Inquiry: The University of Arkansas Undergraduate Research Journal

Volume 12 Article 8

Fall 2011

Parallelizing Scale Invariant Feature Transform on a Distributed Memory Cluster

Stanislav Bobovych
University of Arkansas, Fayetteville

PARALLELIZING SCALE INVARIANT FEATURE TRANSFORM ON A DISTRIBUTED MEMORY CLUSTER

By Stanislav Bobovych
Department of Computer Science and Computer Engineering

Faculty Mentor: Amy Apon
Department of Computer Science and Computer Engineering

Abstract

Scale Invariant Feature Transform (SIFT) is a computer vision algorithm that is widely-used to extract features from images. We explored accelerating an existing implementation of this algorithm with message passing in order to analyze large data sets. We successfully tested two approaches to data decomposition in order to parallelize SIFT on a distributed memory cluster.

Introduction

In certain domains, it is very useful to extract information about objects in images. A specific domain, geospatial sciences, is facing the problem of ever increasing high resolution data. Streams of data from satellites, unmanned aerial vehicles, airplanes, and people need to be accurately georeferenced and registered. Using conventional methods, including desktop computers that run serial programs, to analyze this data takes too long or requires more resources than a single desktop contains. Parallel cluster computing provides more resources than a desktop and allows processing of different parts of the problem at the same time. Using parallel processing, it is possible to solve the problem of analyzing large sets of geospatial data.

Manual time-consuming tasks like image mosaicking, stitching, alignment, and matching of geospatial data collected by multiple sensors can be made autonomous by the use of computer vision algorithms such as Scale Invariant Feature Transform (SIFT). These techniques are extensively used in geospatial sciences. Specifically, there exists a need to take an input image from a user, analyze and describe it, and finally match the image to a known location that has been georeferenced. The work presented here is part of a larger project that is building a system that uses computer vision techniques, databases, and algorithms to quickly and autonomously solve certain geospatial science problems like georeferencing and registering new and existing Geospatial Information Systems (GIS) data. The GIS data sets that motivate this parallel implementation are terabytes in size. A single image may be larger than the memory of a single node, hence the need to extract features and descriptors from an image in parallel. Also, as output of data from different sensors increases, the amount of data that needs to be processed in a timely manner will increase.

This article describes ways to implement a distributed memory parallel version of a popular computer vision algorithm Scale Invariant Feature Transform (SIFT) using the Message Passing Interface (MPI) library in order to solve the problem of timely analysis of large GIS data sets for which the original implementation of SIFT was not designed. There have been successful prior parallel implementations of SIFT, but they are geared toward real-time processing of small data whereas this implementation emphasizes scalability and capacity computing.

Background

A number of basic concepts in image processing and in geospatial science are essential to understanding this research project. SIFT is an example of a feature detection and description algorithm. SIFT++ and VLFeat are examples of SIFT implementations. Clusters are a type of parallel architecture used for executing parallel applications, and InfiniBand is a fast interconnect network technology that is typically used in clusters. Message Passing Interface is a programming model. Each of these topics is covered in more detail in the following sections.

SIFT

There are a multitude of feature detection algorithms. [16] The computer vision algorithm SIFT was chosen as the keypoint detection algorithm for this research because it is well known in the scientific community and it provides the best results compared to the computation effort. [6] [10] [3] This algorithm automatically detects and describes interesting features (blobs/regions in high contrast areas) in images. These descriptions are unique, stable with respect to scale, rotation, and translation, and are used in computer vision applications. [9] SIFT is designed to take an input image and output descriptors of unique points, called keypoints, in the image.

The following are the steps in the SIFT algorithm:

1. Scale-space extrema detection: A scale space pyramid is built. Extrema are detected over all scales and image locations. Difference-of-Gaussian function is used to identify potential interest points that are invariant to scale and orientation.

2. Keypoint localization: Once a potential keypoint is found, location and scale are determined. Keypoints are filtered based on their stability. Keypoints in low contrast areas or ones that are poorly localized along an edge are thrown out.

3. Orientation assignment: Keypoints are assigned one or more orientations based on local image gradients. These orientations are used for all future operations. This step allows the generation of descriptions that are invariant to orientation, scale and position.

4. Keypoint descriptor: The local image gradients are measured at the selected scale in the region around each keypoint. These are transformed into a transformation invariant representation. [8] [7]

At the time of the writing of this paper, there are two major serial implementations of the SIFT algorithm, SIFT++ [18] and VLFeat [19]. The first implementation is a C++ implementation of the SIFT algorithm and was designed to be as close as possible to David Lowe's original implementation. VLFeat is a set of computer vision libraries written in C. SIFT++. It was chosen as the base code for this research because it was faster, used less memory and was already used by researchers at University of Arkansas.

There have been previous attempts to parallelize SIFT. Examples include a Graphical Processing Unit (GPU) implementation [13], a Field Programmable Field Array (FPGA) implementation [1], and a multi-threaded implementation [20]. The GPU implementation cited here achieves 10x speedup over the optimized CPU implementation. The multi-threaded implementation yields a speedup of 2x when using eight processors. Also, [20] explores GPU acceleration of SIFT with offloading the Gaussian convolution to the GPU. The particular part of the code was accelerated by a factor of 13, but the total execution time of the application was accelerated by a factor of 1.9. Another highly optimized multi-threaded implementation [21] was able to achieve an average of 6.4x speedup.

Most of the effort in accelerating SIFT has been in the real-time computer vision domain. This subject area deals with small images, for example 640x480 images streamed at 30 frames per second. This kind of processing does not stress the memory architecture since the data is so small. However, once the scale space generated of an image can no longer be held in a cache, memory bandwidth and memory size become the limiting factors in performance of an application. These solutions cannot be used to solve the geospatial domain problems.

Technology

The Star of Arkansas at the Arkansas High Performance Computing Center and Ranger at the Texas Advanced Computing Center were used in this research. Each system is described briefly.

The system used for development and testing was the Star of Arkansas. This cluster consists of 157 Symmetric Multi-Processing (SMP) compute nodes. Each node contains dual quad-core Xeon E5430 processors, 2x6MB cache, running at 2.66GHz with 1333 MHz FSB. Each core has 2 GB of main memory. The theoretical peak performance of Star is 13.36 teraflops (13.36 × 1012 floating point operations per second).

The network interconnect on the Star of Arkansas is InfiniBand and runs at 10 Gbps. The cluster is interconnected with an additional Gigabit Ethernet network for NFS access, and another Gigabit Ethernet network for management.

The Star of Arkansas has NFS and Lustre file systems. The NFS file system is used for permanent storage and is 4 TB. The Lustre file system resides on Data Direct Networks storage, is used for fast temporary storage, and is 21 TB. Lustre is an open source distributed parallel file system for high performance cluster computing. [11] A Lustre system is composed of file system clients which access the file system, object storage servers (OSS) which provide file I/O service and metadata servers (MDS), which manage the names and directories in the file system. All of this is transparent to applications which access the file system using normal POSIX semantics. [2]

After initial development and testing, TACC's Ranger system was used to conduct large-scale tests. This cluster consists of 3,936 SMP compute nodes. Each node contains four AMD Opteron Quad-Core 64-bit processors (16 total), running at 2.3GHz with 1.0 GHz HyperTransport system Bus, and 2 channels with 667 MHz DDR2 DIMMS. Each processor has 64 KB of L1 cache, 4x512 KB L2 Cache, and 2 MB of on-die (shared) L3 Cache. Each node has 32 GB of main memory. The theoretical peak performance of Ranger is 579.4 teraflops (579:41012 floating point operations per second). The interconnect topology is a 7-stage, full-CLOS fat tree with two large Sun InfiniBand Datacenter switches at the core of the fabric (each switch can support up to a maximum of 3,456 SDR InfiniBand ports). [12]

InfiniBand is a switched communications link with high throughput, low latency, quality of service and failover, and scalability. Applications use InfiniBand as a messaging service. It is used for storage, Inter Process Communication (IPC) or any other communication between the application and its environment. This is different from the byte-stream oriented TCP/IP/Ethernet, which works on transporting bytes of information between application sockets and requires the operating system to move bytes from the program's virtual buffer space, to the kernel's network stack and finally onto the wire. InfiniBand does not request the operating system for access to communication resources. Applications access the InfiniBand messaging service directly. [5]

Message Passing Interface (MPI) is an Application Programming Interface (API) that allows communication between processes using a message passing paradigm. [14] It is used to create scalable high performance parallel applications. Processes can reside on the same machine or on multiple machines in a cluster, and communicate through explicit messages. This is unlike the shared-memory paradigm, where threads communicate using shared buffers and have symmetric memory access to memory. [15]

Methodology

We have discussed the need for a fast and scalable implementation of SIFT that can be used in geospatial science. In order to avoid duplicating work, an existing implementation, SIFT++ by Andrea Vedaldi of University of British Columbia, was used as base code. [17] In the computer vision community, this is a well known open source implementation. [20] This implementation, compiled into a binary called sift, was analyzed for hot spots and memory usage; different parallelization implementations using this base code were tested. The goal was to reduce the overall runtime of the application while generating the same results as the serial implementation.

Performance Metrics

There are many ways to measure performance. One can measure the wall time or the system/user time of an application, latency, response time, rate of integer or floating point operations, or the efficiency of an application. [4] The chosen metrics have to be relevant and meaningful within the application's domain and have to be accepted by the users in that domain.

In the domain of geospatial science and the problems this specific application is trying to solve, three metrics are of most concern. The first is the wall time of the application. Scientists in the field are willing to tolerate the delays between asking a question and getting an answer anywhere from a few seconds to a few hours, so reducing the wall time of an application is important. There is a distinction between wall time and run time. Wall time incorporates the I/O, operating system jitter, and the actual work done by the application. In this paper, wall time and run time are used synonymously. The second metric closely tied to the first is the speedup over the serial implementation. Speed up is the serial run time divided by the parallelized run time. This is the way to measure if the effort and resources spent to make the application run faster were worth it. The final metric is the accuracy of the results because bad answers that are generated quickly are not useful to domain users. The output of the parallelized application has to match the serial version's output.

Single Node Performance

SIFT++ is both memory intensive and computationally expensive. Scale space generation's computation time is deterministic. This process uses a great deal of memory since the scale space is generated once and all of it is stored in memory throughout the life of the application. Analyzing the code, the memory usage by the scale space pyramid is:

$$Memory\ usage = 4\ bytes * l \sum_{i=s}^{m} \frac{w * h}{2^{2i}} a$$

The parameters in this equation are: final octave m, first octave s, current octave i, number of levels per octave l. Using a 800x640 image with standard parameters as an example, generation of scale space is 25% of computation and takes up 62.5 MB of memory.

During testing, the serial implementation of SIFT failed to analyze a 9600 x 7200 image, on a Star of Arkansas node, a system with 16 GB of RAM. The domain space uses images of this size and greater. Besides the serial implementation failing to process large images, given the right parameters this implementation would fail to process relatively small images. The problem of single node memory exhaustion had to be overcome.

As shown in Figure 1, the computation time is highly dependent on the number of keypoints found in an image. The number of keypoints is dependent on the objects in the image and the size of the image. Once all of the keypoints are found, the majority of the computation is spent calculating descriptors for these keypoints.

Parallelization Strategy

In this application, time and memory are the constraining factors when processing data on single nodes. Previous attempts at parallelizing SIFT involved speeding up specific parts of the algorithm using fine grained parallelism. [20] In this implementation, the whole application is being made to run faster using high level data parallelism. Instead of focusing on making a particular part of the algorithm fast, the data used by the application is divided between multiple instances of the application. One approach is to simply partition the image into horizontal slices and distribute pieces of the image among nodes. Each node uses SIFT to process the data and outputs a description of the image slice. The output from the nodes is aggregated to form a final description of the whole image. The second approach partitions the image into blocks. The number and the size of the blocks depends on the dimensions of the image and the number of nodes used to process the image. Each block is processed by different nodes in parallel and the descriptions are aggregated. The parallelization strategy is outlined in the following steps:

1. Partition (decompose) the image into smaller pieces.
2. Either send each piece to a different node, or have each node read a different piece of the image directly from the file system.
3. Compute SIFT descriptors on each piece of the image.
4. Aggregate the descriptors.

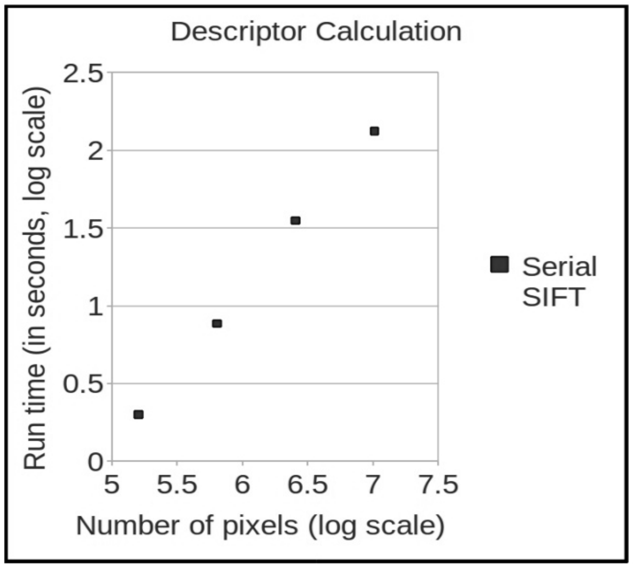

Figure 1. Keypoint descriptor calculation run times.

Implementation

Due to memory limitation of single nodes, the solution was to reduce the memory footprint of the data on each node. This was done by partitioning the image into smaller pieces and sending each piece to a different compute node in the cluster. Each node then ran SIFT on its piece, computed the descriptors, adjusted the coordinates of the descriptors, and output the descriptors. Finally, the descriptors were aggregated into a single descriptor file that is useful to other applications and scientists.

Row-Wise Decomposition

The first attempt at splitting the image was to slice the image horizontally. The image was divided between the nodes; each node received a small slice of the original image. The root node loaded the entire image into memory, then scattered different pieces to different nodes. Each node was responsible for adjusting the x,y coordinates of the keypoints based on its rank. At first, the MPI code was directly integrated with the original SIFT++ code to accomplish this. This approach was simple and generated results in which data loss due to boundary effects was between 1% and 3%. However, this approach was not the most efficient, since certain images can be partitioned in a better way. Before moving on to the next approach, the code was rewritten. Most of the MPI code was transferred to a decomposition driver and the SIFT driver was made into a library function that was called from the MPI driver. The SIFT driver accepted command line arguments with which the program was started, an image buffer, process rank, x and y offsets. This generic SIFT driver allowed easy decomposition driver swapping.

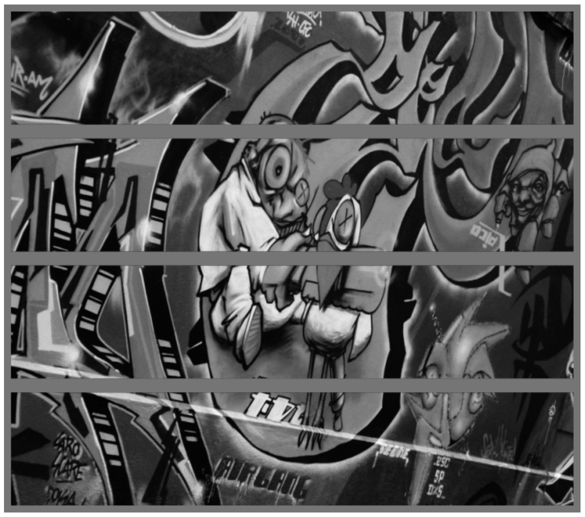
Figure 2. Row-wise decomposition.

Block-wise Decomposition

The second attempt at partitioning the image was to use block decomposition. The image was divided into equally sized blocks, and each block was sent to a node for processing. Each node adjusted the x,y coordinates of keypoints and output the data to a file. This approach was particularly challenging because of how the data is organized. The data was stored in an image format called Netpbm. After the image was loaded into memory, it is stored as a one dimensional array of floats. To properly stride through the data, various MPI mechanisms were used.

Block decomposition was achieved in a first implementation in the following way. First, the original image was loaded into memory by the master process. The height and width of the image were broadcast to all nodes. The master process then calculated the proper dimensions of an individual partition of the original image. The partition dimensions were broadcast to all processes.

A two dimensional Cartesian Communicator was created. The sizes of the dimensions were determined by the ratios between the original width and height, and the partition width and height. The MPI communicator was non periodic and reordering was not allowed. Every process allocated a buffer that contained a partition of the original image.

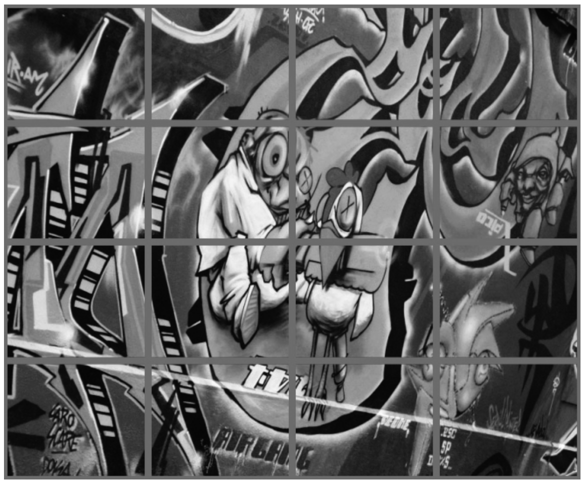
Figure 3. Block-wise decomposition.

A new MPI data type was created so that the original image could be easily split between different processes. The rows of a partition of the original image can be thought of as blocks, the pixels in each row as block elements, and the spacing between pixel rows as the block spacing. A vector that contained the number of blocks, the number of elements in a block and the block spacing was created. A struct was created to hold the vector. Offsets for each image partition were calculated. The original image was then scattered to the Cartesian communicator using the calculated offsets and the new data type as the type. Each process then worked on its portion of the image and output descriptors. The x,y coordinates were adjusted based on the Cartesian coordinates of the process. Later, block-wise decomposition was reimplemented using the driver paradigm described in the row-wise decomposition section.

Decomposition Using Parallel I/O

It was also possible to exploit parallelism in data access and storage. In the first two implementations, the master node read the image and distributed different pieces to different processes using MPI communication. In the third implementation, each process read different portions of the image in parallel using MPI I/O.

The first attempt at using MPI I/O was to implement row-wise decomposition. Each process read the header of the image file, calculated appropriate file offsets, set the file view and read a portion of the image into a buffer. Then each process used the SIFT driver on the buffer. In a later implementation of the row-wise I/O partitioning, only the master process read the header of the image file and determined the header offset. Once that was known, the master thread broadcast the width of the partition, height of the partition, and an offset to all the processes. Each process then created an appropriately sized buffer, set its file view, read its portion of the image using MPI I/O, and executed the SIFT algorithm on its portion of the data.

Block-wise decomposition was accomplished in a similar fashion to the row-wise decomposition. The master process read the image, extracted height and width, and broadcast the information. Each process in turn calculated the block dimensions and created a distributed array. The distributed array was used to create an MPI filetype, which in turn was used to set the file view for each process. Each process then read its portion of the file, stored the data in a buffer, and executed the SIFT algorithm on that buffer.

Experiments and Analysis

Three experiments were set up to test the parallel implementations. The first experiment involved analyzing a sequence of images with SIFT. The sequence consisted of differently sized random pieces of the same geospatial image. The reason for using random pieces is that SIFT's computation time is dependent on the number of keypoints found in an image. Using the same image and upscaling it to create larger images would be an unrealistic test since in the geospatial domain larger images should contain a larger number of interesting features than smaller images.

The second experiment analyzed the correctness of the output of the parallel implementations by comparing their output with the serial version's output. The first and second experiments used the serial, row-wise in-memory decomposition, block-wise in-memory decomposition, row-wise I/O decomposition and block-wise I/O decomposition. The final experiment was SIFTing an actual geospatial image on a TeraGrid resource, Ranger.

Experiment 1 – Run time

Data parallelization successfully reduced the runtime of SIFT. The comparison of serial and parallel implementations can be seen in Table 1. The runtime of the serial implementation increases as the number of pixels in the images increases. In comparison, the runtime of the parallel implementations increases at a slower rate compared to the runtime of the serial implementation. Both decomposition methods achieved significant performance improvements over the serial version.

The average speed-up was 19.5x, with row-wise parallel IO decomposition achieving a speed-up of 20.18. The superlinear speed-up was attributed to the fact that the parallel implementations were able to utilize memory bandwidth better than the serial version by keeping a larger portion of the data in cache. Row-wise decomposition in memory and parallel I/O were slightly better than the block-wise decomposition. This may be due to the fact that C stores arrays in memory in row-major format. Rows of data were accessed more efficiently than columns, since access by rows of data accesses contiguous memory regions. The block-wise decomposition, as with column-major access, requires a number of accesses to memory that were not contiguous. Contiguous accesses to memory have high Central Processing Unit (CPU) cache hit rate, allowing the CPU to fetch data from the cache. Non-contiguous access to memory generates CPU cache misses, requiring the CPU to access main memory, which is slow compared to accessing cache.

Experiment 2 – Correctness

The results of the second experiment are shown in Figure 4. The parallel implementations had data loss due to boundary effects between the partitions of the original data. The figure shows that the block decomposition had less loss on all of the images tested, by more than a factor of two for all images tested.

The data loss may not be a problem, since domain images generate millions of keypoints. With increasing image size, the ratio of keypoints to lost keypoints decreases. Block decomposition created partitions that have smaller perimeters as a function of the partition area than row decomposition.

The loss of keypoints was due to how SIFT finds and filters keypoints. Keypoints found on borders of an image tend to be rejected. Also, the descriptions of a keypoint in the original image and in the fragment were different since different neighborhoods were used for the description. The loss of keypoints and differences in descriptors on image partitions are collectively called edge effects.

Experiment 3 – Scalability

The last experiment was SIFTing an actual geospatial image. A 116987x11005, 1.2 GB image, was SIFTed on Ranger, a TeraGrid resource, using in-memory block decomposition. Data are shown in Table 2.

Row-wise decomposition was attempted, but failed due to lack of sufficient memory on the nodes. MPI I/O was not used because it is not supported on Ranger. In all the trials, a single process ran on a single node. This was to maximize memory availability for each process. This particular implementation scaled well when the number of cores/nodes increased. Increasing the node count reduced the memory usage per node, yielding even better speed-up.

Conclusion and Future Work

Data parallelization of SIFT on a distributed memory cluster is a viable way to find interesting features in geospatial images. Block-wise partitioning scheme is shown to scale well. MPI constructs and advanced communication functions are well suited to accomplish this task. MPI I/O makes the implementation of block-wise and row-wise decomposition methods easier than in-memory block decomposition; it is also faster than in-memory decomposition. Edge effects in large images are almost negligible.

The results from this research suggest several directions for future work. Specific lines of inquiry include memory exhaustion, edge effects, and descriptor aggregation.

If the image is of sufficient size, partitioning the image into pieces and sending the pieces to nodes will fail if during processing of partitions, node memory is exhausted. To solve this problem, the maximum partition size has to be determined before image partitioning. The image then needs to be partitioned in such a way that the maximum partition size is not exceeded. If the

Table 1. Experiment 1 run times (in seconds).

Pixels	Serial time	Row-wise run time	Row-wise IO time	Block-wise run time	Block-wise IO time
160000	1	0.66	0.66	0.66	0.66
640000	6	0.33	0.33	0.33	0.33
2560000	23.3	2	2	2	2
10240000	114	6.33	6	7.33	7
40960000	622	26.66	26.66	28.33	27.66
69120000	848	45.33	44.33	47.66	47
Avg. speedup	1	19.85	20.18	18.7	19.07

number of partitions in this scheme is larger than the number of processing nodes, partitions should be added to a work queue and submitted to be processed in batches or on demand basis.

Keypoints that lie along the edge of an image tend to be filtered out and do not appear in the final solution. Since all of the mentioned partition schemes generate image edges, keypoints are lost. Overlaps between partitions will fix this problem. The overlaps would generate redundant keypoints that will have to be filtered out.

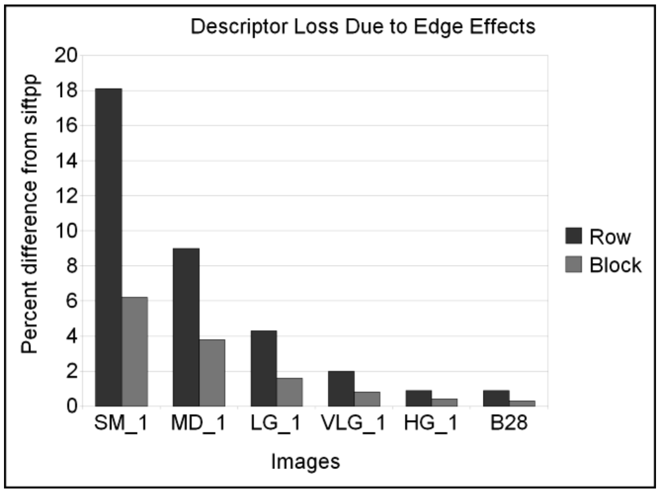

Figure 4. Experiment 2: Image size increases from left to right.

Currently each process writes its descriptors to its own file, and at the end the files are concatenated to generate the final descriptor file. It may be possible to have all the processes write their results to a single file. Both of these additions would utilize a fast file system and take load off the master node.

Table 2. Experiment 3 run times and memory usage.

Core count	Run time (in seconds)	Scale space memory usage per core (in gigabytes)
32	444	11048.16
64	233	5523.32
128	159	2761
256	90	1380.45
512	67	689.96

References

[1] Cristina Cabani and W. James MacLean. 2006. A Proposed Pipelined-Architecture for FPGA-Based Affine-Invariant Feature Detectors. In *Proceedings of the 2006 Conference on Computer Vision and Pattern Recognition Workshop* (CVPRW ‘06). IEEE Computer Society, Washington, DC, USA, 121-. DOI=10.1109/ CVPRW.2006.19 http://dx.doi.org/10.1109/CVPRW.2006.19

[2] High-performance storage architecture and scalable cluster file system. White paper, Sun Microsystems, 2007. Available online (20 pages).

[3] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. Surf: Speeded up robust features. In In ECCV, pages 404–417, 2006.

[4] Jeffrey J. Evans, Cynthia S. Hood, and William D. Gropp. Exploring the relationship between parallel application run-time variability and network performance in clusters. Local Computer Networks, Annual IEEE Conference on, 0:538, 2003.

[5] Paul Grun. Introduction to infiniband for end users. White paper, InfiniBand Trade Association, 2010. Available online (54 pages).

[6] Asaad Hakeem, Roberto Vezzani, Mubarak Shah, and Rita Cucchiara. 2006. Estimating Geospatial Trajectory of a Moving Camera. In *Proceedings of the 18th International Conference on Pattern Recognition - Volume 02* (ICPR ‘06), Vol. 2. IEEE Computer Society, Washington, DC, USA, 82-87. DOI=10.1109/ ICPR.2006.499 http://dx.doi.org/10.1109/ICPR.2006.499

[7] David G. Lowe. Method and apparatus for identifying scale invariant features in an image and use of same for locating an object in an image. U.S. Patent #6711293 issued

03/23/2004, March.

[8] David G. Lowe. Distinctive image features from scale-invariant keypoints. Int. J. Comput. Vision, 60:91–110, November 2004.

[9] D.G. Lowe. Object recognition from local scale-invariant features. In Computer Vision, 1999. The Proceedings of the Seventh IEEE International Conference on, volume 2, pages 1150 –1157 vol.2, 1999. 31

[10] Krystian Mikolajczyk and Cordelia Schmid. A performance evaluation of local descriptors. IEEE Transactions on Pattern Analysis & Machine Intelligence, 27(10):1615–1630, 2005.

[11] The University of Arkansas. Hardware. http://hpc.uark.edu/ hpc/about/hardware.html

[12] The University of Texas at Austin. Ranger user guide. http:// services.tacc.utexas.edu/ index.php/ranger-user-guide.

[13] Sudipta N. Sinha, Jan michael Frahm, Marc Pollefeys, and Yakup Genc. Gpu-based video feature tracking and matching. Technical report, In Workshop on Edge Computing Using New Commodity Architectures, 2006.

[14] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J Dongarra. MPI: The complete reference. MIT Press, Cambridge, MA, 1996.

[15] Sayantan Sur, Matthew J. Koop, and Dhabaleswar K. Panda. High-performance and scalable MPI over infiniband with reduced memory usage: an in-depth performance analysis. In Proceedings of the 2006 ACM/IEEE conference on Supercomputing, SC ’06, New York, NY, USA, 2006. ACM.

[16] Tinne Tuytelaars and Krystian Mikolajczyk. Local Invariant Feature Detectors: A

Survey. Now Publishers Inc., Hanover, MA, USA, 2008.

[17] A. Vedaldi. An open implementation of the SIFT detector and descriptor. Technical Report 070012, UCLA CSD, 2007.

[18] A. Vedaldi. Sift++ source code and documentation, 2009. Available online.

[19] A. Vedaldi and B. Fulkerson. Vlfeat: An open and portable library of computer vision algorithms, 2008. Available online.

[20] Seth Warn, Wesley Emeneker, Jackson Cothren, and Amy W. Apon. Accelerating sift on parallel architectures. In CLUSTER, pages 1–4. IEEE, 2009. 32

[21] Qi Zhang, Yurong Chen, Yimin Zhang, and Yinlong Xu. Sift implementation and optimization for multi-core systems. Parallel and Distributed Processing Symposium, International, 0:1–8, 2008. 33

Mentor Comments: Professor Amy Apon highlights the importance of Stan's work in numerous current and future applications and notes that it is unusual to see such difficult research taken on by an undergraduate.

Image registration takes two or more images and aligns them so that they form a single, larger image. Image registration is an important problem in many areas of research that utilize image analysis, including medical applications, computer vision, and geospatial processing. In the area of geospatial processing there is a need to align and overlay images from a wide variety of sources, including satellite and aerial images. Registration is difficult to do for many of the very large images that are available since the memory required to execute the registration algorithm is very large, and writing partial results to disk storage during execution can increase the runtime of the application by two orders of magnitude. While there are a few very large memory computers that can perform registration on very large images, these computers are still very expensive and uncommon.

The goal of this project is to parallelize the Scale Invariant Feature Transform (SIFT) application, the most commonly used algorithm that is used to do image registration, in order to make possible the alignment of very large images, such as those that come from satellites. The approach uses distributed memory computing on a commodity supercomputing cluster. The developed code uses open source software libraries to divide the images into smaller pieces, distribute them across the memories of the different computers, perform the registration, and then recombine the result. There have been other recent examples of the parallelization of SIFT and this is one of the most effective seen in the literature for this type of problem. Two variations of the developed parallel SIFT application were tested on the Star of Arkansas supercomputer and on a national TeraGrid supercomputer. The techniques are very efficient and result in less than 2% data loss. In addition, the application was shown to scale very well to very large images and to a large number of processors.

Parallelizing applications to run on a supercomputer is very complex and difficult to do well. This development of the scalable parallel SIFT application is a great accomplishment for an undergraduate. An earlier version of this work was presented as a poster at the annual Supercomputing conference, SC10, in November, 2010, a mark of accomplishment of this research.