2-D Object Recognition Using a Parallel Implementation of Geometric Hashing

Carlo Pascoe and Christian Davis, *Electrical Engineering, University of Florida*

***Abstract* – Many image processing applications require the ability to promptly recognize two-dimensional objects within an image or database of images. A technique known as geometric hashing is used in computer vision to enable the recognition of targeted shapes and figures. Objects of interest in this application must be quickly matched with models in a large, predefined database located somewhere in memory. Matches must even be made when the target objects have undergone transformations or have been partially obstructed within the field of view. Computer vision may require this technique to perform its operation on a certain amount of images per second, or frame rate, and is limited by the speed of the architecture it is run on. Hence, it is valuable to explore the advantages of performing such an algorithm on a parallel architecture. The focus of this paper is on comparing and observing the performance benefits of implementing a parallel 2-D object recognition algorithm with geometric hashing over the performance of a serial one. We have shown that the performance of the algorithm can be significantly improved by increasing the number of processors or threads it is run on.**

TABLE OF CONTENTS

I. INTRODUCTION

Geometric hashing is an algorithm used to enable the recognition of objects within an image or stream of images and is needed in several applications seen today. For example, in an application such as robotics it may be necessary for a certain machine to pick out or pick up certain items in a lab or assembly line. These computers use video cameras at a certain frame rate to quickly observe their environment for potential target items. Another application of object recognition is in structural alignment of proteins. In this domain, a sequence alignment is made by establishing similarities between different protein structures, thus, performing a large amount of object comparisons in the associated database. Matching in this case must also occur even though most of the target proteins in the images acquired are rotated, translated, scaled, and even obscured which raises the complexity of the algorithm. Object recognition is becoming essential in many present applications of computer vision and must perform efficiently to function usefully.

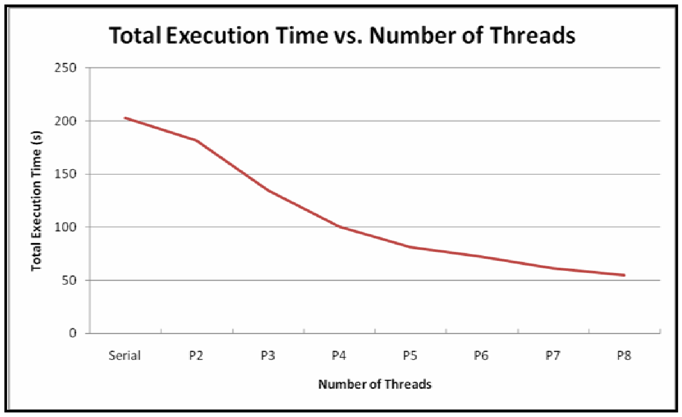

Fig. 1. The data in this graph shows that as the amount of threads is increased, the total execution time of the program decreases.

To successfully perform object recognition, an object must be quickly recognized from a stream of input images. Each image must be analyzed and a possible match should be successfully recorded before the next image is processed. Running this application or algorithm on a single processor may impose a certain limit on the frame rate of the input. The processor can only work on one entire acquired scene at a time and may not continue to the next image until it is finished with the current one. To solve this problem, the geometric hashing technique may be implemented on a parallel architecture to allow for a faster frame rate and an improved throughput of data.

The focus of this work was to create a functional parallel implementation of the 2-D geometric hashing algorithm to perform object detection within a stream of images. A survey of related literature produced no actual code so much time was spent not only creating the parallel implementation, but a serial implementation used for baseline comparison as well. The performance and accuracy of each implementation are compared and it is our goal to quantify and show the performance benefits achieved by running the object recognition program on a parallel architecture. The *Unified Parallel C* (UPC) extension to ANSI C is used to parallelize the algorithm and is executed on the Mu cluster of the High Performance Computing and Simulation (HCS) Research Laboratory at the University of Florida (UF). The UPC implementation takes advantage of the multiple processors offered by the cluster through distributing pieces of the images to multiple processors and analyzing the image fragments simultaneously. This accelerates the processing speed of an individual image and, as a result, is observed to increase the overall frame rate of the application. The function of the geometric hashing algorithm and its associated hash table is explained in more detail in the sections that follow.

The performances of the serial and parallel implementations were compared by measuring their total execution time as well as running them through the Parallel Performance Wizard (PPW) tool provided by the HCS lab. This tool gathers and produces profile data from a program execution and provides a graphical interface that allows the gathered data and performances to be analyzed efficiently and used to identify potential bottlenecks in the code.

The results of our experimentation have shown that the parallel implementation provides the exact same 94.7% accuracy as the serial implementation yet runs 3.71 times faster when run on 8 processors. This accuracy and speedup was achieved when provided a batch of fifty test images representative of a realistic input to the system. Since the execution times of each implementation fluctuate greatly depending on the inputs to the system, speedups will vary. The fluctuations are mainly due to the $O(N^3)$ complexity of the algorithm.

The functionality of the overall algorithm was verified and the behavior of the parallel algorithm was tested over a varied number of processors. Our data illustrates that as the amount of threads or processors increases, the total execution time of the algorithm decreases. A plot of the recorded total execution times of the serial implementation and all parallel processor amount variations is shown in Fig. 1. In addition to comparing the performance of varied processor amounts, our study also tested for the behavior of running a different parallel algorithm that essentially distributed the serial implementation to multiple processors. This alternative parallel distribution demonstrates the need for load balancing and will be further discussed in Section VI.

In Section II, a brief overview will be given of some related work in this application. The information in Section III will provide background information on the topics discussed in this paper. That will be followed by a description in Section IV of the implementation of the algorithm used and then a depiction of the experiment performed and the results obtained in Sections V and VI respectively. Lastly, conclusions and future work will be presented in Section VII.

II. RELATED WORK

A popular area of research for geometric hashing is in the scientific community of protein structural alignment. The article [4] explains how this algorithm is used in finding similar interaction proteins in order to better describe their functions by how they bind to other compounds. More specifically, the function of a protein is described by their binding and the actual structure of the interaction site. By knowing the function of a certain protein by its interaction site, finding another protein with a similar site will help describe the function of that particular protein. In a more useful operation, an analysis of a particular large group of proteins can return the function of each one of those proteins by matching them with those in a pre-existing database of profiles. The authors of [4] describe a concept of “partial” geometric hashing to decrease the limited memory usage associated with the large collection of protein models. The hash table in this case only contains a fraction of the site structures for matching. After potential candidates are established, the overall structures are used to verify the model match.

The area of information and security assurance is also pursuing solutions with geometric hashing. The topic written in article [5] describes the implementation of geometric hashing as a solution to the auto-alignment problem occurring with fingerprint databases. The fuzzy fingerprint vault is a particular realization of a secure fingerprint database discussed in the paper. When scanning a finger, an un-aligned scan will not match a fingerprint in the database. A fast, yet memory inefficient solution, is the use of geometric hashing. To reduce the large amount of memory needed, the authors of [5] have devised an approach that explores a tradeoff between time and memory. Rather than computing all the hash table entries a priori which reduces execution time (but increases table size), the authors have devised a way calculate the hash table entries on-the-fly. This method reduces the memory requirements but is much slower than retrieving from a pre-defined hash table.

Researching related work in implementing a geometric hashing algorithm on a parallel architecture did not turn up any articles that linked a real application of the algorithm to a parallel implementation. Instead, work found in articles such as [8] and [9] only discussed modifying the algorithm from a serial implementation to a parallel implementation. Our work specifically focuses on implementing an application of geometric hashing for a parallel architecture using UPC.

Some other areas of research worth commenting on include database image queries, detection of irregularities in X-Rays, and other biometric identification methods such as facial recognition.

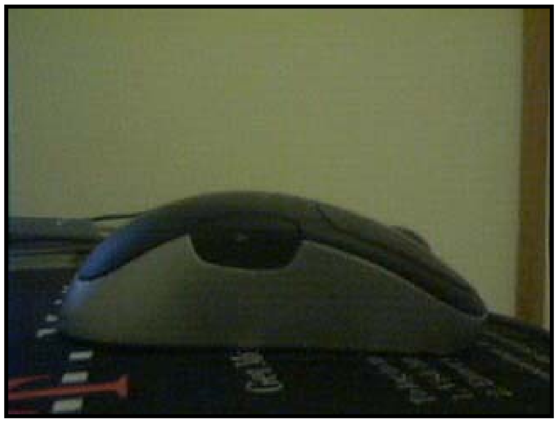
Fig. 2. The mouse in this image is the object to be looked for.

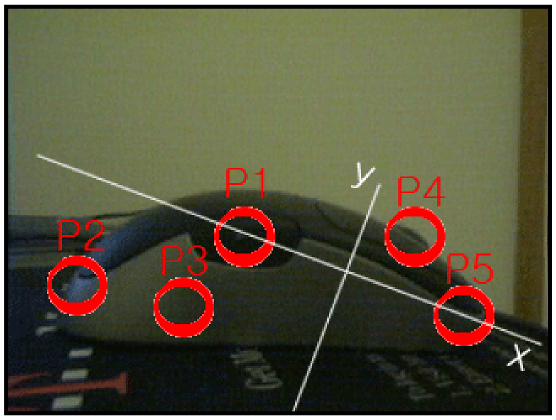

Fig. 3. Shows the object mouse with labeled feature points and the basis formed from points 1 and 5.

III. APPROACH BACKGROUND

Geometric hashing is a computationally efficient way of finding target objects inside of other images. The brute force manner of searching images for objects involves comparing every pixel in the image to every pixel in a database of desired objects for possible matches. In the case of geometric hashing, images are searched for key features and efficiently compared only against certain object features in a hash table. Accesses to the hash table are based on the geometric information in the image which negates the need to search the entire object database. Because of the way the object features are stored and accessed, the objects can be found in an image even if they have undergone certain transformations such as scaling, rotation, or translation. Though geometric hashing can be used to search for 3D objects under all kinds of transformations, this project will only focus on searches for 2D objects under rotations, translations, and scaling that may or may not be partially covered by other objects in an image.

Geometric hashing is implemented as follows. First a collection of objects that need to be found in other images are searched for key features such as lines, curves, or points. Every combination of the discovered features in a particular object is used to define a unit-length basis which is then used to describe the position of the other features in reference to the coordinate system defined by the basis. The object number and reference basis are placed in hash table bins defined by the feature position. This information is pre-computed and used when searching the images for the objects at runtime.

When it is time to search an input image for an object it is only necessary to search the image for key features rather than the entire object. With a list of all key features present in the image (which should be considerably less than the total number of pixels in the image) every combination of two features is used to define a unit-length basis using the same process that the object bases were calculated with. Once again the basis is used to describe the position of every other feature but now the calculated position is used to access the hash table rather than populate it. If an object is present in the image, the positions should map to the hash table bins that hold information for that object; if an object is not present it is highly unlikely that all or even a significant number of the bins containing the object information will be accessed. If the feature coordinates map to a full bin then the object and basis information in that bin is used to “cast a vote” for the object and basis by incrementing a counter. Once all of the coordinates have been checked, the counters are compared against some threshold and considered as potential object matches if they exceed it.

Suppose it is necessary to find the computer mouse from

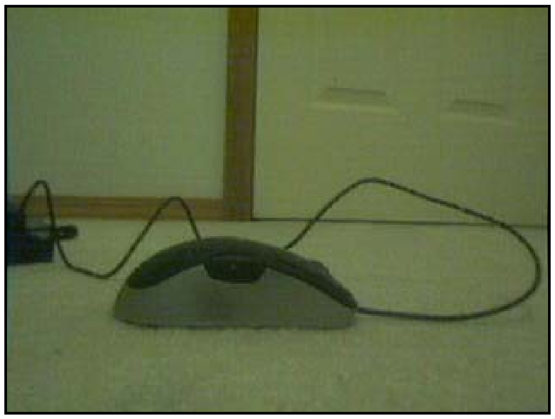
Fig. 4. Input image that will be searched for object match.

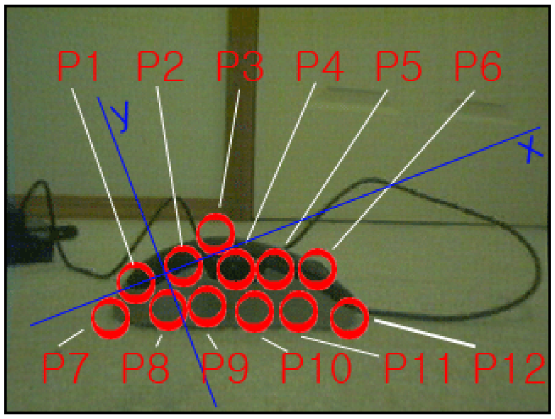

Fig. 5. Searched image with labeled feature point matches and basis formed from points 1 and 2.

Fig. 2 in another image. In order to use geometric hashing the mouse's key features must be placed in the hash table. One way to describe the mouse is to take certain points such as P1, P2, P3, P4, and P5 as feature points for the object. Every combination of these points is used to define a unit length-basis. Fig. 3 depicts an example of possible points P1-P5 with the basis formed from P1 and P5 shown. Using the basis shown in Fig. 3 will result in hash table entries at (0.25, 0.25), (-0.6, -0.5) and (-1.1, -0.6).

Once the feature information is loaded into the hash table, any image can be searched for the presence of the mouse using the geometric hashing algorithm. Say the image in Fig. 4 is searched and 12 key features are found. Every combination of the 12 points is used to define a basis which is then used to describe the position of every other point. Shown in Fig. 5 is an example of the discovered features P1-P12 with the basis from P1 and P2 shown. The basis formed from P1 and P2 above should not yield a correct match with the mouse from Fig. 2 because the coordinates of P3-P12 with respect to the basis should not map to full hash bins. Although, when the algorithm gets around to using other bases, like the basis formed from P1 and P4 or the basis formed from P4 and P6, there should be a match because these are bases formed from feature points identified in the reference object from Fig. 3.

IV. ALGORITHM IMPLEMENTATION

In our research, the geometric hashing algorithm was first implemented serially in the C language to integrate all the requirements of the algorithm necessary to detect an object within an image and provide a baseline to compare with the parallel implementation. This serial program was used to test the functionality and debug the preliminary versions of our parallel program. The parallel implementation was created using UPC which is supported by the Mu cluster. Detailed explanations of both program implementations used in our study are provided next in this section.

A. *Serial Implementation*

Full code for the serial implementation is provided in Appendix A. The flow chart in Fig. 6 describes the general structure of the serial implementation. In this subsection, the serial code will be explained in detail followed by a discussion of several design choices and tradeoffs considered.

The first thing the code does is load in a hash table and key features from a pre-populated DAT file (this corresponds to lines 65-76 in Appendix A). The hash table is populated using a different program that has also been provided in Appendix C. This program searches an input object for the most unique features and chooses a subset that is evenly distributed throughout the object. The subset is then used to populate the hash table using the same algorithm that is used to read it (the algorithm will be discussed a little later).

After the hash table data is stored in local memory the program begins to cycle through all of the images until there are none left. If there are images left to be processed the code loads in the current image (78-92 in Appendix A).

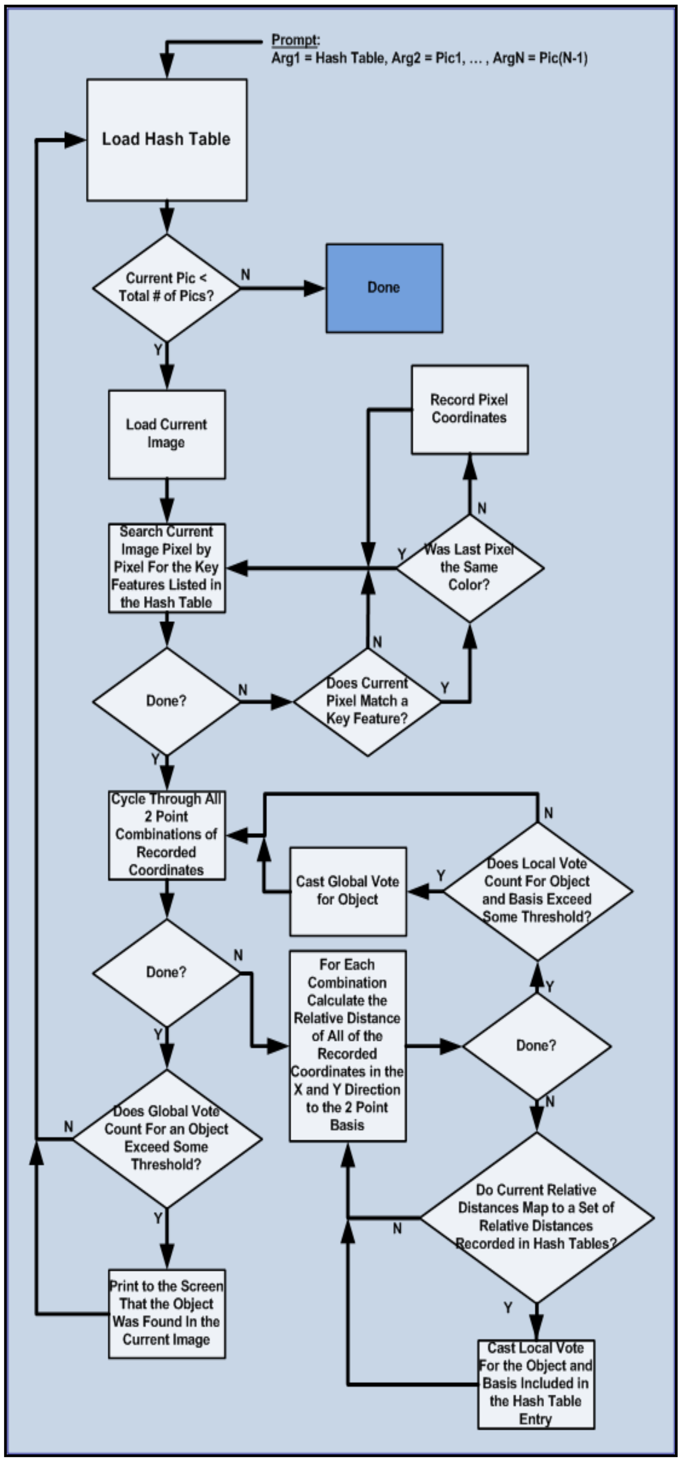
Prompt:
Arg1 = Hash Table, Arg2 = Pic1, … , ArgN = Pic(N-1)
Load Hash Table
Current Pic < Total # of Pics?
N
Done
Y
Load Current Image
Record Pixel Coordinates
N
Search Current Image Pixel by Pixel For the Key Features Listed in the Hash Table
Y
Was Last Pixel the Same Color?
N
Done?
N
Does Current Pixel Match a Key Feature?
Y
Y
Cycle Through All 2 Point Combinations of Recorded Coordinates
N
Cast Global Vote for Object
Y
Does Local Vote Count For Object and Basis Exceed Some Threshold?
Done?
N
For Each Combination Calculate the Relative Distance of All of the Recorded Coordinates in the X and Y Direction to the 2 Point Basis
Y
Done?
Y
N
N
Does Global Vote Count For an Object Exceed Some Threshold?
Do Current Relative Distances Map to a Set of Relative Distances Recorded in Hash Tables?
N
Y
Y
Print to the Screen That the Object Was Found In the Current Image
Cast Local Vote For the Object and Basis Included in the Hash Table Entry

Fig. 6. Flow chart describing the general structure of the serial implementation of the geometric hashing algorithm.

The current image is searched pixel by pixel and compared against a list of key features that were included in the hash table DAT file but are not actually entries in the hash table (96-113 in Appendix A). When a match is found, its coordinates are recorded in a list. The code in Appendix A searches for individual pixels as key features but the features searched for could be anything. The choice of feature type is an accuracy to computation tradeoff. For a pixel feature type there are at most M*N*NumOfKeyFeatures comparisons for an MxN image but it is more likely to find stray pixels that match a key feature possibly leading to false positives. For a five pixel cross feature type, there is a significantly less chance of a false match (due to 120 bits uniquely identifying the feature rather than 24bits), however, there are now at most 5*4*M*N*NumOfKeyFeatures comparisons for an MxN image (this number of comparisons increases dramatically if object scaling is considered). The extra comparisons are not only due to the additional pixel values but also due to their orientation. As you can imagine, this gets a lot worse as the complexity of the feature type increases such as lines and curves. In an attempt to reduce the number of features found the coordinates for a large contiguous block that just so happens to be the same color as a key feature are omitted (i.e. the next pixel is checked only if it is different than the previous pixel).

With a complete list of all key features present in the image (which should be considerably less than the total number of pixels in the image) every combination of two features is used to define a unit-length basis (points P1 and P2) and every combination of that two point unit-length basis and every other feature point (point P3) is used to calculate the inputs to the hash function (114-151 in Appendix A). This is implemented in code with a triple nested *for* loop. The Y input to the hash function is computed by finding the shortest distance between P3 and the line formed from P1 and P2, then dividing that by the distance between P1 and P2 in order to normalize it (line 127 in Appendix A). Mathematically this is equivalent to calculating the cross product of the two vectors P2 and P3 with P1 as their origin (actually equivalent to the area of the parallelogram formed from the two vectors), dividing it by the distance between P1 and P2 (dividing by the length of the parallelogram leaving the height), and finally dividing it again by the distance between P1 and P2 in order to normalize the value. The X input to the hash function is computed by finding the shortest distance between P3 and the line passing through P1 and perpendicular to the line formed from P1 and P2. Once again the distance is then divided by the distance between P1 and P2 in order to normalize it (line 126 in Appendix A). Mathematically this is equivalent to calculating the dot product of the two vectors P2 and P3 with P1 as their origin, dividing it by the distance between P1 and P2 (equivalent to finding the projection of the P3 vector onto the P2 vector), and finally dividing it again by the distance between P1 and P2 in order to normalize the value. The normalization is done in order to accommodate object scaling. If an object is present in the image, the X and Y values should map to the hash table bins that hold information for that object; if an object is not present it is highly unlikely that all or even a significant number of the bins addressed by X and Y will contain object information. If the feature coordinates map to a full bin then the object and basis information is used to "cast a vote" for the object and basis by incrementing a local counter variable addressed by the object number and basis number. Once all of the coordinates for a particular basis have been checked, the local counts are compared against some threshold and considered as potential object match if they exceed it; if the local count exceeds the threshold then a global counter for that object is incremented. The threshold is determined by the number of key features per object and some precision variable that adjusts the accuracy of the algorithm.

Every source read discussed calculating the inputs to the hash function by finding the rotation and scaling transformations about the midpoint between points P1 and P2 that send P1 to the coordinate (-1/2, 0) and P2 to the coordinate (1/2, 0). The transformations are then applied to P3 and its new X and Y coordinates are used as inputs to the hash function. This involves several floating point additions and multiplications when implemented in computer programs. With knowledge of mathematics, we were able to derive the much more computationally efficient technique described in the previous paragraph. When implemented in code many of the computations needed to find the dot product for input X can be reused to find the cross product for input Y. Also, because of the way we have implemented the hash table, all but one of the multiplications and divisions needed per input calculation are integer operations rather than floating point operations.

Once all of the 2 point bases have been cycled through, the global counts are compared against some global threshold and the object is considered present if its counter exceeds the threshold (lines 152-156 in Appendix A). If an object is discovered in an image, the program reports "Object X found in Image Y." At this point if the program has exhausted its list of input images the program terminates, otherwise it continues with the next image.

There are many different design choices that affect the performance of this geometric hashing implementation. Some of the most important are the feature type, the number of features used to describe the object, the threshold precision, and the number of features found in an input image. As mentioned previously, the choice of feature type leads to an accuracy to computation tradeoff. The more complex the feature type the greater amount of computation needed to find that feature in an image. Also, the feature type greatly impacts the number of features found in an input image which above all other factors increases the total execution time by increasing the number of iteration performed in the triple nested *for* loop. Both the number of features used to describe the object and the threshold precision affect the accuracy of the implementation. If the number of features used to describe the object is too low then the probability of finding stray feature points that may map to full hash bins and trigger increments of the vote counters increases dramatically. This problem is magnified when the threshold precision is set too low reducing the number of false hits required to trigger a vote. If the number of features used to describe the object is too high then the number of features found would increase.

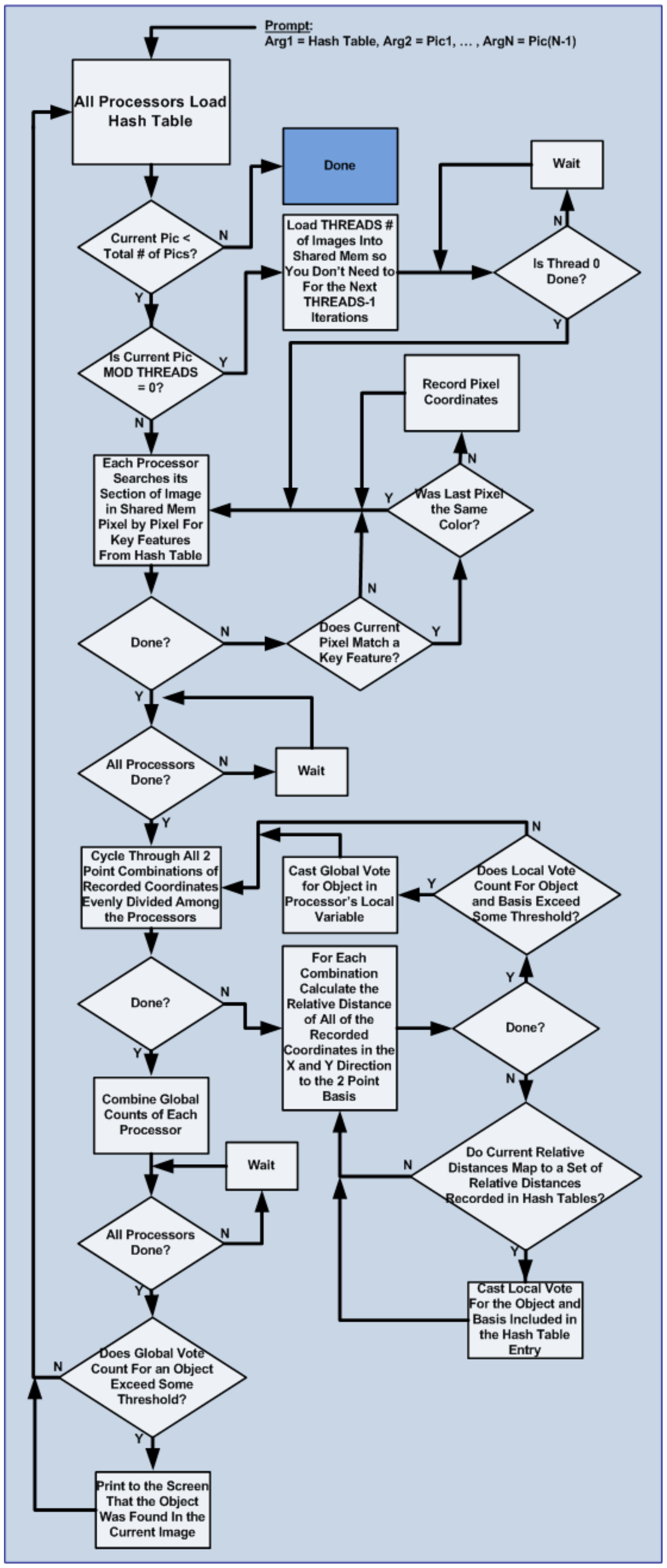
Prompt:
Arg1 = Hash Table, Arg2 = Pic1, … , ArgN = Pic(N-1)
All Processors Load Hash Table
Done
Wait
Current Pic < Total # of Pics?
Load THREADS # of Images Into Shared Mem so You Don't Need to For the Next THREADS-1 Iterations
Is Thread 0 Done?
Is Current Pic MOD THREADS = 0?
Record Pixel Coordinates
Each Processor Searches its Section of Image in Shared Mem Pixel by Pixel For Key Features From Hash Table
Was Last Pixel the Same Color?
Done?
Does Current Pixel Match a Key Feature?
All Processors Done?
Wait
Cycle Through All 2 Point Combinations of Recorded Coordinates Evenly Divided Among the Processors
Cast Global Vote for Object in Processor's Local Variable
Does Local Vote Count For Object and Basis Exceed Some Threshold?
Done?
For Each Combination Calculate the Relative Distance of All of the Recorded Coordinates in the X and Y Direction to the 2 Point Basis
Done?
Combine Global Counts of Each Processor
Wait
Do Current Relative Distances Map to a Set of Relative Distances Recorded in Hash Tables?
All Processors Done?
Cast Local Vote For the Object and Basis Included in the Hash Table Entry
Does Global Vote Count For an Object Exceed Some Threshold?
Print to the Screen That the Object Was Found In the Current Image
Y
N

Fig. 7. Flow chart describing the general structure of the parallel implementation of the geometric hashing algorithm.

Also, the hash table size would increase due to the increased number of hash table entries and bits required to store a single hash table entry; for N features and O objects there are O*N*(N-1)*(N-2) hash table entries and O*(2Log$_2$N+ Log$_2$O) bits required per entry. If the threshold precision is too high then the probability of missing an image that is present increases. This is due in part to the fact that sometimes the feature data is not properly found or the data may be corrupted during transformations such as scaling or rotations. If an object within an image is partially occluded by some other object then some of the feature points may be covered and therefore will not be discovered when the image is searched. Due to the discrete nature of pixels it may be impossible to completely preserve the features of an object under certain transformations.

These design choices are at odds with each other making it impossible to determine the optimum configuration without knowing the input to the program a priori. The design choices were left as variables to the program, allowing for program tweaking after testing.

B. Parallel Implementation

The serial implementation was modified to run on a variable number of processors using the Unified Parallel C extension to the C programming language. The final version of the parallel implementation reads in a collection of images at runtime and processes them one at a time like the serial implementation does, but the work is evenly distributed amongst multiple processors rather than one. Each image is equally distributed amongst all of the processors and each processor searches its own section of the image for key features; this is done rather than sending each processor its own image to improve load balancing (an image with 10 objects in it would require much more work than an image with absolutely no key features in it). Once all processors have completed searching the image, each processor then collects the discovered feature coordinates from each other processor to populate a local list. This amounts to all-to-all communication assuming all processors find at least one key feature. With a complete list of the found features, all processors perform the search algorithm on an equally partitioned subset of the required loop iterations. Each processor keeps a local vote counter. When all processors have completed their subset of the computation, the local vote counters are accumulated in a global variable and a single processor reports if an object has been discovered.

Full code for the parallel implementation is provided in Appendix B. The flow chart in Fig. 7 describes the general structure of the parallel implementation. The next few paragraphs describe the parallel code in detail.

The first thing the code does is load in a hash table and key features from a pre-populated DAT file in the same manner as the serial implementation (lines 68-78 in Appendix B). Since each processor requires extensive use of the hash table, each processor loads its own copy. The load is implemented using standard C file I/O; an attempt to implement the file reading using parallel UPC-IO was quickly abandoned when it was discovered that the Mu cluster does not support it.

After the hash table is stored on each processor, the program begins to cycle through all of the images until there are none left but not in the same way used in the serial implementation (lines 78-108 in Appendix B). Once again, because the Mu cluster does not support UPC-IO, the image file read must be implemented using standard C file I/O. If the image read was implemented the same way in the parallel version as in the serial version, only one processor could load in an image while the others remain idle. In order to fully utilize all of the processors, assuming there are enough images left to be processed, each processor loads a separate image into shared memory every *THREADS* iterations so that no images need to be loaded for the next *THREADS*-1 iterations. On the iterations where images are loaded, the image loaded by Thread 0 will always be the next image to be processed. Because of this, it is only necessary to wait for Thread 0 to finish loading the image rather than waiting for all processors to finish. This is realized with the strategic use of *upc_notify* and *upc_wait* statements.

In an earlier implementation of the parallel code, a different image load scheme was tested. The code was set up so that only one thread was responsible for loading images and the rest of the threads were responsible for processing the images. This is a classic producer/consumer implementation. This scheme was eventually abandoned because of load balancing; it was discovered that the image loading thread was usually idle due to the unbalanced workload.

With the current image in shared memory, each processor searches its section of the image pixel by pixel for the key features loaded from the hash table DAT (112-133 in Appendix B). The search algorithm used here is the exact same algorithm used in the serial implementation but for a subsection of the image. The subsection searched is determined by its affinity to a processor which is ultimately determined by the block distribution in shared memory. The image data was distributed equally into *THREADS* number of blocks with a blocksize equal to image size divided by *THREADS*. That is, the data is striped horizontally across the image. When matches are found, their coordinates are recorded in a local list.

In order to proceed, each processor needs access to all discovered feature coordinates. Since each processor stores the coordinates of discovered feature points from its own section of the image in a block of shared memory with affinity to itself, the program could continue after a barrier synchronization but every other processor would have to perform a remote read when trying to read these coordinates for calculation. This would be acceptable if the other processors only read the other coordinates on occasion but that is not the case for the geometric hashing algorithm; for N feature points, the parallel implementation requires each coordinate pair be accessed at least N*(N-1) times by each processor. This is an unacceptable number of remote accesses for even a small number of features found. One solution to this problem is to have each processor transfer each remote block of feature coordinates from the other processor's shared memory to its own local memory (Lines 134-144 in Appendix B). This is implemented with a loop of *upc_memget* statements that transfer blocks of shared memory to blocks of local memory. *upc_memget* was used because it is far more efficient than transferring data one element at a time. This amounts to all-to-all communication assuming all processors find at least one key feature but is better than the alternative assuming even a few found features. The execution of two versions of code, one with the transfer of shared memory to local memory and one without, confirms these observations; the code with the transfer executes a lot faster assuming the input image has at least a few feature points. This section of code is complete overhead not required in the serial implementations but necessary to implement the parallel algorithm efficiently.

With a local list of all of the found feature coordinates, the processors begin to execute the same triple nested *for* loop that the serial implementation did with the exception that the iterations are equally distributed amongst all of the processors (lines 145-182 in Appendix B). This is implemented by replacing the outer *for* loop with a simple *upc_forall* loop containing the loop counter as the affinity expression. The global vote counter is handled differently as well; instead of having only one global vote counter per object, each processor now has its own global vote counter in local memory. This is done because there would be too much contention for access to a single counter and would result in a lot of CPU cycles wasted waiting on a lock. Lines 183-194 in Appendix B are added overhead not present in the serial implementation responsible for combining all of the global counters in local memory into a single counter on a single processor (thread *THREADS*-1)

Once all other processors have finished updating the global counter on thread *THREADS*-1, the other threads continue onto the next image while thread *THREADS*-1 prints the object discoveries to the screen (lines 195-202 in Appendix B). When an object is discovered the same message, "Object X found in Image Y," is printed to the screen. At this point, if the program has exhausted its list of input images, the program terminates otherwise it continues with the next image. As mentioned previously, only on one out of *THREADS* iterations does continuing with the next image involve loading the image from file. On the other *THREADS*-1 out of *THREADS* iterations loading the image is not required.

The same design choices important for the serial implementation are important for the parallel implementation. As in the serial implementation, the design choices were left as variables to the program, allowing for program tweaking after testing.

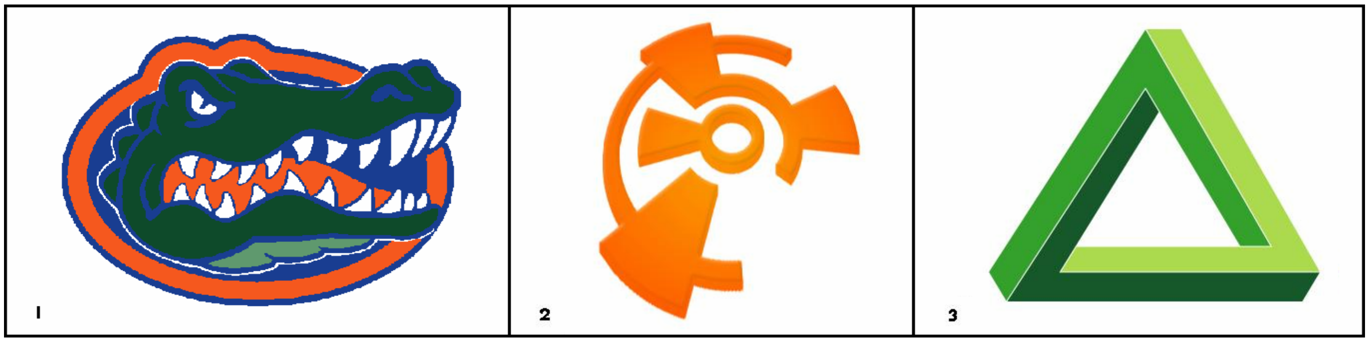

Fig. 8. These are the three target objects searched for in the test batch of images

V. EXPERIMENT

To carry out the performance analysis, the serial and parallel implementations explained in the previous section were executed on the Mu cluster of the HCS lab. This allowed both programs to run on the same AMD 2.0 GHz Opteron processor platform and system architecture. More information on the Mu cluster can be found on the website in [7]. The serial implementation will be compared against the parallel implementation run on a variable number of threads. Also included in our experiment are extra tests that validate several design decisions. The total execution time and accuracy of the algorithm were used as metrics for validation. Additionally, the PPW analysis tool was used to measure the relative performance of the parallel implementation on multiple thread configurations. The results of our experiment will be discussed in the next section.

In order to analyze the performance of our implementations, a test batch of fifty 840x600 2-D 24-bit BMP images was used to detect the three different 2-D target objects shown in Fig. 8. This test batch is a load representative of a realistic input to the system and was kept constant from test to test. The three target objects were placed randomly among fifty images under various transformations where each image contained either all three, a combination of two, just one, or no objects at all. Supported transformations are rotations, translations, and scaling. Also, objects may or may not be partially covered by other objects in an image. Each image contained a random amount of extra target feature points added in an attempt to trick the algorithm into producing false positives and vary the workload produced by each image. This random amount varied from having no extra feature points in certain images to a few images containing a heavy load of extra feature points aside from those detected in the target objects. Each image also contained a mixture of different geometric shapes with dissimilar colors. All images were then saved on the cluster and loaded into the program at runtime.

The reason for choosing the number 840 as the pixel height of the images was because the largest number of threads used in this study is 8. To balance the workload in the parallel implementation, each image is divided horizontally into equal block sizes for each thread running in the particular execution. Using a height of 840 pixels allows the program to evenly divide the block sizes by any number of threads up to eight. The width dimension of 600 pixels was not significant in this study, however, it was arbitrarily chosen as an amount that together with the height would not cause the partitioning of the entire image in the algorithm to exceed the maximum block size.

The intention of using 2-D images in our work, as opposed to 3-D images, was to avoid inaccuracies and misdetections introduced by the azimuthal angle present in 3-D imagery. An object shown in one picture may appear in a second picture at a different angle from the reference plane and vector of the first picture. For the 2-D geometric hashing algorithm to perform accurately, it is important for this angle to remain constant. In our work, attempts to run 2-D image recognition on 3-D images such as license plates or stop signs failed to find any objects at all; a problem attributed to capturing images at different angles.

The following subsections explain the different experimental setups for our work.

A. *Hash Table Population*

The three objects from Fig. 8 were used as inputs to the hash table creation code in Appendix C. This program searched the objects for the most unique features and chose an equally distributed subset. The subset was then used to populate the hash table using the same algorithm used to read it.

B. *Serial Baseline Run*

The serial algorithm was considered the baseline performance metric and used to compare with the performances of the other algorithm executions. This program was executed on one of the processors in the Mu cluster. The total execution time along with gathered profile data from PPW was collected and recorded.

C. *Parallel Run*

The parallel algorithm was executed on the Mu cluster with a varied number of threads ranging from 1 to 8. Each time, the total execution time and profile data from PPW was collected and recorded.

D. Load Balancing Test

To illustrate the importance of load balancing, the performance of our parallel implementation was compared to the performance of an alternative parallel implementation that essentially runs the serial code on multiple threads. Minimal parallelization overhead exists in this alternative implementation, however, the workload in each thread varies depending on the input images fed into the program.

TABLE 1: Total Execution Times

Implementation w/ P Processors	Total Execution Time (s)
Serial - P=1	**203.0**
Parallel - P=1	**276.1**
Parallel - P=2	**181.5**
Parallel - P=3	**134.8**
Parallel - P=4	**100.2**
Parallel - P=5	**81.3**
Parallel - P=6	**72.1**
Parallel - P=7	**61.3**
Parallel - P=8	**54.7**
Serial Distributed - P=8	**160.7**

VI. RESULTS

Before experimentally comparing the performance of the serial and parallel implementations, their accuracies were tested using the test batch of fifty images. After execution of each implementation, the output was used to observe the number of successes and failures. It is considered a success when an object is detected within an image and the object is present as well as when it is not detected when the object is not present. In contrast, it is considered a failure when an object is detected within an image and it is not present as well as when it is not detected within an image and it is present. Since there are three objects to search for, each image processed has the possibility of three successes or failures; with a test bench of fifty images, this leads to a total of 150 possible successes. In our tests, the execution of all implementations reported the same number of successes and failures which totaled 142 successes and 8 failures equivalent to a 94.7% success rate. One of the failures was due to not detecting an image when it was actually present and is because too many of the object feature points were covered by another object. The other 7 out of 8 failures were due to detecting an object when it was not present and can be attributed to an excessively high number of feature points found. Subjectively, we believe it is better to falsely detect an object when it is not present than to miss an object when it is present.

TABLE 2: Speedup

# of Threads	Speedup Over Serial
2	**1.12**
3	**1.51**
4	**2.03**
5	**2.50**
6	**2.82**
7	**3.31**
8	**3.71**

The total execution times for each implementation are shown in Table 1. The execution times in the table show that the parallel code outperforms the serial code as the number of threads increases. Except for the parallel implementation run on a single processor, the execution times of each parallel execution are increasingly lower than the serial implementation time. The best case performance achieved was 54.7 seconds with the parallel implementation executed on 8 processors which is significantly faster than the 203 second time achieved by the serial baseline. The execution of the parallel code run on 1 processor achieved a slower total execution time than the serial execution due to the negative impact of increased overhead due to parallelization without the added benefit of extra processors. It is also important to note that as the number of threads increased, the performance gained due to adding one additional processor decreased. The reason for this is that with each added processor, additional overhead is added but the amount of useful work remains constant. Table 1 also shows the execution time for the alternative parallel implementation that runs the serial code on multiple threads and will be discussed later in this section.

The execution times from Table 1 were used to calculate the relative speedups of the parallel executions using the serial implementation as the baseline. Those calculated values are listed in Table 2. As expected from our previous discussion,

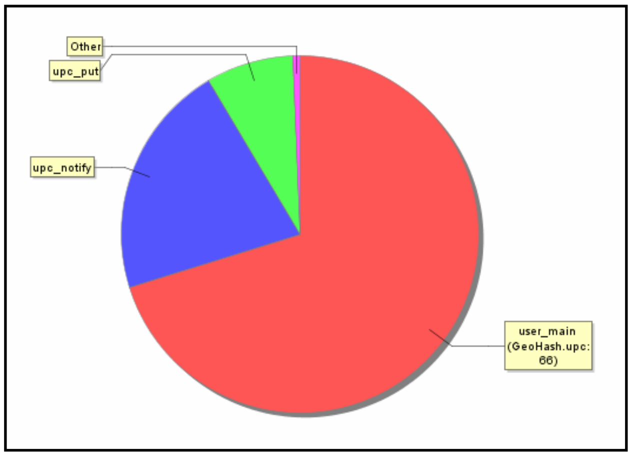

Fig. 9. Pie chart of profile data for the parallel implementation executed on 8 threads.

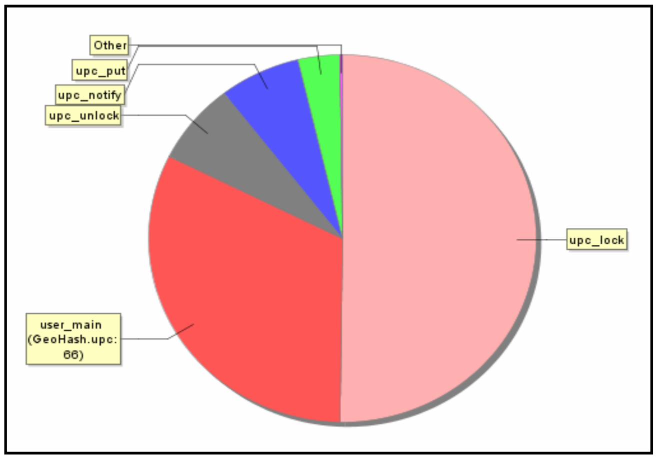

Fig. 10. Pie chart of profile data for an early version of the parallel implementation executed on 8 threads.

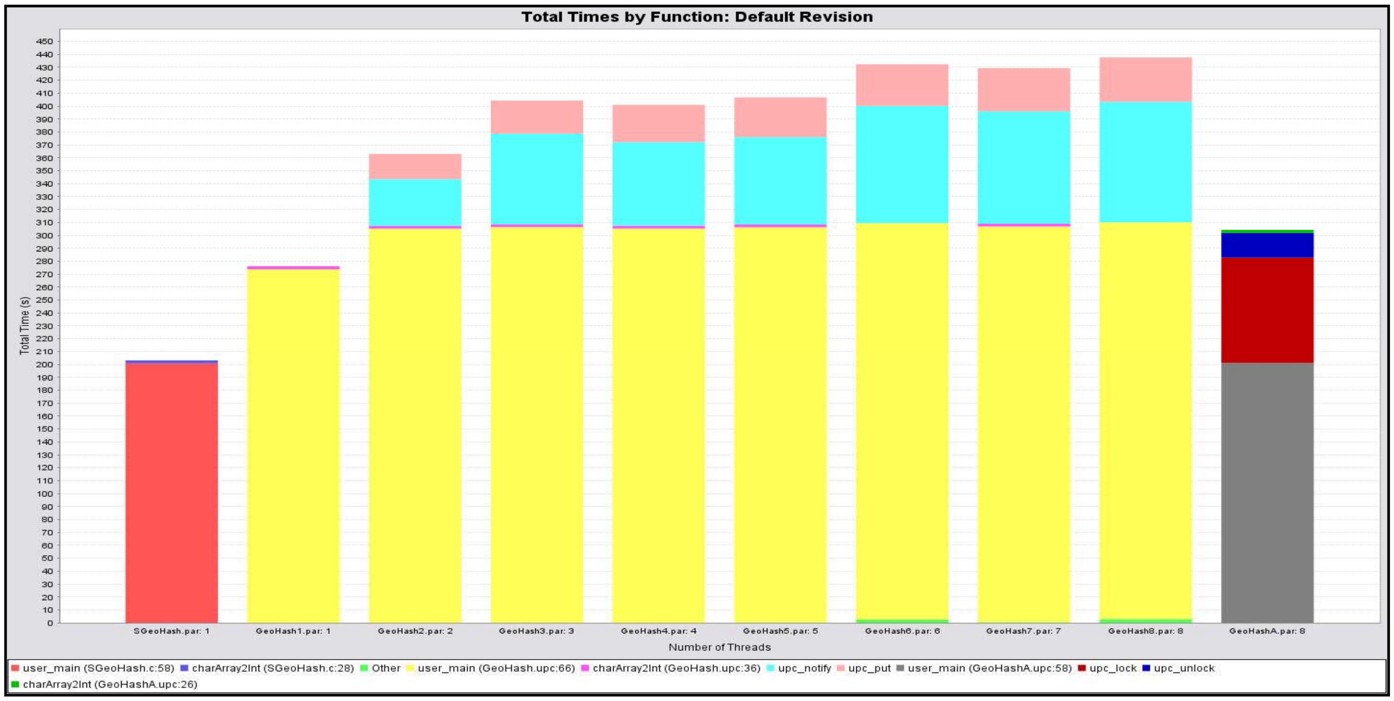

Fig. 11. The bar graph depicting the accumulated execution times of each thread per program execution.

the speedup due to parallelization increases as the number of threads increases but at a decreasing rate.

In addition to collecting the execution times and calculating the speedups, the profile data collected from the Parallel Performance Wizard was used to analyze where time was being spent in each implementation. The pie-chart in Fig. 9 shows a breakdown of the time spent performing certain functions in the final version of the parallel code executed on 8 threads. The green, blue, and small purple portions of the pie represent the amount of time spent taking care of overhead. The green slice corresponds to having a single processor loading images into shared memory. This overhead should be drastically reduced if the use of UPC-IO was possible. The blue slice corresponds to the time wasted implementing barrier synchronizations. The red portion represents the amount of time spent executing useful code. Overhead associated with the purple sliver is time spent doing everything else. Fig.9 clearly shows that approximately 70% of the execution time was spent doing useful work while 30% was spent on overhead.

Profile data from PPW was also used to optimize our original parallel code into the final code used for our tests. Testing of the original implementation showed that a large percentage of time was spent notifying under small loads and an even larger percentage was spent in a lock under high loads. The pie-chart in Fig. 10 corresponds to profile data from one of the first versions of our parallel implementation run with the same number of processors and same load as the execution in Fig. 9. By observing the chart, it can be clearly seen that approximately 57% of the execution time was spent dealing with lock issues. Before running PPW, we did not expect this to be an issue but viewing this information brought

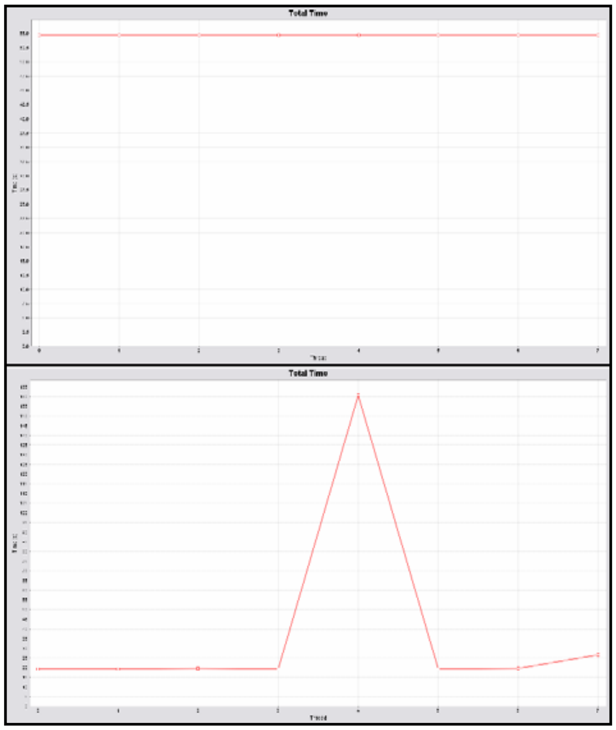

Fig. 12. Graphs showing execution time of each thread for the parallel implemention (top) and alternate parallel implementation (bottom).

the problem to our attention. The resulting modifications in our code led to more efficient use of synchronization and lock statements which improved parallel performance and led us to the final distribution in Fig. 9.

Another feature of PPW allows further analysis of our implementation by accumulating the time spent in each section of the code by each thread and displaying it in an easy to understand bar graph. Fig. 11 displays a comparison between the serial implementation and our parallel implementation run with a varying number of threads. It also shows a comparison between the un-load-balanced implementation that will be discussed in the next paragraph. Overall, it can be observed from Fig. 11 that the amount of useful work remains relatively constant among parallel executions with varying processors while the overhead increases as the amount of threads increases. This however does not mean that the overall execution time increases as the number of threads increases; the sum total of all processor's work increases but the average work per processor decreases, and assuming a balanced workload, leads to a decrease in the total execution time. Our observations from testing echo this.

An imbalance in the workload per thread prevents the execution time from being faster than the thread with the most work. This is illustrated in Fig. 11 with the comparison between the bar of the 8 thread parallel execution and the bar of the alternative parallel 8 thread execution with an unbalanced workload (second to last and last bar respectively). The last bar shows that the sum total of work on each processor for the alternate parallel implementation is less than that of the 8 thread parallel implementation, although the later executed far more quickly. The explanation for this can be seen in Fig. 12. The top graph in Fig. 12 shows that all threads finished executing at approximately the same time while the bottom graph shows that under the same batch of input images, the majority of threads finished their executions at relatively the same time while a single thread continued to execute. The top graph was taken from our parallel implementation and shows an almost perfectly balanced workload across each thread. The bottom graph, taken from the alternate parallel implementation, shows that one processor ended up doing more work than the others. Fig. 11 shows less parallelization overhead exists in this alternative implementation, however, the workload in each thread varies depending on the input images fed into the program and leads to the poorer performance.

VII. CONCLUSIONS AND FUTURE WORK

Applications that require analysis of every image in a stream of incoming images are limited by the speed of the hardware they are utilizing. The example applications mentioned in Section II have the need for fast data processing and that need may exceed the performance capabilities of a single processor. Implementing the object recognition algorithm on a parallel architecture will simply improve the performance; that means the ability to search for a larger number of objects at an increased rate.

We have shown that running the 2-D geometric hashing algorithm on a parallel architecture can provide a speedup of 3.71 over a serial implementation when executing the parallel algorithm on 8 processors. In this study, the test batch of images was kept constant while the number of threads and implementations were varied. As more threads were used in the execution of the parallel algorithm, a decrease in overall execution time was observed. Our results lead towards the conclusion that running the geometric hashing algorithm on a well-balanced, efficiently designed parallel implementation executed on multiple processors performs better than a serial implementation executing on only one.

Potential future work includes modifying the implementations to realize the 3-D geometric hashing algorithm, changing the feature type searched for, and running the current implementations on more of the 8 threads available on to us on the Mu cluster.

VIII. REFERENCES

[1] Wolfson, H.J. and Rigoutsos, I, 1997. Geometric Hashing: An Overview. IEEE Computational Science and Engineering, 4(4), 10-21

[2] Y. Lamdan and H. Wolfson, "Geometric Hashing: A General and Efficient Model-Based Recognition Scheme," Proc. Int'l Conf. Computer Vision, IEEE Computer Society, 1988, pp. 238-249

[3] A. Kalvin et al., "Two-Dimensional Model-Based Boundary Matching Using Footprints," *Int'l j. Robotics Research,* Vol. 5, No. 4, 1986, pp. 38-55

[4] Kiuchi, Yasuhiro and Ozaki, Tomonob, 2007. Partial Geometric Hashing for Retrieving Similar Interaction Protein Using Profile. 4th International Conference on Information Technology, 589-596

[5] Lee, Sunqiu and Moon, Daesung, 2008. Memory-Efficient Fuzzy Fingerprint Vault based on the Geometric Hashing. 2nd International Conference on Information and Security Assurance, 312-315

[6] *Figures 2-5 from:* http://en.wikipedia.org/wiki/Geometric_hashing

[7] *Information on the Mu cluster of the HCS Research Lab at the University of Florida:* http://www.hcs.ufl.edu/lab/mu.php

[8] Riqoustos, I and Hummel, R, "Massively Parallel Model Matching: Geometric Hashing on the Connection Machine," Computer, Vol. 25, No. 2, 1992, pp. 33-42

[9] Khokhar, A., Prasanna, V., and Cho-Li, W., "Scalable Data Parallel Implementations of Object-Recognition on Connection Machine CM-5," Proceedings of the Twenty-Seventh Hawaii International Conference, 1994, pp. 130-139