that are maintained in memory can be stored on disk, as in the SM algorithm. In that case, we expect that the performance of ODM would be similar to that of SM. We plan to investigate these strategies in a future work. The SM algorithm does better at the 768MB tile size, however if disk space is limited, it will be impossible to execute it. Thus, the precise situation to apply the ATSP algorithm is when there is not sufficient extra disk space available to execute the SM algorithm, and the tile size is large enough that the ODM cannot be executed either.

We also executed the ODM warping algorithm on a 1 Terabyte image to test scalability to very large images. The input image had 690276x534060 pixels, 3 channels, 138 tiles in the x direction, 4416 tiles in the y direction, and 1.73 MB per tile. We used 16 compute nodes. The total time taken was 68830 seconds (about 19 hours), a close to linear scaleup from the 256GB case; the amount of computation time was equal to 60860 seconds and the I/O time was 7962 seconds (12% of the total execution time). The average number of tiles read per node was 206106, with a standard deviation of 208. Thus, the balance in workload among all compute nodes was quite good, for this experiment.

6 Conclusions

In this paper we have investigated some of the trade-offs in designing an image processing pipeline that supports large images. For the Z projection, normalization, automatic alignment, and stitching steps, we have shown that our algorithms scale both in terms of number of compute nodes and image size. For the warping step, our results show that the ODM algorithm performs well in most cases, as it does not compute the inverse mapping for each pixel twice like the ATSP algorithm does, and it does not spool temporary data to disk like the SM algorithm does. It can also scale to Terabyte-sized images. For parallel inverse warping, this algorithm is a good choice unless the tile size is very large. If the tile size is very large, the SM should be applied if disk space was plentiful, otherwise the ATSP algorithm should be applied. We expect that the SM algorithm can perform well, provided sufficient temporary disk space is available and I/O and communication bandwidth is high; however, the cost of its additional I/O may prove to slow it down significantly as the images grow large. The ATSP algorithm achieves poor performance compared to the other algorithms in our experimental setup, since it does twice the inverse mapping computation on every pixel, and we have observed that this computation is expensive.

7 Acknowledgements

This research was supported in part by the National Science Foundation under Grants #CNS-0203846, #ACI-0130437, #CNS-0403342, #CNS-0406384, #CCF-0342615, #CNS-0509326, #ANI-0330612, #CNS-0426241, and by

NIH NIBIB BISTI #P20EB000591, Ohio Board of Regents BRTTC #BRTT02-0003 and #ODOD-AGMT-TECH-04-049, and Sandia National Laboratories under Doc.No: 283793.

References

AFEWORK, A., BEYNON, M. D., BUSTAMANTE, F., DEMARZO, A., FERREIRA, R., MILLER, R., SILBERMAN, M., SALTZ, J., SUSSMAN, A., AND TSANG, H. 1998. Digital dynamic telepathology - the Virtual Microscope. In *Proceedings of the 1998 AMIA Annual Fall Symposium*, American Medical Informatics Association.

ARGE, L., TOMA, L., AND VITTER, J. S. 2000. I/o-efficient algorithms for problems on grid-based terrains. In *Proceedings of 2nd Workshop on Algorithm Engineering and Experimentation (ALENEX '00)*.

BAJAJ, C. L., PASCUCCI, V., THOMPSON, D., AND ZHANG, X. Y. 1999. Parallel accelerated isocontouring for out-of-core visualization. In *Proceedings of the 1999 IEEE Symposium on Parallel Visualization and Graphics*, 97–104.

BEN-ARIEH, D., GUTIN, G., PENN, M., YEO, A., AND ZVEROVICH, A. 2003. Transformations of generalized atsp into atsp. *Oper. Res. Lett. 31*, 3, 357–365.

BEYNON, M. D., KURC, T., CATALYUREK, U., CHANG, C., SUSSMAN, A., AND SALTZ, J. 2001. Distributed processing of very large datasets with DataCutter. *Parallel Computing 27*, 11 (Oct.), 1457–1478.

CHIANG, Y.-J., AND SILVA, C. 1999. External memory techniques for isosurface extraction in scientific visualization. In *External Memory Algorithms and Visualization*, J. Abello and J. Vitter, Eds., vol. 50. DIMACS Book Series, American Mathematical Society, 247–277.

CHOW, S. K., HAKOZAKI, H., PRICE, D. L., MACLEAN, N. A. B., DEERINCK, T. J., BOUWER, J. C., MARTONE, M. E., PELTIER, S. T., AND ELLISMAN, M. H. 2006. Automated microscopy system for mosaic acquisition and processing. *Journal of Microscopy 222*, 2 (May), 76–84.

CONTASSOT-VIVIER, S., AND MIGUET, S. 1999. A load-balanced algorithm for parallel digital image warping. *International Journal of Pattern Recognition and Artificial Intelligence 13*, 4, 445–463.

COX, M., AND ELLSWORTH, D. 1997. Application-controlled demand paging for out-of-core visualization. In *Proceedings of the 8th IEEE Visualization'97 Conference*.

FU, C.-W., WONG, T.-T., AND HENG, P.-A. 1998. Triangle-based view interpolation without depth-buffering. *Journal of Graphics Tools: JGT 3*, 4, 13–31.

GONZALEZ, R. C., AND WOODS, R. E. 2001. *Digital Image Processing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

HELSGAUN, K. 2000. An effective implementation of the Lin-Kernighan traveling salesman heuristic. *European J. Oper. Res. 126*, 1, 106–130.

HUANG JIANG, Y., MING CHANG, Z., AND YANG, X. 2004. A load-balanced parallel algorithm for 2d image warping. In *ISPA*, Springer, J. Cao, L. T. Yang, M. Guo, and F. C.-M. Lau, Eds., vol. 3358 of *Lecture Notes in Computer Science*, 735–745.

LAWLER, E. L., LENSTRA, J. K., KAN, A. H. G. R., AND SHMOYS, D. B., Eds. 1985. *The Traveling Salesman Problem*. Wiley.

LIN, S., AND KERNIGHAN, B. W. 1973. An effective heuristic algorithm for the traveling salesman problem. *Operations Research 21*, 498–516.

MARCATO, R. 1998. *Optimizing an Inverse Warper*. Master's thesis, Massachusetts Institute of Technology.

RUTT, B., KUMAR, V. S., PAN, T., KURC, T., CATALYUREK, U., WANG, Y., AND SALTZ, J. 2005. Distributed out-of-core preprocessing of very large microscopy images for efficient querying. IEEE International Conference on Cluster Computing.

UENG, S.-K., SIKORSKI, K., AND MA, K.-L. 1997. Out-of-core streamline visualization on large unstructured meshes. *IEEE Transactions on Visualization and Computer Graphics 3*, 4 (Dec.), 370–380.

WITTENBRINK, C. M., AND SOMANI, A. K. 1993. 2d and 3d optimal parallel image warping. In *International Parallel Processing Symposium*, 331–337.

WOLBERG, G. 1994. *Digital Image Warping*. IEEE Computer Society Press, Los Alamitos, CA, USA.