Parallelization of the Mosaic Image Alignment Algorithm

Laughton M. Stanley

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the Requirements for the Degrees of
Bachelor of Science in Electrical Science and Engineering and
Master of Engineering in Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology
May 26, 1999
[June 1999]

Author [illegible]
Department of Electrical Engineering and Computer Science
May 25, 1999

Certified by [illegible]
Seth Teller
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by [illegible]
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Parallelization of the Mosaic Image Alignment Algorithm

Laughton M. Stanley

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the Requirements for the Degrees of
Bachelor of Science in Electrical Science and Engineering and
Master of Engineering in Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology
May 26, 1999

Abstract

As processing power becomes cheaper and more readily available in networks of powerful desktop machines, the need for parallel algorithms capable of using these often untapped resources will increase. One application for parallel algorithms and implementations is found in the City Scanning Project at MIT's Laboratory for Computer Science. This research effort uses digital photographs to model the three dimensional geometry and textures of urban scenes. One element of this process is the accurate estimate of the orientation of groups of photos taken from a common optical center, known as spherical mosaicing. This process, which is both time and memory intensive, is a natural application for parallel programming techniques. This thesis describes improvements to spherical mosaicing using the Message Passing Interface (MPI) to create a parallel algorithm which may be run across multiple machines on a common network.

The improvements make appreciable gains in several areas. They substantially reduce the wall-clock time of the calculation, while simultaneously decreasing the amount of memory required on any one machine. This in turn makes possible the execution of the program on full resolution data, which had not been possible with a serial algorithm.

Thesis Supervisor: Seth Teller
Title: Associate Professor of Electrical Engineering and Computer Science

Acknowledgements

I would like to thank Professor Teller for being my thesis advisor and for all of the direction and advice that he provided during the course of my work. The encouragement that he provided was instrumental in helping me to set high goals for this project.

I would also like to thank Neel Master for guiding me through the mosaic code and for all of the times that he answered my questions or took on extra work so that we might solve some new problem. For his advice in parallel structure and all of the work he put into the porting of code to Linux, I would like to thank Qixiang Sun. For their technical advice when I was stumped, I thank David Korka, Calvin Lin, and Jon Zalesky. My gratitude also goes out to Elaine Chin for her help in editing my work.

Finally I would like to thank my parents for supporting me during my years at MIT.

Contents

List of Figures

1 Introduction

The problem of resculpting a serial algorithm to run in parallel is a problem which is considered often in an age where the availability of computational power in the form of multiple networked desktop machines is becoming increasingly frequent. This thesis describes the parallelization of one such program, the spherical mosaicing algorithm, which is used by the City Scanning Project at MIT's Laboratory for Computer Science. The City Scanning Project is described briefly in this section. The spherical mosaicing algorithm and the parallel version of the algorithm are covered in depth in later sections.

The City Scanning Project is a research effort of the Graphics Group at MIT's Laboratory for Computer Science. The broad goal of the project is to create a method for rapidly acquiring computer models of three-dimensional (3-D) urban environments. This representation could be used to create a virtual environment for numerous applications including games, virtual tours, and urban planning [Coo98]. Research is currently underway to create a fully automated process which uses images from an urban setting to produce a 3-D CAD model of the scene. This model includes not only geometric information, such as the shapes, locations, and orientations of building facades, but also textural information which is needed to replicate the look of the buildings.

1.1 Camera Apparatus

The process begins with the acquisition of digital photographs from the urban scene. For a given scene, a set of locations (nodes) are selected which provide good views of the buildings of interest from all sides. This insures that later stages of the algorithm will have enough information to reconstruct the building from each side. A visualization of the nodes in the Tech Square data set (81 nodes total) is shown in Figure 1.

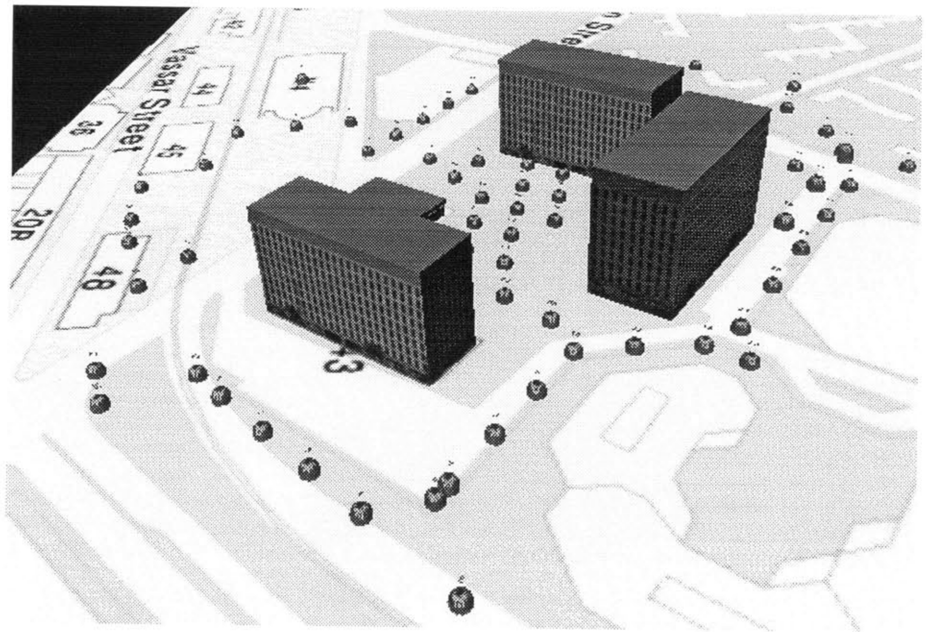

Figure 1: Node locations for Tech Square Data Set

At each node the sensor obtains a hemispherical view of the scene. Since the digital camera which acquires the images has a limited field of view, pictures are taken in an overlapping pattern, called a tiling, created by rotating the camera to face all directions. This rotation is performed about the camera's optical center in order to avoid parallax distortion. When each picture is taken, the node location and camera orientation are recorded and stored along with the digital image. [DeC98]

Taking all of these pictures is a huge undertaking. To aid in automating the process, the City Scanning Project uses a camera apparatus known as Argus. Shown in Figure 2, Argus is a bit more sophisticated than your average camera tripod. To accurately estimate the position of each node, it uses a Global Positioning System (GPS) and an Inertial Navigation System (INS). During the photographing of a node, Argus keeps track of the orientation of the camera and records this information with the digital photographs.

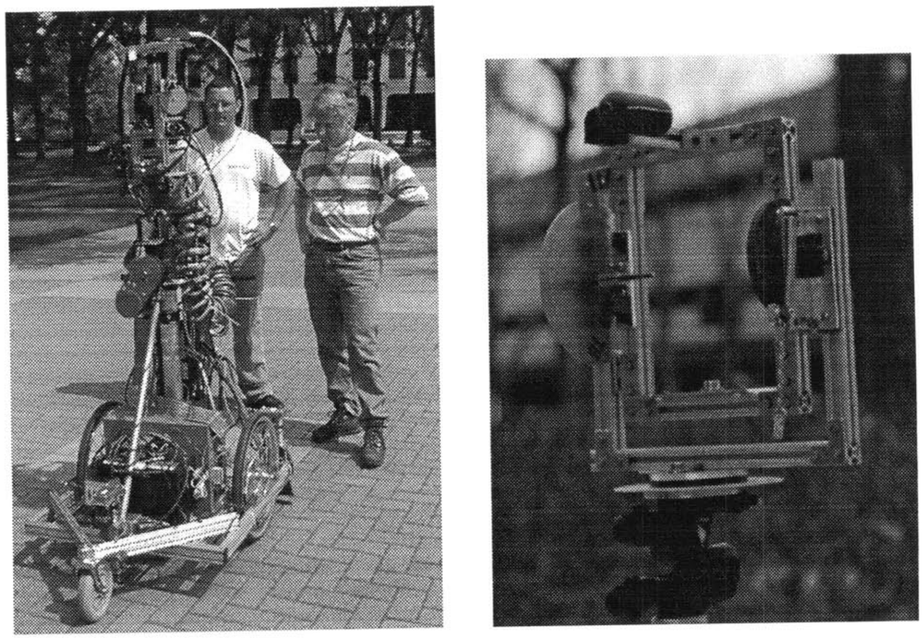
Figure 2: A view of Argus and close up of pan tilt head.

1.2 Spherical Picture Tiling

The images taken at each node are arranged in a spherical tiling pattern which catches all useful views. While photographing a node, the camera has two rotational degrees of freedom. One is to face any horizontal direction, shown in the left side of Figure 3, and the other is to face various vertical directions, shown in the right side of Figure 3. For any given vertical inclination, the camera pivots completely around the tripod's vertical axis to produce a ring of pictures. For inclinations of 40° South to 40° North the camera takes pictures at 30° intervals, for a total of 12, as illustrated in the left half of Figure 3. At the 60° inclination, 9 images are taken (spaced 40° apart), and at the 80° inclination two images are taken (spaced 180° apart).

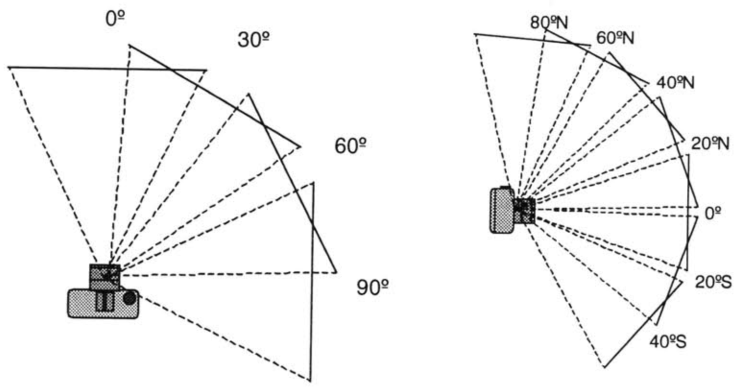

Figure 3: Directions of rotation of the camera while photographing a node.

A representation of the resulting spherical pattern is shown in Figure 4. Typical patterns use either 47 or 71 images. A 47 image node is created by using inclinations of 0° and above. A 71 image node, used when a node is well above the ground, is created by using all inclinations between 40° South and 80° North. When combined, these image tiles produce the view from a given node.

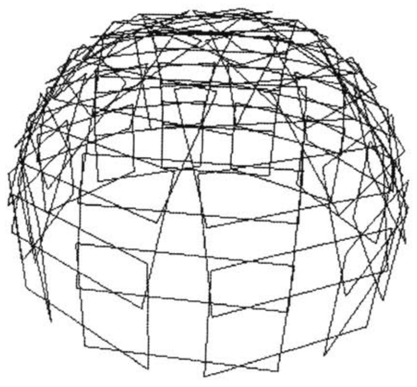

Figure 4: Spherical tiling pattern

1.3 Mosaic of Combined Images

Spherical mosaicing is the process of aligning and combining all of the images from a node. This process is performed once the image and pose data has been uploaded from Argus. In order to combine the images accurately, estimates must be made of their position and of the internal parameters of the camera. The information recorded by Argus is accurate to within about one degree, however 3-D reconstruction algorithms require accuracy one to two orders of magnitude greater [Coo98]. To provide this level of precision we use the pose refinement algorithm which is the subject of this thesis.

The spherical mosaicing algorithm makes use of the data contained within the images to align and orient them with greater accuracy. This is accomplished through an iterative global optimization algorithm which refines orientation and camera internal parameter information by considering overlapping sections of images. The algorithm uses gradient descent methods to better align overlapping sections, and judges the goodness of the fit by the squared difference in luminosities between matching pixels. The end result is a pose estimate for each image from which a hemispherical representation may then be generated, as shown in Figure 5.

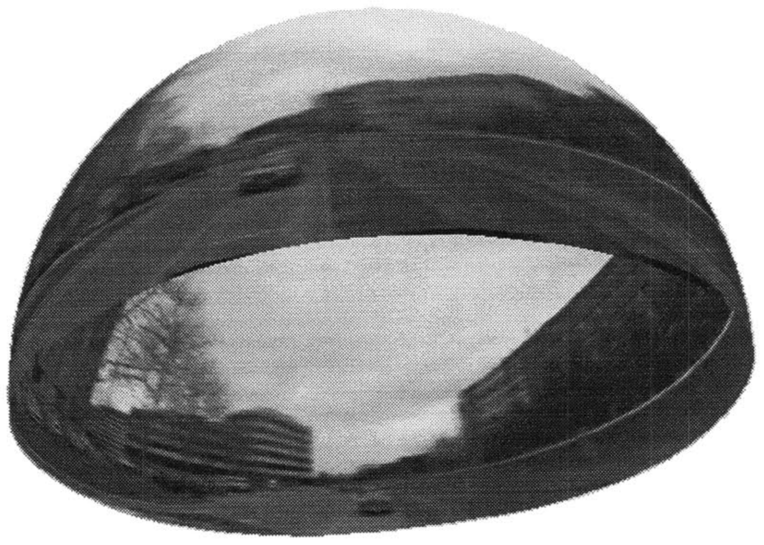

Figure 5: Hemi-spherical panorama generated by the spherical mosaicing algorithm.

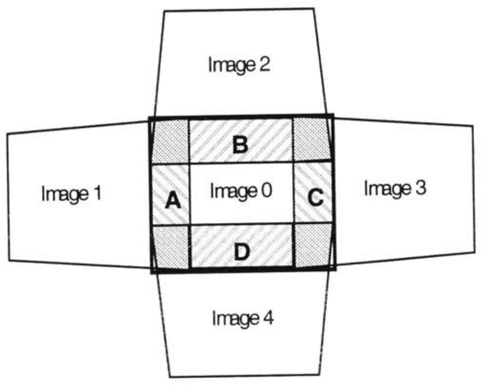

Figure 6: Illustration of an image (image 0) and its overlaps (A, B, C, D) with images 1, 2, 3, and 4.

This process is time consuming number of reasons. The algorithm requires the execution of operations on a large number of image pairings. For example, in a tiling of 47 images, there are 98 pairwise overlaps. (An illustration of pairwise overlaps is shown in Figure 6.) Calculations must be performed on each of these sections twice, creating in effect 196 overlapping sections. For each iteration of the algorithm three operations must be performed on this set of overlaps, and a typical run of the algorithm on half size images can easily require 20 iterations. The result (196x3x20 = 11,760 pairings which need to be considered) is a very large number of computations, considering that each full size image is 1012x1524 pixels. On a single Pentium II 450, this calculation takes approximately 2 hours for a half size image. Attempts at running full size nodes of 47 images with the serial algorithm indicate that times of 90 or more hours are not unreasonable due to the intense amount of virtual memory swapping required. The numbers for a 71 image node would be much larger. A further reason that the algorithm is time consuming is that the image files are large enough to exceed the available memory of most machines. For instance, a node of 47 half size images requires 400 Mb of memory. While it is possible to use disk space as virtual memory, this will slow down the speed of calculations.

The parallel algorithm which is the subject of this thesis alters the spherical mosaicing algorithm so that its operations may be split amongst multiple machines. This reduces the computational requirements for each machine, and allows responsibilities to be divided such that fewer images must be loaded on an individual machine. As a result, this algorithm saves time and uses less memory.

1.4 Node Exterior Orientation

Once all of the node images have been refined for pose, the problem of estimating the necessary parameters for each image is reduced to one of estimating only the location and relative orientation of each node. The city project uses global optimization to refine these estimates based on correspondences between image features. For example, an easily distinguishable feature such as the corner of a building might appear in several nodes and provide a method for aligning them relative to each other. [Coo98]

1.5 Creating the 3D Model

With all images accurately described, the 3D model may now be extracted. Through the detection of horizontal and vertical edges, and other methods, an algorithm extracts the 3D geometry of the scene, creating a polyhedral model. The textures for these surfaces are then estimated using median statistics based on views from multiple nodes. The end result is a 3D CAD model which can be rendered from any viewpoint, as shown in Figure 7.

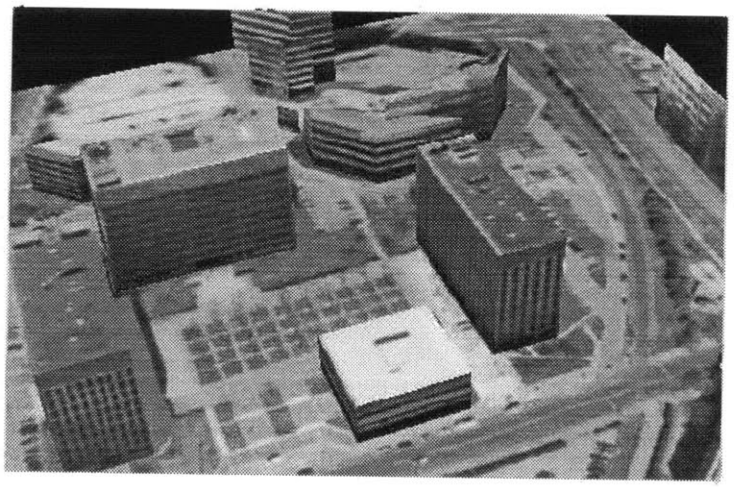

Figure 7: 3-D CAD rendering of Tech Square produced by the City Scanning Project.

2 Spherical Mosaicing

Once the images have been collected, the image pose information is refined so that the images may be used by 3-D reconstruction algorithms. These algorithms require very accurate estimates of both the node's position, and of the relative orientation of the images. In addition, the parameters of the camera model must be correct, as they affect the estimate of the projection of 3-D reality onto the image.

The optimization of all of these parameters together can be quite daunting. At the minimum there are three parameters to describe the camera, three parameters to describe the node location, and three to describe the relative orientation of each image. Instead of trying to cope with all of these at once, spherical mosaicing addresses the optimization of the camera and rotation parameters, leaving the estimation of absolute node position and orientation to a later step in the process. This is possible because the grouping of images into spherical mosaics allows them to be treated as rigid bodies.

This limiting of parameters is possible because the orientation and camera parameters can be reliably estimated from correspondences between images with no information about node placement. As a result of temporarily disregarding placement, the optimization complexity is reduced by a factor of fifty. [Coo98]

2.1 Parameters Optimized

Spherical mosaicing addresses two key sets of parameters: image orientation and camera internal parameters. Both sets are key to later stages of the algorithm which rely on the seamless, well-aligned hemispherical images that are created by the optimization process.

Orientation refers to the way in which a particular image is facing with respect to a base image. Orientation is different from position, in that the orientation specifies the rotation of an image and does not imply or specify any location in space. There are several possibilities for the representation of orientation. The most compact one is the Euler representation, which is a 1x3 vector. One manifestation of this form is the roll, pitch and yaw framework from aeronautics, show in Figure 8.

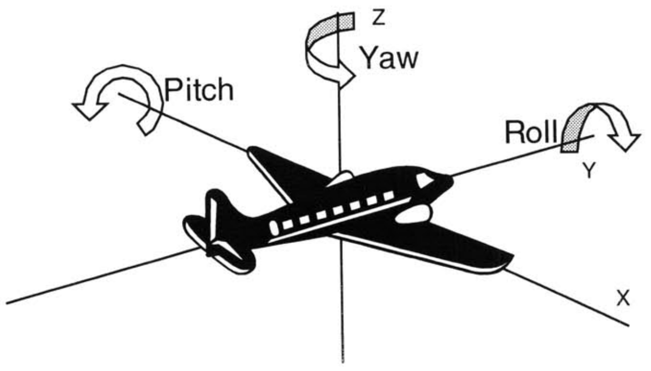

Figure 8: Illustration of roll, pitch, and yaw.

An equivalent Euler framework is a 1x3 vector, ([α, β, γ]), defined as follows:

γ = rotation about the vertical, or Z axis

β = rotation about the Y axis

α = rotation about the Z axis displaced through β

Although compact, the Euler representation has singularities (the case of no rotation, for instance) which can disrupt optimization routines. The preferred representation is the quaternion, a 1x4 vector, because it has no singularities. The quaternion is difficult to illustrate pictorially, however it may be formed from the α,β,γ Euler representation as follows [VAN78]:

$$A = \sin\left(\beta/2\right)\sin\left(\frac{\alpha-\gamma}{2}\right)$$
$$B = \sin\left(\beta/2\right)\cos\left(\frac{\alpha-\gamma}{2}\right)$$
$$C = \cos\left(\beta/2\right)\sin\left(\frac{\alpha+\gamma}{2}\right)$$
$$D = \cos\left(\beta/2\right)\cos\left(\frac{\alpha+\gamma}{2}\right)$$

The spherical mosaicing process makes use of these quaternion representations of rotations: both the initial rotation estimates and the refined estimates are represented this way.

The other set of parameters is for the camera model. This consists of three values: the focal length, and the x and y coordinates of the image center.

2.2 Global Optimization

The global optimization algorithm seeks to minimize a global objective function formed from the sum of the squared differences in luminosities between overlapping pixels. Specifically:

$$O = \sum_{i,j \text{ adjacent}} \left[\sum_{x_i, y_i} \{L_i(x_i, y_i) - L_j(P_{ij}(x_i, y_i))\}^2 + \sum_{x_j, y_j} \{L_j(x_j, y_j) - L_i(P_{ji}(x_j, y_j))\}^2 \right]$$

$L_i(x_i, y_i)$ is the luminance value of a pixel at the given coordinates (x,y) in image i, and P_{ij} is the projection of the supplied image i coordinates onto image j. The projection calculation computes which pixel on the adjacent image overlaps the current pixel being calculated. The objective function considers both permutations of each pair of adjacent images (i.e. it considers the intersection of image i with image j and the intersection of image j with image i for each pair of adjacent images). Thus for a 47 image node with 98 pairs of adjacent images, the objective function considers all 196 image pairs which can be made from the 98 pairings. For each overlapping section that it considers, it takes the sum of the squared differences between each pixel in the base image and the corresponding pixel (established by the P matrix). Thus for an overlapping section of 100 pixels by 100 pixels, approximately 10,000 squared differences will be summed. To avoid unnecessarily searching through pixels in sections where the images do not overlap, bounding boxes are created around the search area so that only pixels likely to overlap with the other image are considered. It should be noted that one potential minimization, that of adjusting image pairs so that they do not overlap (and thus have no overlapping pixels to create errors), could be achieved by the optimization, however it has not done so in practice [Coo98].

The optimization routine adjusts both the rotation parameters and the camera internal parameters in order to minimize the objective function. Both adjustments are made by following partial derivatives with respect to the parameters. Here I give a summary of some of the key mathematical concepts as well as some intuition. For a more detailed derivation of the math, please see Shum, Han and Szeliski [TCM98].

2.2.1 Rotation Optimization

The objective function with respect to rotations is minimized by computing derivatives with respect to the orientations and then performing a Levenberg-Marquardt (LM) non-linear optimization based on the initial orientation estimates. For an error term ($e_{x,y}$) formed by the difference of two pixel luminosities squared, the derivative with respect to the quaternion **q** can be shown to be[1]:

[1] For a derivation of this derivative, see section 3.4.1 of Satyan Coorg's thesis [Coo98].

$$\frac{\partial e_{x,y}}{\partial \mathbf{q}} = \frac{\partial L'}{\partial x''}\frac{\partial x''}{\partial \mathbf{q}} + \frac{\partial L'}{\partial y''}\frac{\partial y''}{\partial \mathbf{q}}$$

where $e_{x,y}^2 = \left(L(x,y) - L'(x'',y'')\right)$ and $\mathbf{q} = (a,b,c,d)$

The optimization calculates a gradient to be associated with each quaternion. This is done by summing over all error terms which depend on $\mathbf{q}_i$:

$$\mathbf{G}_i = \sum_{x,y} e_{x,y} \frac{\partial e_{x,y}}{\partial \mathbf{q}_i}$$

The negative of this gradient indicates the direction of rotation that each image should undergo to obtain the greatest reduction in its overall squared error. At the same time a Hessian term corresponding to each image pair is calculated:

$$\mathbf{H}_{ij} = \sum_{x,y} \frac{\partial e_{x,y}}{\partial \mathbf{q}_i} \left(\frac{\partial e_{x,y}}{\partial \mathbf{q}_j} \right)^T$$

The Hessian represents error derivatives with respect to each pair of quaternion "directions." For the purposes of spherical mosaicing, the gradients and Hessians are concatenated into single global variables: one of length 1x4n for the gradients, **G**, and one of size 4nx4n for the Hessians, **H**. Note that the Hessian matrix is quite sparse. In the example of a 47 image node, no one image overlaps more than 5 other images, so each row or column in **H** would contain no more than 20 values and no fewer than 168 zeros. Using the current quaternions, **Q**, as well as **G**, and **H**, a linear solver can solve for the changes to each $\mathbf{q}_i$ while enforcing the constraint that quaternions must be unit vectors. These changes are computed until the objective function changes by less than a defined fraction. This fraction was set to one part in a thousand (0.001) during my work.

2.2.2 Internal Parameter Optimization

The camera parameters- focal length, horizontal center, and vertical center- must be estimated as well. Although the camera is calibrated off line, variation or misestimation of these parameters could cause misalignments of images. By estimating the camera parameters from the images themselves, spherical mosaicing avoids the problem of working with inaccurate parameters. The algorithm uses methods similar to those used for the rotations to compute derivatives of the error term with respect to the parameters. It uses these in a calibration technique to determine the optimal values for the parameters.

2.3 Global Optimization Architecture and Implementation

The previous section described some of the math and theory behind the global optimization. Here I explain it with respect to chronology and the flow of information.

The master function for the optimization routine is "updateAll." It defines the sequence of execution of the algorithm. In a typical run of updateAll the algorithm first calls updateInternalParameters which revises the estimated camera parameters and then calls updateAllRotations which performs the rotation optimization described in the last section. Finally it calls computeCorrelationInternal function to determine the new objective function value and checks this value against the previous one. If the stop criterion is met, then the loop terminates; if not, the loop is repeated.

The actual implementation of the optimization algorithm is more complicated than the one described in the theory section. To obtain better initial refinements, the algorithm begins by using bandpass filtered versions of the image rather than the luminances. This helps to obscure high frequency information which is not initially important and might distract the algorithm. The updateAll function loops on the bandpass images until the objective function changes by less than 100 times the stop criterion (100*0.001 = 10%). Once the bandpass loop is finished, a new loop is started on the regular images. This loop finishes when the objective function changes by less than the stop criterion (0.1%). The algorithm does not process some images which do not have sufficient texture to be matched with gradient descent.

These images are selected by comparing image high frequency energy to a threshold and rejecting those which have too little high frequency information. Once the normal image loop has finished, the algorithm runs a loop considering only these previously rejected images. This loop skips the internal parameter update, but still runs the rotation update and objective function updates. The rejected image loop stops when a more relaxed stop criterion, 10% (100 times the luminance stop criterion), is met. The sequence of these loops is summarized in Table 1.

Loop #	Data Used	Loop Sequence
1	Bandpass	updateInternalParameters updateAllRotations update objective function stop if less than 10% improvement
2	Luminance	updateInternalParameters updateAllRotations update objective function stop if less than 0.1% improvement
3	Rejected Luminance	updateAllRotations update objective function stop if less than 10% improvement

Table 1: Sequence of operations for implementation of global optimization in spherical mosaicing.

2.3.1 Updating Internal Parameters

The function updateInternalParameters updates the camera parameters by taking into account all pairs of images which are not rejected due to lack of texture. For each pair of images (196 for a node of 47 images), it calls the function computeCorrelationInternal. This function computes a 1x3 gradient and a 3x3 Hessian matrix for a given image pair. This gradient and Hessian contain the accumulation (over the image overlap) of the partial derivatives of the pixel errors with respect to the three camera parameters. The gradient and Hessian produced by computeCorrelationInternal are summed over all of the valid image pairs. Once this summation is finished, updateInternalParameters updates the internal parameter estimates for the entire image set. Note that since the results of computeCorrelationInternal are summed over all images before any parameter updates are made, the order in which the image pairs are considered is not important, nor is it important that they be done in series.

Function	updateInternalParameters
Requires	image pixel data image rotation quaternions camera internal parameters
Produces	updated camera internal parameters new objective function value

Table 2: Input/output for updateInternalParameters.

2.3.2 Updating Rotations

The function updateAllRotations is similar in form to updateInternalParameters. The function creates a 1x4n derivative vector and a 4nx4n Hessian matrix to collect the values generated during the rotation optimization. It then calls the function computeCorrelationRotations for all valid image pairs. The set of valid image pairs changes depending on where the function is called in the overall operation. As may be seen from Table 1, the function will use non-rejected images during loops 1 and 2, and rejected images during loop 3. The computeCorrelationRotations routine requires camera parameters, image rotations, and image data to compute the error gradients and Hessians with respect to the quaternions. Within the function, error gradients and Hessian blocks are summed over all overlapping pixels. The result of these summations is returned to updateAllRotations which then stores them in the appropriate places in the total gradient and total Hessian variables. When the information from all valid image pairs has been collected, the quaternions for all valid images are updated using the method described in 2.2.1. Note that these operations are insensitive to the order in which the images are considered, and that the images need not be considered in series.

Function	updateAllRotations
Requires	image pixel data image rotation quaternions camera internal parameters
Produces	updated image rotation quaternions new objective function value

Table 3: Input/output for updateAllRotations.

2.3.3 Evaluating Objective Function

The final step in the optimization loop is to create an updated objective function value. When this stage is reached, the last valid objective function value was computed before the image rotations were updated and generally (except in the case of rejected images) before the camera internal parameters were updated. The algorithm ignores the objective function value computed during the rotation update since it falls in the middle of two update processes. To update the objective function, the computeCorrelationInternal function is called for all valid image pairs. The 1x3 gradient and 3x3 Hessian are ignored this time, and the algorithm considers only the sum of the squared errors which is returned by computeCorrelationInternal[2]. This objective function value from each image pair is summed over all valid image pairs to produce an objective function value for the entire node. This value is compared to a threshold according to the following formula:

$$\text{Finish if } \frac{\text{last value - new value}}{\text{last value}} < \text{threshold.}$$

Note that, as before, the objective function value is insensitive to the order in which the images are considered and does not require that they be considered in series.

Function	update objective function value (part of updateAll)
Requires	image pixel data image rotation quaternions camera internal parameters
Produces	new objective function value

Table 4: Input/output for the update of the objective function value.

2.4 Challenges

If one counts the number of operations required to execute just one loop of the optimization, it will quickly become obvious that there are a large number. To reiterate, a full sized image is 1012x1524 (or 1.5 megapixels). A node may have as few as 47 images, but some have as many as 71. Each loop must consider every overlapping pair of most of these images three times. Each time calculations must be performed on every overlapping pixel. The optimization must go through numerous iterations of the loop before the final criterion is met. It is common for the algorithm to require 30 iterations for half size images, and even more for full size images. The bottom line is that the optimization requires large numbers of calculations and requires large amounts of memory.

To put some perspective on the problem, I ran the optimization on a single node on a Pentium II 450 MHz processor. I considered several different image sizes to give some perspective of how the execution time grows. The results are shown in Figure 9.

[2] Another possibility for improvement of the algorithm is to update the camera parameters at the same time as the objective function value is updated since both tasks call the same function.

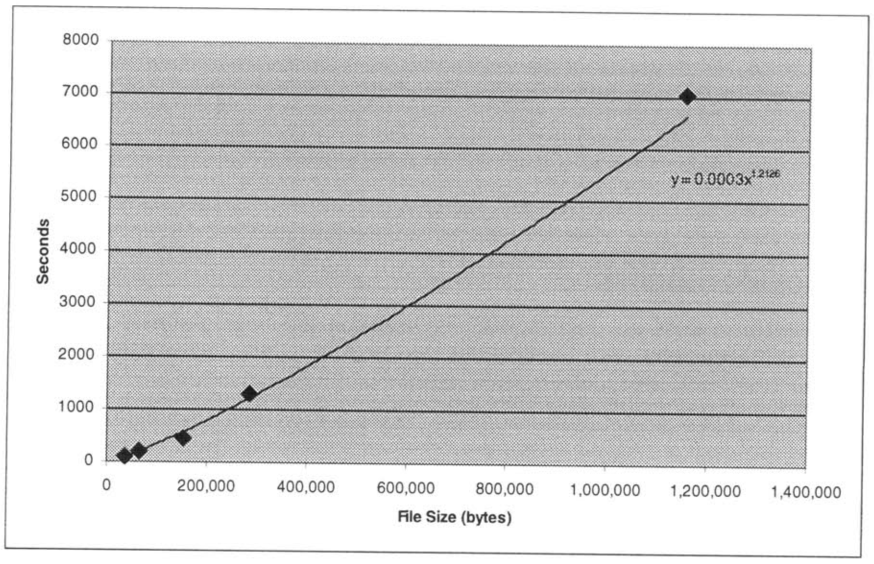

Figure 9: Plot of execution times at several image resolutions.

The plot shows that the execution times can become substantial. The execution time for the quarter size images (~289K) was nearly 22 minutes, and the time for the half size images was nearly 2 hours. Since the machine being used has ample memory for all of the image resolutions in the chart, memory swapping was minimal and the data reflect almost entirely computation time alone. The graph shows a least squares regression fit trend line which suggests that the execution time is somewhat more than linear with image size. By this model, the execution time for full size images (~4.6 Mb) would be on the order of 10 hours. However there is good reason to suspect that it will be even worse than this. The images require a large amount of memory. Not only do all of the image pixels themselves have to be stored, but the associated bandpass images and derivatives must be stored as well. Empirical results show that a node of 47 half size images requires 400 Mb of memory space to be optimized. Extrapolating from this, a similar full size node would require 1600Mb of memory, and a 71 image node could require 2400 Mb or more. Since these latter figures easily exceed the RAM on most machines where the optimization is likely to be run, the algorithm would have to rely heavily on virtual memory space. The amount of time required by the resulting memory swaps would increase the total execution time substantially.

2.5 The Case for Parallelization

The spherical mosaicing algorithm does a good job of optimizing the camera and orientation parameters for a node of images. However it is quite costly in terms of the amount of time required to process a node. This should be considered in the context of the goals of the city project. Currently the City Scanning Project has fully processed the data from an 81 node set into a 3-D CAD model. However this is only the initial data set. The group is in the process of perfecting a process that can be used to take in data from whole neighborhoods and cities. The immediate goal of the project is to model the entire MIT campus. The number of nodes required to do this is considerably more than those collected already. Processing these large numbers of nodes will quickly add up to weeks or more of computer time and wall-clock time.

One way to address this is to change the serial spherical mosaicing algorithm to run in parallel. There is potential for this because the optimization routines that update camera parameters, update rotation parameters, and update the objective function value all rely on summations of operations which are independent. These operations can be run simultaneously across multiple machines to create a substantial speed up. Apart from speed alone, another advantage of a parallel algorithm is that it could split the memory load among the machines being used. This could potentially bring the amount of memory required per machine to a level that would lead to little if any memory swapping. For these reasons I created the parallel algorithm which is the subject of this thesis.

3 Methods of Parallelism

Several decisions must be made when recasting a serial program as a parallel one. One consideration is the platforms on which the program will be run. Others include the ease of programming, the structure of the execution environment, and the portability of the finished code from platform to platform. Sections 3.1 and 3.2 review some of the options available, including their good points and drawbacks.

3.1 Application requirement considerations

I reviewed several factors when picking the appropriate method for writing the parallel algorithm. The first was the platforms on which the algorithm might run. Currently the graphics group makes significant use of both SGI and Intel architectures. There is also some use of parallel processing SUN machines. Although the parallel algorithm would likely be used primarily on only one of these systems, having the ability to port the program to others would be helpful. In this spirit, the ease with which this porting could be accomplished is also an issue. Methods that rely heavily on platform specific code would obviously not be good choices in this respect.

One important consideration peculiar to parallel frameworks is the hardware configurations that they support. There are several conceivable configurations for parallel processing. The most obvious is that of a single machine with multiple processors which share a memory environment. There are several of such machines available through the graphics lab, so this was a significant consideration. Indeed, the City Scanning project currently supports a parallel version of spherical mosaicing which works within this framework [Sun99]. A second fundamental framework is that of several single or dual processor machines which work together through a network. One notable example of this strategy are the RSA code breaking schemes used in recent contests. These designs assigned sequences of numbers to volunteer machines on the internet and were able to decrease the time needed to crack the sequence. Other frameworks might include combinations of single and multi-processor machines, or combinations of machines of multiple architectures. All of these possibilities could apply to machines available to the City Scanning Project.

3.2 Available implementations

There are several methods of creating parallel programs. One of the more common is by using threads. These integrate easily into a C++ code environment. Threads can be used to spawn multiple tasks on a machine which, provided that the machine has more than one processor, can then be executed in parallel. Since all of these tasks share a single memory space, starting them is quite straightforward. They are also quite portable to a large number of platforms. The limitation of threads is that they use a single memory space, and thus are not useful in situations which make use of multiple machines which, by necessity, have separate memory spaces.

Other possibilities included tools under development at MIT's Laboratory for Computer Science, including Cilk and Parallel Matlab. Although these are interesting choices, their drawbacks are potentially significant. CILK supports only ANSI C, so the C++ code base for spherical mosaicing would have to be modified and restructured. In the case of Matlab, the environment makes development very easy, but it would require a significant reworking of the original code. We decided to abandon these choices in favor of tools which are more widely distributed and are more certain to be available on a large number of platforms.

The Message Passing Interface (MPI), a freely available package created through a government sponsored consortium, has several nice features. It is available for most architectures, including SGI, SUN and Intel (Linux/Windows). MPI integrates well with existing C++ code, by providing libraries of functions for the passing of messages between processes and the management of a multiple process environment. The package is very versatile; it allows for the use of most imaginable parallel configurations. For instance it allows for the running of multiple processes on a multi-processor machine, the running of multiple processes across multiple networked machines, and the use of multiple architectures simultaneously. I chose MPI as my parallel method for the spherical mosaicing algorithm because its versatility will make it useful to the City Scanning Project through any future changes in hardware.

4 Implementation

4.1 Opportunities for parallel operations in Mosaic

When looking to parallelize the spherical mosaicing process, we initially considered several alternatives. The algorithm does several operations which are potentially time consuming. When the process begins, a bandpass version is created through convolution by an 81 element filter. During the calculation of Hessians and gradients thousands of independent partial derivative calculations must be made. For each round of rotation, camera parameter, or objective function updates, over 100 pairings of images must be considered. Once the Hessians and gradients have been calculated during the rotation update, the nonlinear optimization applied to calculate the new quaternion involves decomposing a Hessian matrix of 40,000 terms or more.

Each of these operations has the potential to be parallelized. Any two or more independent operations may be performed simultaneously, thus it is possible to filter several images at the same time, to calculate numerous partial derivative terms at once, and to work on multiple pairings at the same time. It is also possible to rework some matrix operations, such as the rotation optimization, into parallel operations. Hence from a feasibility standpoint, there are many potential operations to attack.

The key to picking operations to optimize in parallel is to consider the potential gains that could be achieved by speeding them up. A good first approximation to make is to consider what percentage of the total time is spent on any one operation, and then calculate the improvement that could be made if that operation took zero time. For instance, if the nonlinear optimization calculations took 25% of the total time, then the potential gain in speed by parallelizing the operation is a 33% gain in speed. This approximation is a good pointer to the best opportunities for improvement, but it makes two assumptions which are not valid. One assumption is that there is no overhead associated with parallelization. In reality, the creation of a parallel program has some overhead associated with it, and each parallel operation performed has overhead as well. One of the primary causes of this is the need to communicate between processes. A second implied assumption is that infinitely many processors can be devoted to the parallel task. In reality, the computation resources at hand are not infinite, and the problem may not be infinitely divisible. For instance, if one chose to divide up the image pairings among processors, 200 processors would exhaust the image pair combinations available and any further processors would be unable to share the work load.

While we considered the possibility of parallelizing such repeated functions as the nonlinear optimization of the rotation quaternions, some data from the spherical mosaicing process helped focus the search. The results of timing the functions in the program are shown in Table 5. From this data, one obvious path to investigate is a parallelization of computing the rotation updates. This could be done at several levels, and here other considerations come into play. Since parallel processing relies on communication between processes for each piece of work, there will be overhead associated with each operation. In the case of this algorithm, the amount of overhead is of a reasonably fixed size, so the optimal amount of work to be given to a processor at one time is large. For this reason, it made the most sense to parallelize the calls to the function computeCorrelationRotations. Since the function is called for each valid image pair during an iteration, there are ample operations to split amongst processors. At the same time the amount of work represented by one call is substantial. A further selling point is the compactness of the function. Making parallel operations out of single functions can reduce the amount of the programmer's work considerably.

Operation	Percent of total run time
computeCorrelationInternal calls while updating internal camera parameters	20%
computeCorrelationRotations calls while updating quaternions	67%
computeCorrelationInternal calls while updating the objective function value	6.4%
Total of all three	93%

Table 5: Percent of total run time used by selected functions.

Parallelizing computeCorrelationRotations has the theoretical potential for speeding up the calculations by 200%. However if one were to parallelize computeCorrelationInternal as well, the total speed up could be as much as 1300%, since only 7% of the total calculation load would be unparallelized. Obviously the overhead limitations on the amount of hardware used make realization of the 1300% unreasonable, but the numbers do indicate that this is potentially a very nice parallel problem, since such a large portion of the work can be parallelized by reworking calls to only two functions.

4.2 Parallel Algorithm Structure

There are multiple ways in which a parallel framework can be formed. MPI imposes some structural restrictions on how processors interact, but even then there is still some freedom available to design a means of coordinating the processors. Before implementing parallel spherical mosaicing, I considered these possibilities briefly.

MPI works by establishing links between the computers being used, and using those links for control. It is possible to create links with several protocols. In the case of my implementation, the protocol is remote shell (rsh). Once these links have been established, MPI runs a copy of the program on each machine. The code used from machine to machine does not vary (except in the case of multiple architectures), so each process is nearly identical when it begins. The only difference is that each process has a unique identifier number. These numbers are sequential starting at zero, so a set of 6 processors would have the numbers 0 to 5. To get the processes to behave differently, one has simply to write code conditioned on the process number.

MPI (Message Passing Interface) is so named because it is built on messages that processors send to each other. Messages can vary in size, destination, and data type as well as in other more technical ways. A typical message might be as follows: a vector of 96 floating point numbers being sent from process 0 to process 4. In the context of spherical mosaicing, this message could contain information on which operation to perform (such as computeCorrelationRotations), what data to use (images 12 and 15 for example), and parameter values to be stored in the processes local memory space. MPI also provides for messages which originate from one process and go to all processes.

It may be possible to create any number of hierarchical systems to perform parallel computations, however the simplest is the farmer-worker model. Under this model, there is one central process, the farmer, which is responsible for coordinating the efforts of the whole processing group. The rest of the processes in the group are worker processes which perform the work assigned them by the farmer. The only communication within the group is between the farmer and individual processes, as shown in Figure 10. Individual workers do no talk amongst themselves.

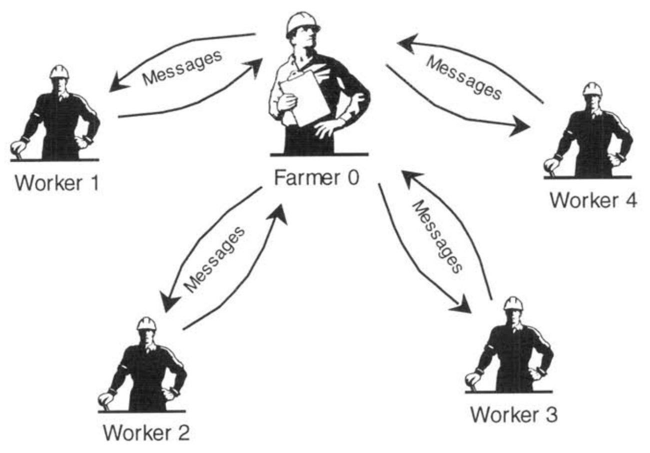

Figure 10: Message routing in the farmer-worker model.

The farmer-worker model is convenient to program because the farmer process has full control over the execution of the available work, and thus is aware of all ongoing operations. This centralization makes it much easier to follow from the standpoint of the programmer. Only one set of control code need be written. The code for the worker processes is straight-forward as a result. Worker processes simply need to receive messages, interpret them and execute the included instructions. In this framework, it is possible to have the farmer process double as a worker process as well. The risk of giving work to the farmer is that it will not always be ready to assign work and receive work from the workers. For this reason, I chose not to assign work to the farmer, even though it means that the farmer may not be used as fully as possible.

4.3 Changing a function to run in Parallel

When converting serial programs to parallel ones, some restructuring of the program function calls is necessary. In the case of spherical mosaicing, since calls to specific existing functions are parallelized, the changes necessary for conversion are easily illustrated. Figure 11 shows the communication paths between a master function and one call of a sub-function. In the case of spherical mosaicing, this master function could be updateAllRotations and the sub-function could be computeCorrelationRotations. The master function sends the sub-function some information at the time of execution. The information can include instructions for execution, and data needed for execution. The data can be passed either directly, or as a pointer to data in the common memory space. The sub-function returns data, which again can be either direct or by modifying the data pointed to by a pointer.

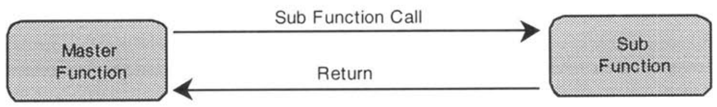

Figure 11: Typical function/sub-function structure for a serial process.

When this call is modified to work as a parallel call for processes in separate memory spaces, some changes must be made. Within the MPI parallel framework, a process on another computer cannot simply be called. Instead a message must be created, sent to the worker process, and interpreted. A further complication is that pointers cannot be passed between machines with separate memory spaces. Instead, any necessary data not available to the worker must be collected and passed across in the message as well. The same complications occur on the return path.

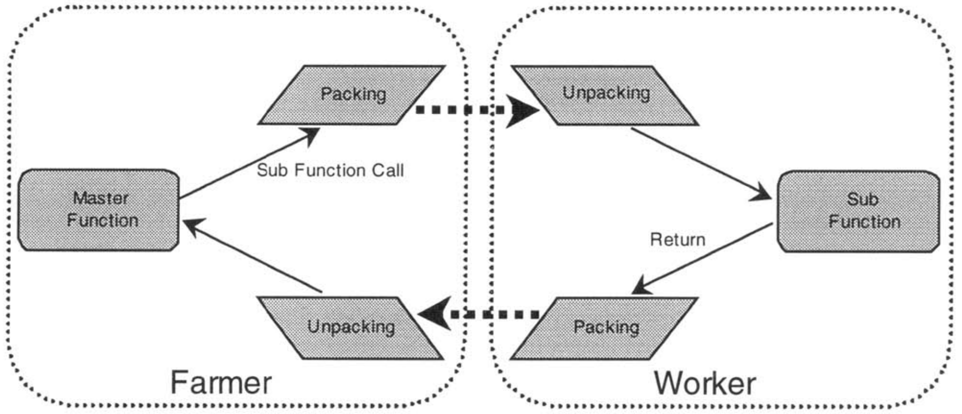

Figure 12: Function/sub-function structure for a parallel process with no shared memory space.

To work around the added requirements of a parallel framework, one can create intermediate processes which handle the specifics of parallelism. This is convenient because the master function and sub-function can remain essentially unchanged since the added steps are hidden behind a barrier of abstraction. These added steps are shown in Figure 12. The packing functions handle the collection of all necessary data and instructions for the destination process and the sending of the message. The unpacking functions receive the message, unpack the data into the recipient's memory space, and either execute the function call (in the case of the worker) or supply the returned value (in the case of the farmer). The call to the packing function for the farmer is essentially the same as the call to the original sub-function. However, the farmer must execute a second call, to the unpacking function, to complete the return of data from the worker process. Thus with this framework the master process may execute a sub-function call on any other machine in the processing group.

The process of creating the packing and unpacking functions is not always as easy as it may sound. In the case of the spherical mosaicing algorithm, all of the information about a particular image is contained in a single structure. In the serial version, a pointer to this structure is passed across to the sub-function. When creating a packing function for this call, all of the necessary information contained in the image structure must be passed across. Since the worker has a copy of this structure as well, only the parameters which are needed by the sub-function and are different (i.e. newer) from the worker's versions must be found. This requires both knowledge of the types of information needed by the sub-process and a thorough search of the sub-process code for the use of particular information. Here the abstractions, which make programming in a shared memory space convenient, hide references to data and make the process difficult. In addition, the structure has multiple representations of the same information, at times, so just knowing what to look for is not always enough.

4.4 Creating a parallel framework

When creating a parallel process, some provision must be made for splitting the execution paths of the farmer and workers. When a process is started under MPI, it runs the standard code provided by the user. It knows to behave as a farmer or a worker only through code conditioned on the process' identification number. In the case of spherical mosaicing, it makes the most sense to have all processes run the same code initially, but then to split off the executions before actual calculations began. This allows the processes to each allocate the necessary memory space to store algorithm parameters, such as the number of images or the quaternions for each image. The farmer and worker code split before the code begins the optimization iterations. The farmer code goes on to execute what is essentially the same path as the serial process, while the workers go immediately to code that has them wait for instructions. The farmer can execute the serial code (with a few modifications) because the whole parallel algorithm mimics the serial one. The workers may be thought of simply as extra muscle to help dispatch time consuming operations that the otherwise serial process comes upon.

Some amount of work must be done to make the serial execution sequence take advantage of its parallel resources. For instance, the serial execution sequence will only call one sub-function at a time. It will wait for the sub-function to finish before calling another one. To make the execution sequence parallel, the for loop which cycles through all of the image pairs as it makes the calls to the sub-function must be re-written. The way to achieve parallelization is to create two functions in place of the original call. One function starts the sub-function on a worker machine, the other retrieves the returned data from the worker. Thus it is possible to write a for loop which starts sub-functions on all of the workers, and then retrieves the returned data from the workers and gives them new assignments. By keeping track of which image pairs have been assigned, the complete set of image pairings can be processed.

The serial algorithm was not designed for parallel execution, so some modifications in the way that functions are called is necessary. In the case of the update of camera parameters, a 1x3 gradient and a 3x3 Hessian are built from data generated from all valid pairings. The serial call to the computeCorrelationInternal function passes pointers to the gradient and Hessian so that they may be updated. These are then returned and passed to the function for the next image pairing. The parallel implementation cannot work this way. Since it is parallel, functions are called before the results of the previous calls are returned. Thus a single Hessian or gradient cannot be available for passing from one function to the next. The way to get around this is to add up all of the results after they are returned. To make this possible, gradients and Hessians of zeros must be passed to the functions initially, so that the returned values are due only to that particular function call. These results can then all be summed together to get a result which is approximately the same as the serial version. There are slight differences in values introduced by this procedure because it will accumulate less floating point rounding error than the serial procedure.

The order in which work is assigned to processors is something to be considered. The optimal assignment of work is such that each processor does work on as few separate images as possible. This reduces the amount of memory needed by each processor and limits the amount of virtual memory swapping (if any) that will be required, thus making the execution faster. How to accomplish this for an arbitrary number of processors on a set of images is still an unsolved problem. Since images are typically adjacent to images which have nearby identification numbers, the current spherical mosaicing program has a processor execute image pairs sequentially as they appear in the list of adjacent images used by the program. The processors are started on images which are chosen to distribute the images evenly amongst themselves. Pointers to the list of adjacent images track which image pairs have been completed and insure that all are assigned.

4.5 Parallel Results

Data showing effectiveness of parallel computation over speed/memory.

4.5.1 MPI Efficiency

Before committing to creating the MPI code, it seemed wise to test the efficiency of MPI on various numbers of processors. To do this I devised a simple task consisting of basic arithmetic operations. These operations were assigned to workers by a farmer process and then collected when finished. The speed of the overall group of processors was determined by taking the amount of work finished and dividing by the wall time required for execution. As can be seen from Figure 13, the growth of the speed with the number of processors is linear. Putting this another way, the amount of work done per processor over a fixed period of time is roughly constant. This indicates that the MPI process is efficient. However these times did not include MPI initialization times, which must be taken into account when predicting the overall efficiency of a program. When accounting for initialization times, these results indicate that MPI will be most efficient for applications which give work to a worker in large chunks.

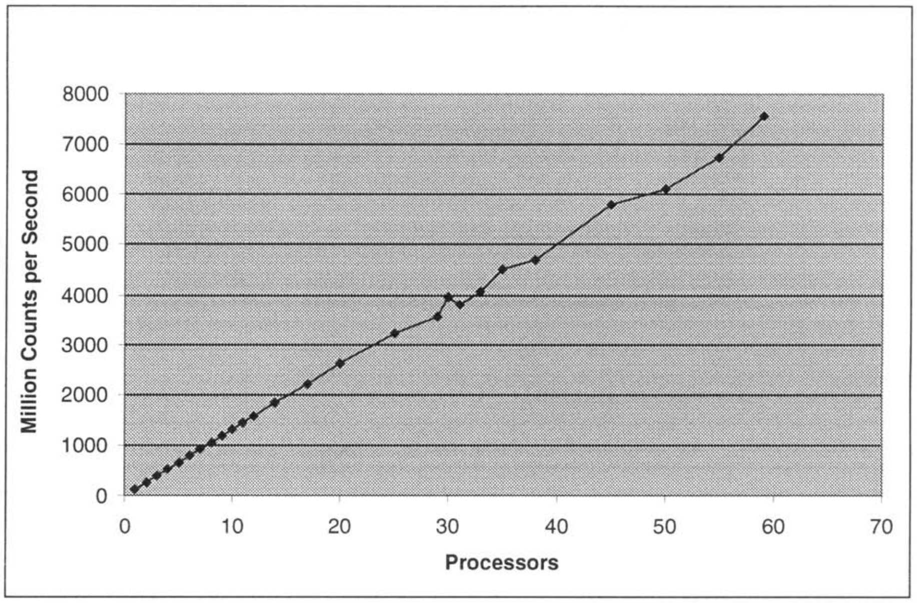

Figure 13: Performance of Tesserae[3] on simple arithmetic as a function of the number of processors.

4.5.2 Output Verification

Perhaps the most important measure of any program is that it generates correct results. Spherical mosaicing is certainly no different. To verify that the program output is valid, I compared images aligned by the parallel algorithm against images aligned by the serial algorithm and images aligned only by Argus. Shown in Figure 14 is an image aligned only by the orientation data from Argus. The images make a reasonably good match in most places, however it is clear that the light pole and leftmost building do not align vertically. The results of the serial algorithm and the MPI parallel algorithm are shown in Figure 15 and Figure 16 respectively. In both cases, the images are very well aligned and there are no alignment errors. The building and light pole are now aligned correctly.

[3] A network of 32 dual processor Pentium II 450 MHz machines.

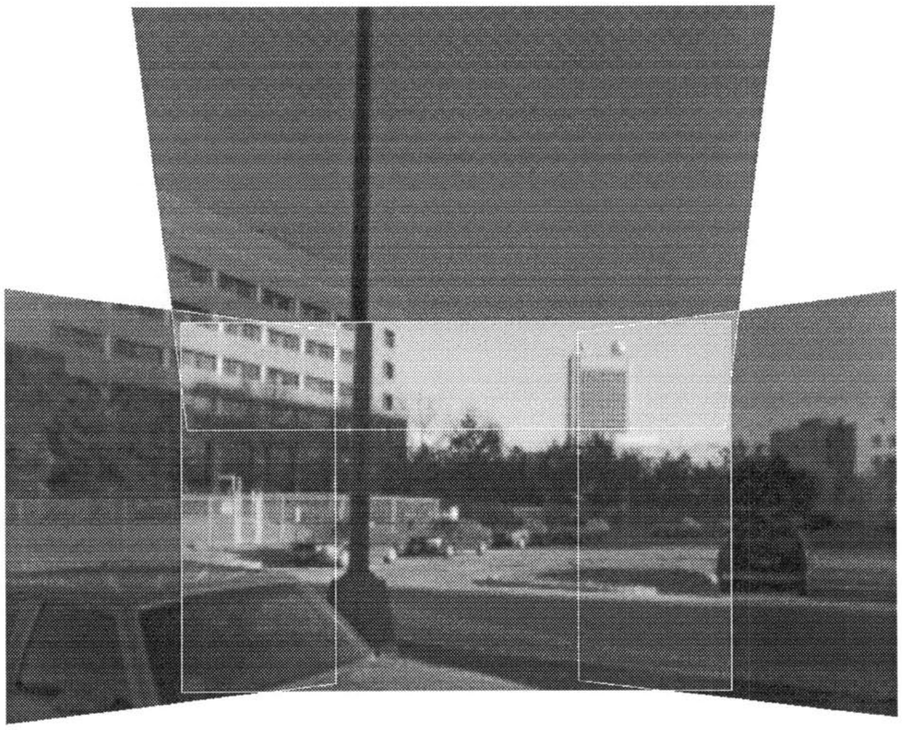

Figure 14: Alignment of images based on Argus measurements alone.

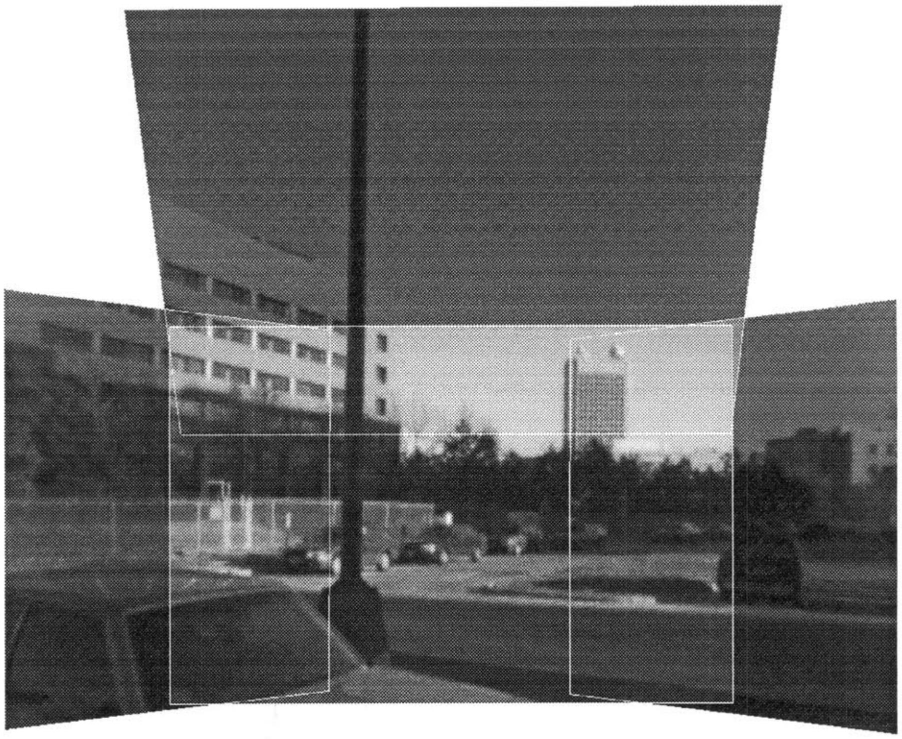

Figure 15: Alignment of images based on serial algorithm.

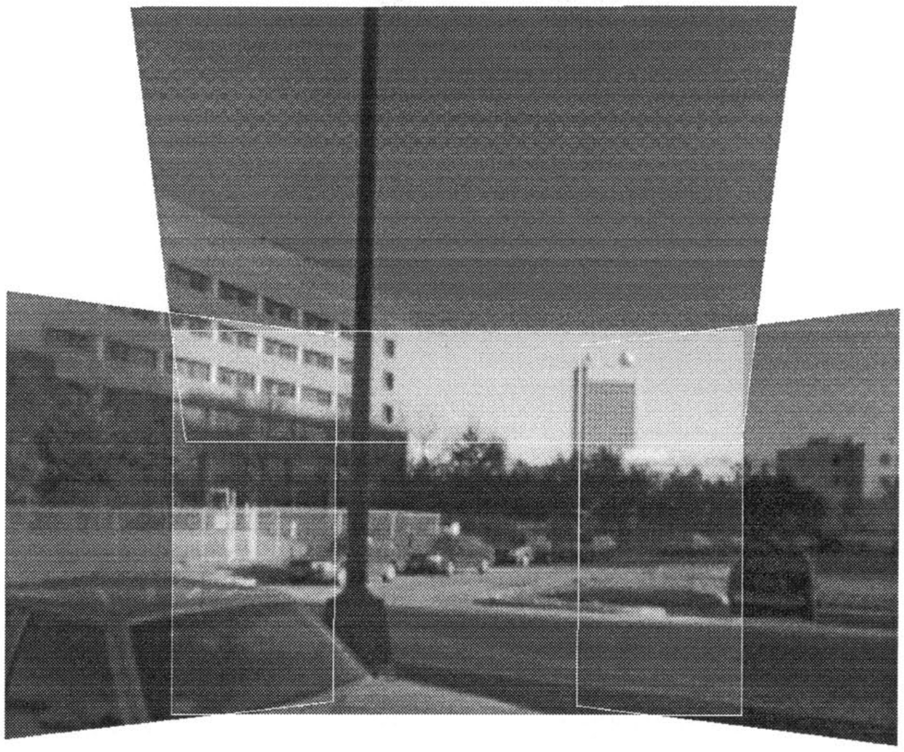

Figure 16: Alignment of images based on MPI parallel algorithm.

A second method of output verification is to look at the parameters themselves. A comparison of the key parameters is given in Table 6. The values themselves are quite similar, as might be expected judging from the similarity of the output images. The errors are all much better than one part in 1000. Errors of this magnitude are most likely due solely to discrepancies introduced by floating point accumulation (see page 30 for more information). Based on the image output and the minimal discrepancies in output values, it is fair to judge the algorithm to be accurate.

Parameter	Serial Algorithm	MPI Parallel Algorithm	Error
Focal Length	507.439	507.429	1 part in 50,000
X Center	199.522	199.510	1 part in 16,500
Y Center	127.829	127.835	1 part in 21,000
Objective Function Value	2717.307	2718.827	1 part in 1,750

Table 6: Comparison of output values of serial and MPI parallel algorithms based on quarter size images.

4.5.3 Wall-Clock Execution Time

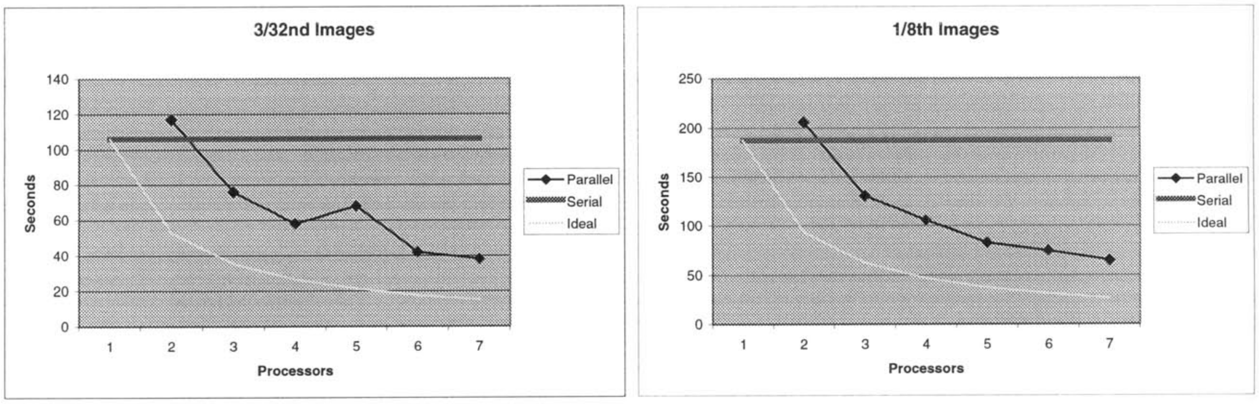

Figure 17: Speed of parallel algorithm against serial algorithm on 3/32nd and 1/8th size images.

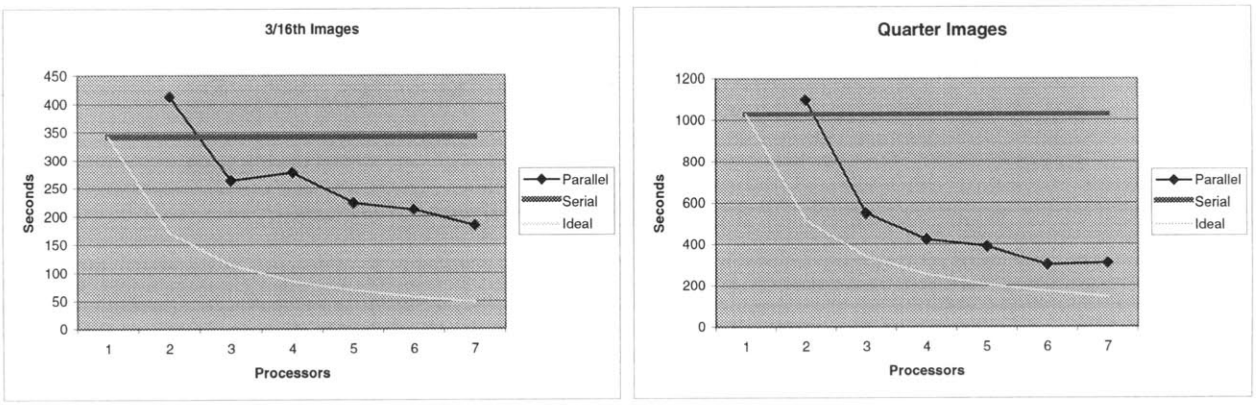

Figure 18: Speed of parallel algorithm against serial algorithm on 3/16th and 1/4th size images.

The execution time in wall time is one measure of the performance of a program. The wall times required for four image sizes are shown in Figure 17 and Figure 18. These image sizes are small, so the overhead associated with MPI is readily apparent. In the case of two processors, the amount of time required is exceeds that of the serial algorithm. This should be expected because using only two processes means that one will be the farmer and the remaining one will be a worker. With only one worker, there is no true parallel processing taking place, and there is still some overhead from MPI, so the overall execution is slower than with a serial process.

Overall, the MPI parallel algorithm definitely improves the wall-clock speed of spherical mosaicing. For all groups of three processors or more, the serial algorithm has a greater wall-clock execution time. This is exactly as one would hope. The ideal wall-clock execution time can be calculated by dividing the number of processors by the wall-clock time required by the serial process. The MPI parallel wall-clock execution times come very close to matching the ideal wall-times (lowest line on the graph). The offset seen is due to two factors. The use of a separate farmer process means that this process is not doing work, and thus the algorithm has one fewer workers than the ideal calculation assumes. The other factor is the overhead introduced by MPI, including the initialization, packing, and message sending times.

The algorithm tends to perform better for larger image resolutions. This is predictable as well, since the overhead associated with MPI parallelization generally does not scale with image size. The algorithm should perform even better on half resolution images and substantially better on full resolution images (since full resolution images require virtual memory swapping under the serial algorithm).

5 Problems

Currently there are several problems and opportunities for improvement in the parallel implementation and algorithm. They indicate areas which should be addressed to further assure the correctness of the implementation, as well as items which could be studied to provide better performance and robustness.

5.1 MPI Initialization

The MPI initialization process is worth brief consideration. The task of bringing all of the processors on line requires some time, and in the case of large numbers of processors, can become lengthy. A graph of the total initialization time required for a simple program is shown in Figure 19. The initialization time is the wall time measured from the command line execution of the process until the first statement of the program after the MPI initialization commands. The trend is definitely worse than linear. I found that a second order polynomial fit the data reasonably well. The reasons for this behavior are not entirely clear because the protocol for initialization is not specified in the MPI documentation. If initialization required only setting up rshell connections from the base machine to all of the machines in the group, then one would expect startup time to be linear. If the initialization process set up a network of rshel connections from each machine to the rest of the machines in the group, then the startup time could be factorial. One possibility is that the initialization sets up n factorial connections, but does so in parallel, giving a response such as Figure 19. The bottom line is that starting an MPI process on 30 machines takes about 9 minutes. This should not be a problem for very large jobs, but it does make it inefficient to use large numbers of processors on small jobs. A good rule of thumb is that anything up to 10 processors has a negligible start time.

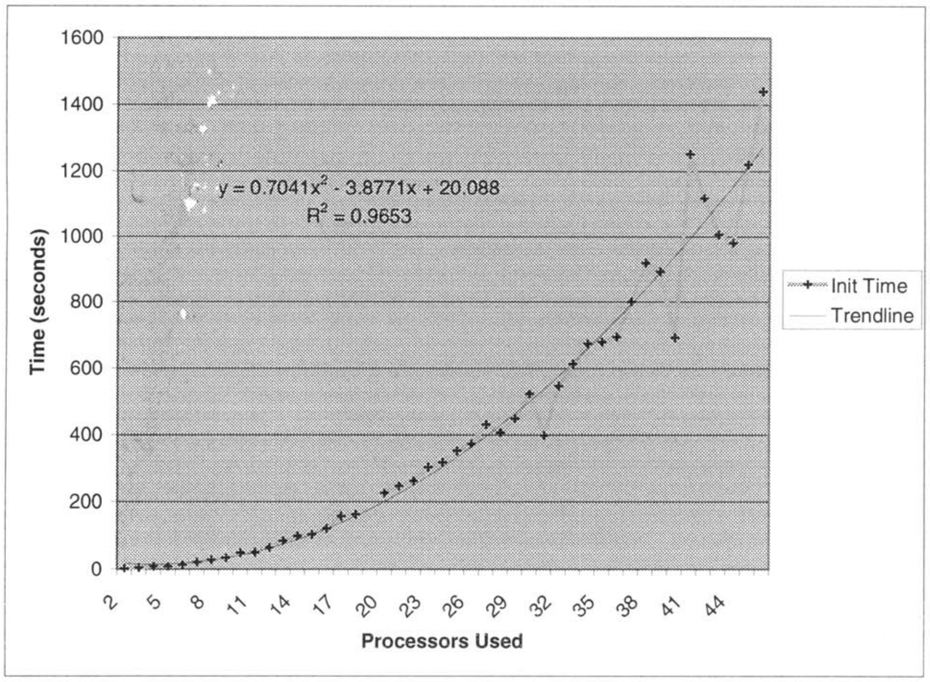

Figure 19: MPI Initialization time as a function of the number of processors used.

5.2 Accumulation Differences

One problem with the changing of the order of the operations to go from serial to parallel is that functions no longer generate exactly the same values. At least some of this variation is due to floating point accumulation errors. These errors occur due to the way in which floating point numbers are stored. When data first starts to accumulate into a floating point number, the destination number is small, so most of the significant digits from the source can be kept. However when the destination variable contains a large number, then more of the significant digits of each addition are lost. One would hope that the error associated with this process would be randomly distributed about zero, very small, and make little difference. However in practice these errors tend to be on the order of 1 part in 10,000 and easily cause divergences of 1 part in 5000 in the image center estimates. These divergences may not be entirely bad, however, since the structuring of the serial process is such that it is subject to much more accumulation error than a parallel process. One method of reducing accumulation error is to accumulate into a double precision floating point variable, and then to convert back to a single precision when done. This preserves more of the significant digits, and has improved the agreement of results between parallel calculations using various numbers of processors.

5.3 Other Issues to be Resolved

The computation of spherical mosaics of full size images is a challenge well suited to this parallel implementation. However the actual implementation of parallel spherical mosaicing on full size images has not been implemented because it is currently being merged with a new version of the code. This new code, known as the largeset version, has added features that handle situations in which the amount of memory required exceeds the physical memory of the machine(s) being used. One of these features is the ability to load only images which are needed at a particular time. This is a big improvement over the original code which loaded all images indiscriminately and used very large chunks of memory as a result. Since discriminate image loading had already been implemented in largeset, it made the most sense to make use of this code and not duplicate efforts. However, since the largeset code is not entirely debugged under Linux, the final version of parallel spherical mosaicing code is not yet available.

There are several issues which need to be resolved with regard to this code. The old code version had trouble running on large images. I found that it would abort when used on full or half sized images. I have no good explanation for this, apart from to say that these sizes easily exceed the memory capacity of the computers being used, and give a very intense load to the file server initially. The debugging of the largeset code will obviously need to be completed. The merged version of the corrected largeset code and the MPI parallel code will also need to be debugged. There should be little trouble with this, however the version that I have implemented currently does not get the same output values as the serial largeset version. In my opinion there are currently too many factors operating on this code to make it reasonable to debug, however this should change once the serial largeset code has been debugged.

6 Conclusion

Parallel programming is a potentially powerful tool for dividing and conquering problems requiring large numbers of computations or large amounts of memory. The ability of MPI to span not only multiple processors, but multiple machines and multiple architectures makes it a useful tool for parallel architectures. Although the results of the parallel algorithm already achieve substantial improvements, the merging of this code with the new swap handling largeset code should produce a program which is not only faster than any previously available, but should also be able to handle the large image sizes which were unmanageable under serial algorithms.

6.1 *Usefulness of MPI Spherical Mosaicing*

As it stands, the MPI parallel algorithm should have a substantial impact on the wall-time required to complete spherical mosaicing on new nodes gathered by the City Scanning Project. Once the MPI parallel algorithm is fully merged with the largeset algorithm, it should be possible to achieve similar speed ups on full images. Without the MPI parallel algorithm, the processing of full size nodes would be prohibitively slow, as the virtual memory swapping pushes the execution time beyond 30 hours.

The MPI parallel algorithm will remain useful to the City Scanning Project over time because it is flexible enough to run under most operating systems and with most practical configurations of single or multiple machines.

6.2 *Further Work*

There are several improvements which could be made to the parallel spherical mosaicing code to improve both wall time execution time and efficiency. These improvements could even be expanded to provide further robustness of the system.

6.2.1 Determining machines with least loads

The Tesserae cluster on which the parallel MPI program was developed is constantly changing. Numerous users make use of its 30 machines, so the work load of any given machine varies with time. When MPI starts a process, it runs on a set of machines designated by a machine file. If a pre-program were to assess the load of the available machines and write the names of the least loaded machines to the machine file, then the wall time of the execution would be less, and the overall computing resources would be used more efficiently. I do this process by hand when starting a large amount of work, so it should be very feasible. It may even be possible to determine the optimal number of processors to use based on the load distribution amongst the available machines.

6.2.2 Dynamic redistribution of loads

Once an MPI process has started, it is still possible and useful to redistribute the work load amongst the participating machines. One way to do this would be to have the master process assign a portion of work to each worker process. When a worker process finishes its work, it can then be assigned work belonging to other workers that are not yet finished. In this way, their are no idle hands. A primitive version of this is currently implemented, however it biases the extra help towards later images in the list rather than distributing help more evenly. A further improvement could be made by considering the case in which a single worker becomes so overloaded by another job that it is unable to finish its assignment in a reasonable amount of time. In this case, other workers could be assigned identical work (assuming all other jobs had been finished), in the hopes that one of the free processes could finish the job more quickly. This sort of robustness is not a necessity for the algorithm, but it could make it much more reliable and would be interesting to implement.

6.2.3 Optimizing image pair order

An interesting problem to ponder is the optimal order in which to consider the pairs of images. The current method of working on images in ascending order probably does a reasonable job of getting workers to work on the same image as often as possible considering the simplicity of the assignment procedure, however one could certainly do better. For instance, each pair of images must be considered twice. Since each consideration requires that exactly the same information be ready in memory, an optimal algorithm might take advantage of this by considering both versions of a pairing in sequence. On the whole, the problem is an interesting one to consider because both the number of images in a node and the number of processors being used may be changed. There may be no easy way to compute an optimal solution, in which case one might attack the problem by pre-computing solutions using brute force methods.

6.2.4 Combining camera update with objective function value update

As mentioned earlier, the function which generates new camera parameter estimates and the function which updates the objective function value make calls to the same sub-function, computeCorrelationInternal. Since these two functions are sequential when viewed across multiple iterations, it is possible to combine them into one step. The new single step would replace the objective function value update. It would execute the normal camera parameter estimate update and give the objective function value as well. (This is exactly what updateInternalParameters does.) There would be some advantage gained in terms of computation time, however this would not be substantial since the two individual steps are already implemented fairly efficiently. The real gain would be in terms of virtual memory swaps on full size images. With this modification the number of loops over all image pairs would be reduced from three to two, resulting in an overall 33% reduction in swapping activity. This reduction would be significant on large image sets which spend a substantial amount of their wall-time on virtual memory swaps.

7 Bibliography

7.1 References

[Coo98] Satyan Coorg. *Pose Imagery and Automated Three-Dimensional Modeling of Urban Environments.* PhD Thesis, MIT, August 1998. (http://graphics.lcs.mit.edu/pub_theses.html)

[DeC98] Douglas S. J. DeCouto. *Instrumentation for Rapidly Acquiring Pose Imagery.* Master's Thesis, MIT, June 1998. (http://graphics.lcs.mit.edu/pub_theses.html)

[Sun99] Qixiang Sun. *Parallelization of the Spherical Mosaic.* AUP, MIT, May 1999.

[TCM98] Seth Teller, Satyan Coorg and Neel Master. *Acquisition of a Large Pose-Mosaic Dataset*, in Proc. CVPR 1998, pp. 872--878. (http://graphics.lcs.mit.edu/~seth/pubs/cvpr98.ps.gz)

[VAN78] Alan VanBronkhorst. NATO Advisory Group for Aerospace Research and Development (ed.). "Strap-down System Algorithms." AGARD Lecture Series No. 95. *Strap-Down Inertial Systems* NATO 1978.

7.2 Image Credits

Figure 1: City Scanning Project. (http://graphics.lcs.mit.edu/city/city.html)
Figure 2: IBID.
Figure 4: IBID.
Figure 5: IBID.
Figure 7: IBID.

8 Appendix: Quick Guide to the Parallel Code

This is a quick user's guide to the code. The aim is to help anyone who might need to use or modify the MPI parallel code.

8.1 Versions

At the time of this writing there are several directories containing versions of the MPI code. Only two are of interest. One is the non-largeset version and the other is the largeset version. The non-largeset version is the one that I created first. It is based off of a version of the code which we ported to Linux before Neel created the largeset code. The largeset version is the result of my merging the largeset code with the MPI code. I haven't fully checked these in, but Neel should have access to both before I leave.

8.2 Files

There is a lot of source code required to make mosaic work, but only a small portion of it is of interest. The files to be worked with should be in a directory named "$BASE/src/image/Apps/(mosaic-version)". In this directory there are several files which have been modified for MPI. They are primarily main.C and mosaic.C as well as their corresponding .H files. The files mos_mpi.C and mos_mpi.H are new to the code and contain most of the MPI information. All of this should be available by checking out the city_base code through CVS.

The MPI compiler and associated libraries and framework are also necessary. Currently they are installed on the Tesserae machines. Should further work be done on a new machine, they will need to be installed there as well. If this is the case, I highly recommend grabbing the MPI RPM and installing MPI this way. RPM's contain pre-compiled binaries and take much of the confusion and frustration out of building it yourself. These are available at RPM repositories such as http://rufus.w3.org/linux/RPM/. Once installed, they work like a charm.

8.3 Code Sections

Here is a brief run through of the MPI code. In the version of the code which is merged with largeset, there are markers at each section of MPI code which indicate where it has been added to the largeset code. A quick grep for "MPI" should reveal these markers.

8.3.1 Split between Farmer and Workers

main.C is the file which starts the mosaicing process. I have added a few lines here to get the MPI process rolling. There is an initial startup call which initializes MPI and lets each process know what its number is. The major change is that there is code which requires the processes to branch. This is so that the farmer process can go about making mosaics and the workers can go about waiting for instructions. Look for an if statement conditioned on "myrank == 0" for the split in execution paths. Here the farmer process proceeds as normal, and the slave processes are directed to bypass this code and enter a waiting loop which allows them to receive and execute instructions from the farmer. Once one knows what to look for, these should be very easy to spot. There is also code which closes down all of the MPI links once mosaicing is done. I placed this by trial and error since the code may finish down different paths. I'm pretty sure that I chose correctly, but it is possible that it could be done better.

8.3.2 Sections of mosaic.C

mosaic.C is the file which contains the key mosaicing functions. I modified three of these as described in my thesis. Apart from that there is very little modification of this file. There are three for loops which I re-wrote so that they could execute parallel commands. One is in the camera update function, updateInternalParameters, the second is in the rotation update function, updateAllRotations, and the third is in the objective function evaluation, contained within updateAll. All three of these new loops are very similar, so I will visit them only once.

I start the loop with some code which initializes data structures needed by the loop. One of the first is something called "imagelist" where I store all of the adjacency information. This information is also available through the imageset structure, but I found that my own format was easier to deal with. imagelist is an array which has one vector for each image. Within a vector are stored the numbers of all adjacent images to the image associated with the vector. I create pointers to each image vector in order to keep track of which image pairs have been processed (processing happens in the order in which the images are stored in a particular vector, though the order in which image vectors are considered may vary). I also create some variables to help out in the accumulation process. These may be zero valued variables which are sent to functions, or variables made of double precision floating points to help in the data collection process.

The loops begin with a while statement (while(!done)). Inside this loop there are two sub loops. One loop assigns work to processes and the other loop retrieves finished work from processes. The assignment loop begins by cycling through each process and assigning it a beginning image. Currently I assign each process a beginning image which is spaced evenly apart from those given to the other images. Once a process has a beginning image, it looks up that image in the imagelist and takes the next available image pair. In this way, it starts with its base image and moves down the chain until all images after it have been processed. This process is crude at best and would be relatively easy to reprogram. Once a process has been assigned an image, the other processes are assigned images as well. Then the sequence proceeds to the retrieval loop. The retrieval loop retrieves data from the processes in the order in which they were assigned work. Once all data has been received, the process returns to the assignment loop. The retrieval process could be reprogrammed as well. It would be much better if it retrieved from the completed process first instead of process 1. After the retrieval, new work should be assigned immediately instead of waiting for all computers to report back (since actual speeds on Tesserae machines can vary).

Once the loop has finished, any data which has been collected during the process is put back into the appropriate variables (in some cases this is handled within the loop itself).

8.3.3 A look at mos_mpi.C

I placed as much MPI related code as I could in a separate file, mos_mpi.C. This file contains several types of functions: some handle starting and stopping function calls, some handle packing and unpacking data for MPI messaging, one handles instructions for worker processes, and one is a mathematical function that I found useful.

The functions which handle starting and retrieving computeCorrelationInternal calls are called **cCI_MPI_Start** and **cCI_MPI_End**. The start function is called more-or-less like a normal computeCorrelationInternal call from updateInternalParameters or updateAll. It calls a packing function to pack the necessary data and then sends this data to a worker process. The end function does this in reverse: it receives a message from a worker function, unpacks it, and returns the information to the calling function via pointers. There are analogous functions for computeCorrelationRotations (**cCR_MPI_Start** and **cCR_MPI_End**) which work very much the same way.

The next function in the file, **slaveProcess**, is the loop used by all worker processors. In main.C when the farmer and worker processes are split, the worker processes run this function. The function waits for a message from the farmer process, executes it, and then waits for another message. There are four operations which the slaveProcess is likely to run: one is cCI_Norm which is called during updateInternalParameters, one is cCI_Reduced which is called during updateAll, one is cCR_Oper which is called during computeCorrelationInternal, and the final operation is to quit and exit the slaveProcess routine. The first three of these operations have fairly predictable execution sequences: they unpack the received message, make the necessary call to computeCorrelationInternal or computeCorrelationRotations, pack the returned information, and send it back to the farmer process. The quit function executes the MPI_Finalize routine which shuts down MPI on the worker process and allows it to exit execution gracefully.

The next six functions handle the packing and unpacking of data for MPI messaging. Their function is fairly obvious. In the case of a packing function, a vector of floating point numbers is created and important pieces of data are stored in it. If some of these are integers, such as the image number, then they are first converted to floating point numbers. The unpacking operation is similar. The floating point vector which has been received is used to update important variables. It is then necessary to run the "images[i]->cam->computeOtherReps" function because there are redundant representations of data which must be updated (the camera internal parameter 3x3 matrix, for instance). There are two functions associated with computeCorrelationInternal: **cCI_Pack** and **cCI_Unpack**. These functions work both for the passing of data to the worker and passing it back to the farmer. (In this case there was sufficient symmetry in the calls to allow only two functions.) computeCorrelationRotations uses four calls:

cCR_Pack_MtoS (Master to Slave), **cCR_Unpack_MtoS**, **cCR_Pack_StoM**, and **cCR_Unpack_StoM**. The first two calls are used when transferring data from the farmer to the workers and the second two are used when transferring from the workers to the farmers.

Finally, the function **round** was added since I had trouble finding a function which would round a floating point number to a whole number.

8.4 Compiling and running

MPI requires its own compiler (provided in the RPM and available on Tesserae.) Switching to this compiler is simply a matter of setting CC to "mpiCC" or cc to "mpicc" depending on whether you are using C++ or C (probably the former). Also, make sure that the makefile includes mos_mpi.C since this is a new file.

The execution environment for MPI is not specified by the MPI standard. However, there is a command which spawns programs on specified numbers of processors. This is called "mpirun". An example of a mosaic command under MPI which I have used is:

```
mpirun -v -np 5 ./mosaic -texture 0.01 -I img/3_32nd -C
pose/full/initial -W pose/3_32nd/mosaic_MPI -batch -start 0 -end 46
/d8/city/tsq/100nodes/node06
```

This starts mosaic on 5 processors.

There are a few other necessities when running MPI that I will try to remember here. MPI requires a file listing which machines it can use. Under Linux, this file is named "machines.LINUX" and needs to be stored somewhere where mpirun can find it. (I put mine in my root directory.) The format of this file is to list each machine by its internet alias (Tesserae015.lcs.mit.edu for instance) on its own line. Another requirement is the ability to use rsh to get to each of these machines. This can be accomplished by listing them in your ".rhosts" file.

[illegible] - 73