A Parallel Implementation of a Correspondence-Finder for Uncalibrated Stereo Image Pairs

Brendon Cahoon, Sharad Singhai, Glen Weaver, Eric Wright
Department of Computer Science
University of Massachusetts
Amherst, MA 01003-4610

Technical Report 97-13

October 17, 1997

Abstract

We report on our experience with parallelizing a computer vision algorithm. The algorithm employs low-level image processing techniques which are relatively easy to parallelize and intermediate-level computer vision techniques which lack the regularity and locality of image processing algorithms. The application is an excellent candidate for use as a benchmark. We implement two parallel versions of this algorithm; the second one based on our experience with the first version. We program the parallel implementation in the Single Program, Multiple Data (SPMD) model using the MPI message passing interface.

We evaluate our implementation on a four node IBM SP. Our results show excellent speedup numbers for the image processing portion and good speedups for most of the application. However, part of the application is inherently sequential. Our second parallel implementation is not only more efficient than the first, but it also has better speedup numbers. In addition, we suggest changes to our final parallel implementation that should improve its performance.

1 Introduction

General benchmarks such as SPEC95 and PERFECT Club provide rough comparisons of the performance of different machines. These performance measurements, however, may not represent the true relative performance for a particular application because general benchmarks are not likely to have the same performance characteristics as programs from the application domain. Moreover, the results of popular benchmarks may be misleading because vendors sometimes tune their compilers and machines to perform well. Instead, when machines are considered for a specific application domain, an application based benchmark, such as Linpack and IU Benchmark, which consists of representative tasks from the application domain should be used to compare the relative performance of machines.

A succession of benchmarks have been developed for the computer vision domain [Pre86, UPLD86, Ros87, WRHR91]. The earlier benchmarks tended to include very simple tasks and provide only vague guidelines about how each task was to be accomplished. Hence, the benchmarks did not stress the machines or accurately represent their performance on a full vision application, and the results were hard to compare because vendors could drastically rewrite the code to make their machine look better. The image understanding benchmark was developed to represent a broader range of vision processing, discourage excessive tuning, and provide more challenging tasks [WRHR91]. Our experience with these previous benchmarks along with the continued advance of hardware capabilities suggest that a new benchmark is needed. The new benchmark should include more intermediate-level vision tasks to better represent such code and be more computationally demanding to better exercise the capabilities of hardware.

This paper describes our parallelization of David Wren's image correspondence matching algorithm, called *correspondence finder*. For two images with overlapping content, an image correspondence algorithm locates common points of interest in the images. Correspondence matching is a broadly useful vision task and incorporates both low- and intermediate-level vision tasks.

We implement two serial versions of *correspondence finder* and port both serial implementations to the MPI message passing interface (Section 3). We present several experiments comparing the performance of each implementation on a four node IBM SP and evaluate the overall success of our parallelization effort (Section 4). We also include analysis of shortcomings and suggestions for possible improvements to our implementations (Section 5). Finally, in Section 6 we summarize our experience and lessons from our implementation effort.

2 The *Correspondence Finder* Application

Computer systems that use computer vision and interact with the physical environment may require three dimensional knowledge of the environment. For example, a mobile robot needs to know the proximity and height of obstacles, and an assembly line inspection system may need to know the precise location of defective products so that it can coordinate their removal. Stereo techniques can provide accurate depth knowledge by using multiple observations of the same scene from slightly different angles and the epipolar constraint to extract depth information [ZDFL94]. This approach reflects the capabilities of the human visual system which uses the slight disparity between what each eye sees to extract depth information about nearby objects.

Computer vision systems may obtain stereo information in one of two ways. The first approach, which is most often associated with the term stereo, is to simultaneously use multiple cameras to take images of the same scene. This configuration closely reflects the configuration of the human visual system. The other approach is to use multiple images from the same camera, while moving the camera between images. This configuration is appropriate for environments that do not change much or for use in moving vehicles.

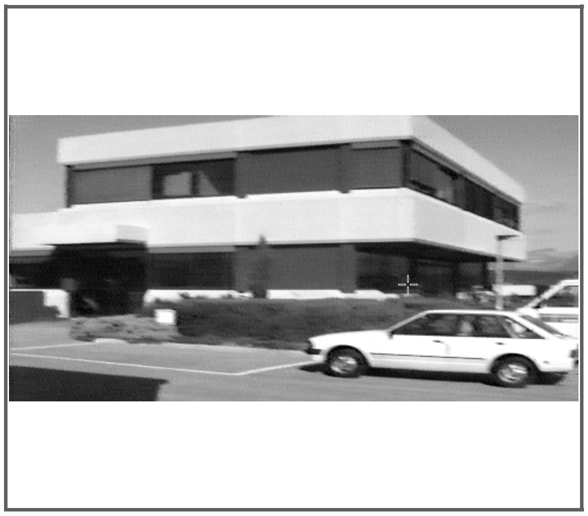
Using epipolar geometry requires precise knowledge of the differences between the cameras which acquire the images. In some cases, such as a roving vehicle, precise measurements are not available. In these cases, vision systems must discover the difference between the cameras. Once approach to discovering these differences is to find objects that appear in both the left and right images. For example, the same car appears in both images in Figure 1.

However, identifying high-level objects such as cars and buildings is difficult. Therefore typical vision systems use lower level image features such as corners, lines, and regions. After image features have been extracted from each image, we must decide which feature in the left image corresponds to which feature in the right image. This task is the responsibility of correspondence finder algorithms which often use heuristic constraints, such as proximity, to determine correspondence. Finally when correspondences are established, this knowledge may be used to extract stereo information about the scene.

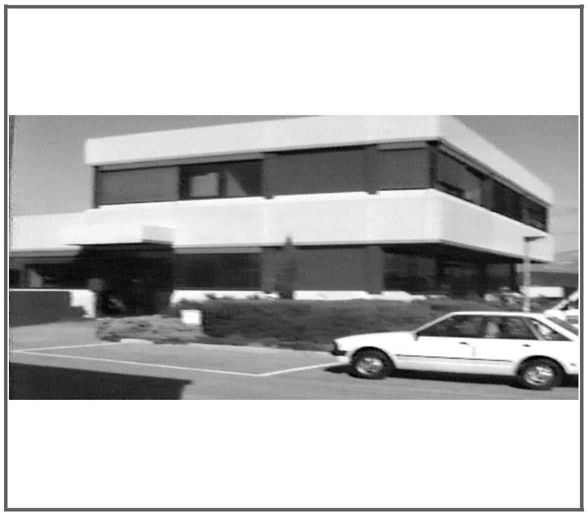
The *correspondence finder* application performs the first two stages of the stereo extraction process: identifying low-level image features and discovering correspondences between these features from a pair of images.

2.1 Algorithm Description

We use a low- and intermediate- level vision application called *correspondence finder*. Our application is adopted from David Wren's algorithm [Wre96] which in turn is based on an algorithm from INRIA [ZDFL94].



A. Office Left Image



B. Office Right Image

Figure 1: The office stereo image pair.

The *correspondence finder* algorithm uses three levels of abstractions of image data: pixels, corners (also called points [1]), and matches. Each abstraction has an associated data structure. An image is a two dimensional array of pixels. A corner (or point) contains information about a single point of high curvature such as its position, intensity, and neighbors. A match[2] is a set of two points, one from each image and represents the possible correspondence of these points. A match data structure also contains other information such as strength, unambiguity, and supporting matches for the given match.

Correspondence finder has three phases, one for each abstraction. The image phase operates on pixels and extracts points of high curvature. The point phase operates on these corners and produces a list of potential matches. The match phase evaluates the strength of each match and uses a relaxation algorithm to choose the best matches.

A high-level description of *correspondence finder* steps is shown in Figure 2 where module names are in parenthesis following the individual steps. The functional diagrams are depicted in Figures 3–5. The boxes in the functional diagram represent both individual stages in the computation and separate source files. Figure 3 is the top level flow graph for the *correspondence finder* algorithm. The Input box represents the acquisition of image data, and the Output box represents passing the final matches to the external environment. Figures 4 and 5 expand the Corner and Match boxes (in Figure 3), respectively.

2.1.1 The Image Phase

The image phase extracts points of high curvature from the image through a series of convolutions (Figure 4 and Corner in Figure 3). Our implementation uses the Kitchen-Rosenfeld corner detector [Wre96] which applies five different 5×5 convolutions to a smoothed image (a 3×3 convolution). The image phase thresholds the results of the convolutions and combines them into a corner measure. The algorithm selects points with both minimum and maximum "cornerness" in a 3×3 neighborhood as corners. Module AppendTks[3] merely combines the points of minimum and maximum corners into a single list.

[1]Throughout this paper corners are often referred to simply as points.

[2]We use 'match' to refer to either the match phase or match data structure. Which one is being referred to will be clear from the context.

[3]AppendTks is not present in all our implementations.

Image phase

1. Find corners in images (Corner)

Point phase

2. Find neighboring points for each point (PtNbrs)
3. Find candidate matches for each point (CandMatch)

Match phase

4. Calculate correlation between points in each match (PtMatch)
5. Remove matches with poor correlation
6. Identify supporting match pairs (SupMatch)
7. Calculate support score for each supporting match pair (SupScore)
8. Calculate strength of each match (CalcStrength)
9. Select best matches and discard inconsistent matches (Update)

Figure 2: High level overview of *correspondence finder*

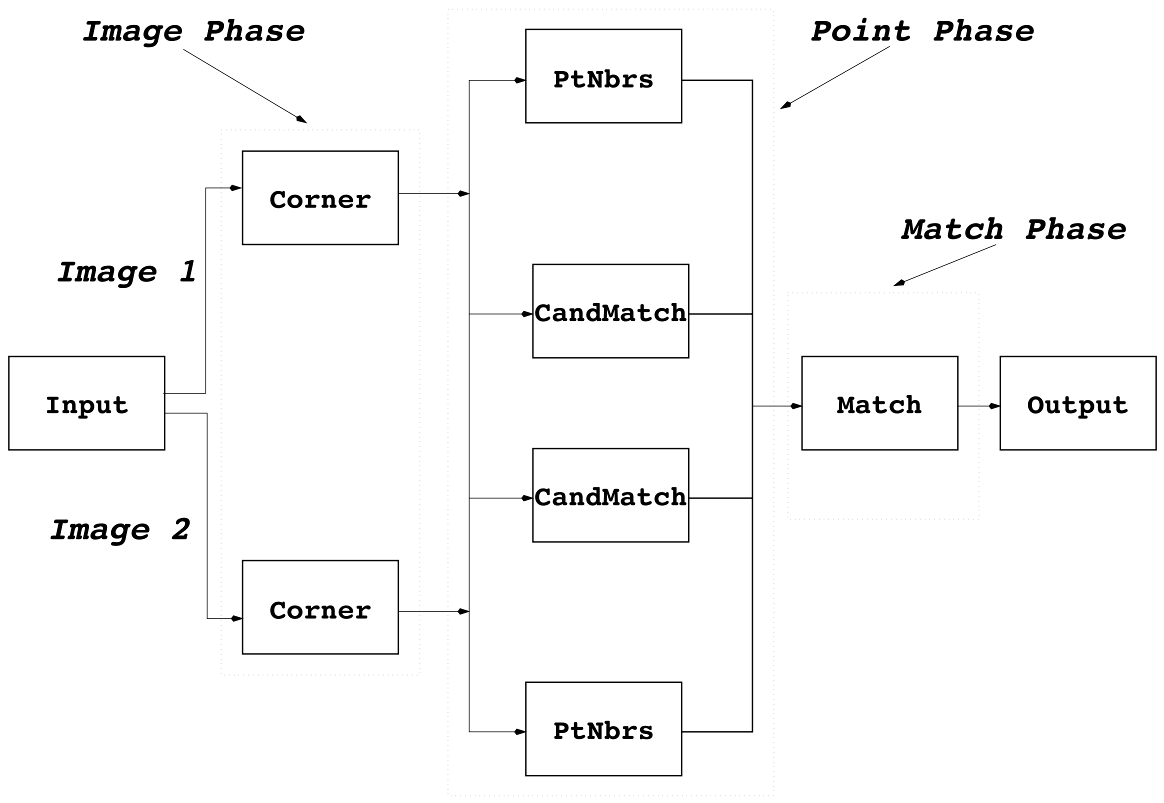


Figure 3: Functional diagram for *correspondence finder* algorithm.

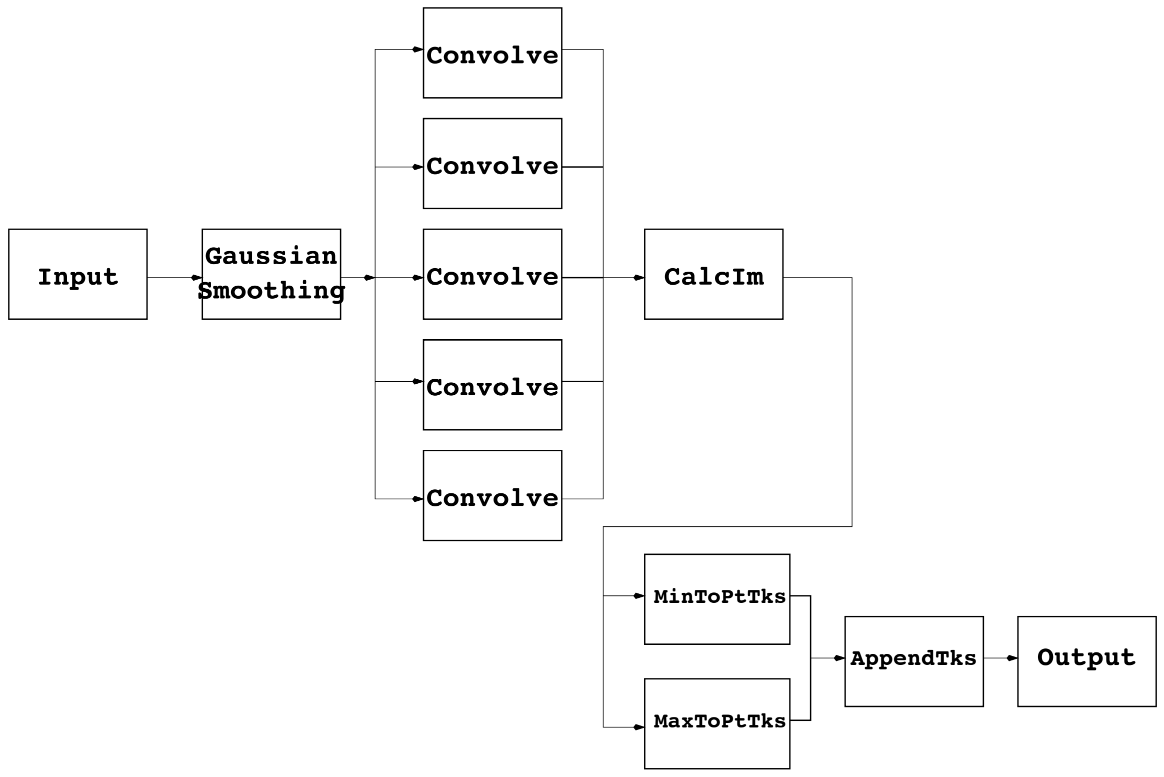


Figure 4: Functional diagram for image phase of *correspondence finder* algorithm.

2.1.2 The Point Phase

The point phase establishes two types of relationships between points (middle portion of Figure 3). In step 2 (module PtNbrs), the algorithm finds each point's neighbors by collecting all points within some distance of the current point. The distance we use is a quarter of the length of the image's x dimension. For a square image, this implies that points from up to a quarter of the image will be counted as a point's neighbors. Step 3 (module CandMatch) builds a list of possible corresponding points from the other image by collecting all points within some distance of the current point's coordinates (no conversion of the coordinates is performed). The distance we use is a half of each of the image's dimensions which implies that points will be used from up to half of the other image.

2.1.3 The Match Phase

The match phase determines which of the possible matches are valid (Figure 5). Module PtMatches performs steps 4 and 5 by building match records using the list of candidate points from step 3, and filtering potential matches by comparing the underlying pixel data for each point. The filtering step assures that corresponding points actually appear similar in the images. For each match, step 6 (module GetSupMatch) identifies supporting matches. A supporting match is a match that if valid would suggest that the current match is also valid. If point A is right above point B in the left image and point A has been matched with point C in the second image, then one would expect to find point B's corresponding point below point C. Step 7 (module GetSupScore) calculates the amount of support each supporting match offers to the current match. Finally, step 9 iterates over modules CalcStrength and Update using a relaxation algorithm to select the final matches. Module CalcStrength tallies the strength of a match while avoiding using a single point in multiple matches. Update uses a *winner-takes-all* or *some-winners-take-all* algorithm to iterate

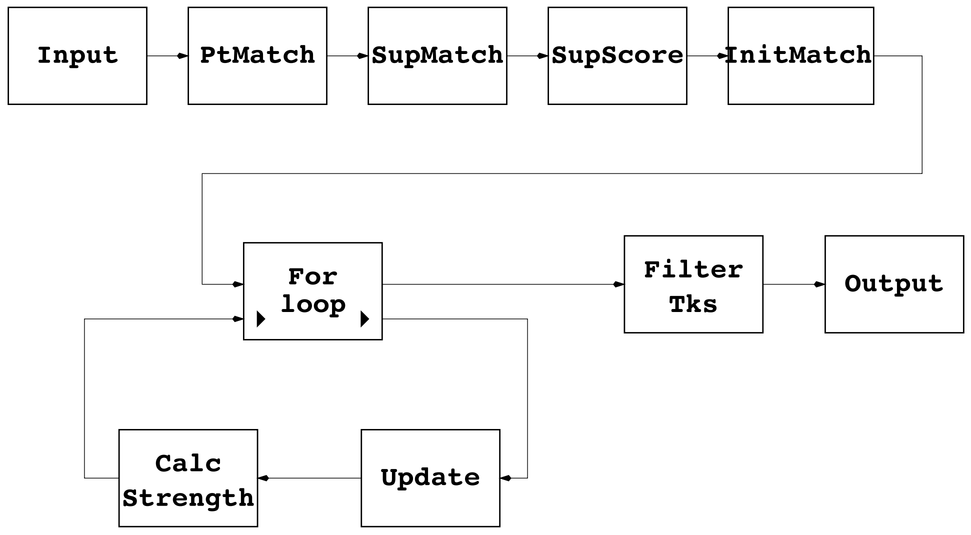


Figure 5: Functional diagram for match phase of *correspondence finder* algorithm.

over the matches and selects the best matches in each pass and deletes the matches conflicting with the selected ones. Update uses two distinct criteria to select matches, match strength and match unambiguity. If both the end points of a match are involved in no other higher strength match then the match is termed *unambiguous*. Using two criteria, Update avoids choosing ambiguous matches with a slightly higher strength over unambiguous but lower strength matches. The relaxation algorithm takes 4-6 passes for typical images. Finally, module FilterTks outputs the selected matches.

3 Design and Implementation of the *Correspondence Finder*Algorithm

In this section, we discuss the design of the *correspondence finder* algorithm and a message passing parallel implementation using MPI on a four-node IBM SP. Section 3.1 discusses the design of our parallel algorithm using Foster's methodical design [Fos95]. We implement four versions of of the *correspondence finder* algorithm: two serial and two parallel. The serial version using index lists is our first attempt and we describe the implementation in Section 3.2. We discuss the corresponding parallel version using index lists in Section 3.3. Both initial versions suffer from various inefficiencies and we redesign the implementation in order to improve performance. In the process we learn valuable lessons which we describe in Section 5. We discuss the second serial and the second parallel versions in Section 3.4 and Section 3.5, respectively.

We start with a version of *correspondence finder* developed at Amerinex Applied Imaging, Inc. (AAI) that uses the KBVision system [Wre96]. KBVision is a proprietary development environment for image applications [Ame]. KBVision has a set of library routines for manipulating pixels, corners and images. We eliminate all the code which relies on KBVision and implement our own data structures[4].

3.1 Design of Parallel *Correspondence Finder*

We discuss the design of our parallel *correspondence finder* algorithm using Foster's four-stage methodical design process [Fos95]. The four stages (called PCAM) are:

Partitioning Decompose the problem into small tasks.

[4]The process of converting the AAI algorithm to C code which still uses the KBVision library routines was performed by Dawn Werner [Wer96].

Communication Determine communication requirements between the tasks

Agglomeration Combine tasks to improve performance

Mapping Assign tasks to processors

In the following sections, we discuss each of stages in more detail.

3.1.1 Partitioning

We partition using a data decomposition (also known as domain decomposition). The algorithm uses three different data structures: images, points, and matches. For the three data structures, the finest granularity of data decomposition is on a pixel, point, and match basis. That is, we can treat each individual element independently and create a task for each element.

It is possible to use a functional decomposition for parts of *correspondence finder*. For example, Figures 3 and 4 illustrate that we can use a functional decomposition for modules Corner, Convolve, PtNbrs, and CandMatch. However, we did not implement any functional decomposition since it is difficult to achieve using MPI.

3.1.2 Communication

The communication patterns depend upon the data element type.

Pixels
For the image data, each task needs to communicate with its neighboring tasks. The amount of communication depends upon the size of our convolution masks. Our largest mask is 5×5 which means that each task must communicate with two adjacent neighbors to the north, east, south, and west. We perform 6 convolutions (we implement Gaussian Smoothing as a convolution).

Points
During *correspondence finder*, each point needs to obtain copies of neighboring points in the same image and in the other image. We perform this communication during the point phase and at the beginning of that match phase. The amount of communication depends upon the number of neighboring points, NP. For the same image, we look in a window with a radius of $X/4$ (where X is the width of the image) around a given point. For the other image, we look in a window with a radius of $X/2$. In the point phase, we obtain the neighboring points in 4 places, twice in PtNbrs and twice in CandMatch, resulting in $4 \times NP$ communication operations. We also need to perform a global communication operation in order to group all the points for later phases. In the match phase, we look at the number of neighboring points in PtNbrs and GetSupMatch. The amount of communication is $2 \times NP$.

Matches
For each match we calculate a score for its supporting matches. The number of supporting matches depends upon the number of matches that are close by in the image (we explain the algorithm in Section 2). We only compute the score once which results in NM communication steps for each point, where NM is the number of neighboring matches for each match. We need a global communication step to sort all matches by their supporting scores. The communication pattern in the final step of *correspondence finder* is very complex. Each match consists of two points. For each match, we obtain the matches that each point is involved in. For example, if a match has point 10 and point 30, but point 10 is also involved in 5 other matches. Our algorithm iterates over each match several times and requires communication each time.

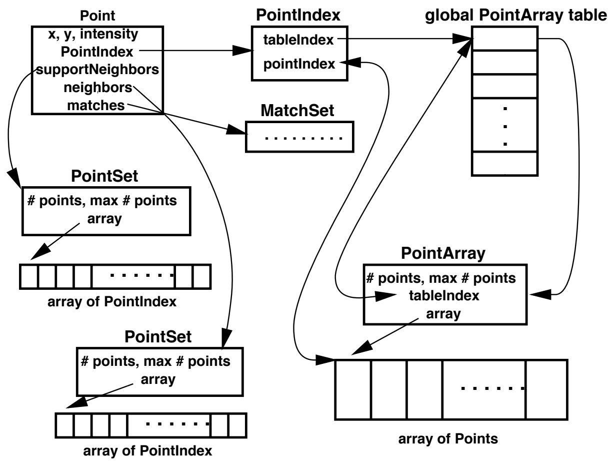


Figure 6: Points Data Structure

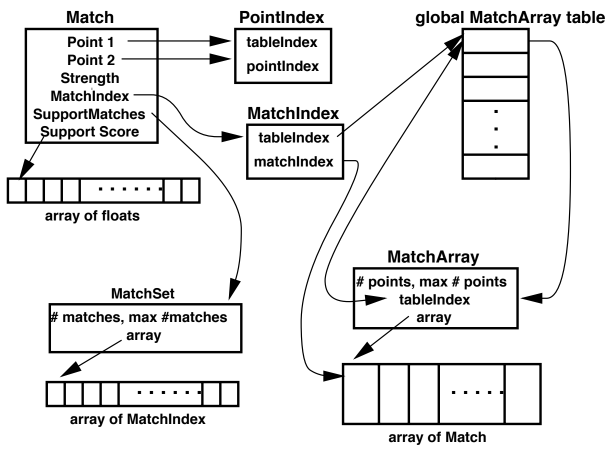


Figure 7: Match Data Structure

3.1.3 Agglomeration

We need to agglomerate pixels, points, and matches to reduce communication. For pixels, the most natural grouping is to agglomerate by rows in the image. On a four processor machine, we divide the image into 4 parts. Using this agglomeration, the only communication involves the pixels on the borders. For points and matches, the easiest agglomeration is to group the points and matches that are created on the same processor. For example, we agglomerate all the points created on processor 2. Our results show that this is not the most efficient agglomeration since the points and matches end up unevenly distributed among the processors. Each point and match is involved in communication with many other points and matches so this agglomeration does not have any effect on communication.

3.1.4 Mapping

Mapping is simple. We allocate each image agglomeration to a different processor.

3.2 The Index List Version– Serial *(Serial1)*

Our initial version of *correspondence finder* eliminates all references to the KBV library routines. We designed a set of data structures to manage points and matches. Dawn Werner started this process by defining a data structure for images [Wer96]. Our design of the data structures for the serial version takes into account our goal of producing a corresponding parallel version.

3.2.1 Point and Match Data Structure

Figures 6 and 7 illustrate the point and match data structures for the serial version. The two data structures are very similar. We represent an individual POINT or MATCH as a structure, or record. We have different data structures to represent sets of points and matches. In Figures 6 and 7, the sets are POINTSET and MATCHSET. In the following discussion, we describe the point structure in more detail. The match data structure is similar (in most cases just replace POINT with MATCH). The difference between the two data structures is the actual fields which make up a POINT and MATCH.

The significant part of the actual point token set is the array of POINTINDEX. The POINTINDEX array represents the points in a token set. A POINTINDEX is a pointer into the collection of points. It is not essential to use index values to represent a token set in the serial version, but we use index values instead of actual pointers to facilitate parallelization. We represent the complete collection of points for a single image using a POINTARRAY. A POINTARRAY is an array of POINT structures. There is a single copy of each POINT in the POINTARRAY, but multiple token sets may reference a single POINT. We use the POINTARRAY table to maintain separate collections of points. We create a separate collection for the points found in image 1 and the points found in image 2. The `tableindex` field of MATCHINDEX indicates the appropriate collection.

It is important to note the interaction between the point and match data structures. Figure 6 shows that a point contains a list of matches (*i.e.*, a match token set). In Figure 7, we see that a match consists of two points. The interaction is not significant for the serial version, but it has significant impact on parallelization strategy.

3.2.2 Evaluation of Serial Version

Our serial version does not fully work on several images. For some images, the program fails due to a memory allocation.

Parts of our serial version run more quickly than the original KBV version. We believe the performance improvements are due to our point and match token set library routines. We only implement the required token set functionality for our application. The KBV libraries are more general and more complex. However, our code does not always run more quickly. In particular, the phase which identifies supporting matches runs very slowly for images containing a large number of matches. The inefficiency is from the union and intersection operations we perform on the match token set. Using our data structure, union takes $O(n)$ time and intersection takes $O(n^2)$ time.

3.3 The Index List Version– Parallel *(Parallel1)*

We base our parallel version of *correspondence finder* on the serial version from the preceding section. We use MPI to handle communication. Our parallel algorithm follows the SPMD programming model. In the SPMD model, each processor runs the same program, but on different data. As we previously mentioned, the algorithm operates on three types of data structures: images, points, and matches. We discuss our parallel implementation in terms of these data structures.

3.3.1 Image Phase

The image phase is responsible for reading two images from disk and finding significant points, or corners, on each image using a series of convolutions[5].

We divide the image equally among the processors. The image size that each processor reads in and operates on is $1/P^{th}$ the total image size (where P is the number of processors). We partition the image by row (*e.g.*, on a system with 4 processors, processor 1 contains the top 25 rows of an image with 100 rows). The image phase requires little communication. Each processor sends n rows to the neighboring processors, where n depends upon the convolution mask size. Our algorithm uses 3×3 and 5×5 convolution masks. Note that processor 1 and processor P only have 1 neighboring processor. For the 3×3 mask, each processor sends 1 row and for the 5×5 mask, each processor send 2 rows. The communication occurs three times during the image phase since we update the image pixel values and then continue to perform calculations.

[5]This part of the algorithm was already parallelized by Dawn Werner.

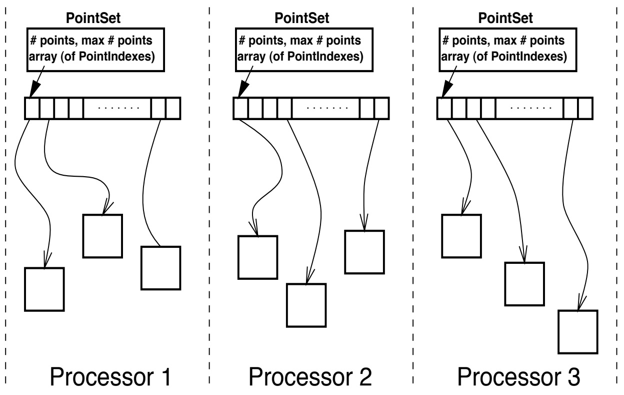


Figure 8: Diagram of a distributed point set.

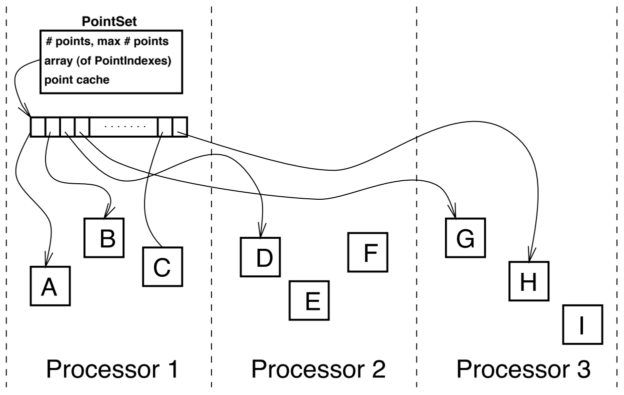


Figure 9: Diagram of a local point set.

3.3.2 Point Phase

The point phase finds neighboring points within an image and in the other image. We maintain points in point sets which come in two flavors: distributed and local. Both types of point sets have their constituent points spread across the processors. The difference occurs in where they store the knowledge of which points are members of the point set. As shown in Figure 8, a distributed point set has only partial knowledge about which points are its members on any one processor. A local point set (Figure 9) has this information on a single processor.

We use the *owner computes rule* for changing fields of a point record. Any processor may request a copy of a point record, but only the processor which owns the point is allowed to update the record. The owner computes rule works well up until the points' matches field must be updated. When a match is created, the matches field must be updated for both component points, but these points may reside on different processors. This problem limited our parallelization effort on this implementation.

Points reside on (and are owned by) the processor on which they were created (during the image phase). Hence, the distribution of points depends on the distribution of the image and where points are found in the image. Operations over points are performed by traversing a point set. For distributed point sets, each processor works on their local points[6] independently. For local point sets, the processor must collect the points in the point set which reside on other processors. Therefore, communication is in the form of requests for remote point records and the subsequent fulfillment of these requests.

For each point set, we send each processor a single request for all the remote points that it owns. The retrieved points are maintained as part of the point set. Hence, any computation which uses a point set first collects all the remote points (Figure 10), computes on its local copy of the points (but may only update those points which the processor actually owns), and finally deletes its local copies of remote points. The local copies of remote points are deleted in lieu of a more sophisticated coherency protocol.

Evaluation

This design emphasizes scalability by ensuring that memory requirements per processor decreases as the number of processors increases. Hence, copies of remote information are not maintained longer than necessary, and point sets are only as large as necessary. However, our strategy has several problems. The first and most significant is that the owner computes rule is insufficient because the creation of a single match may trigger the update of points on two separate processors. Second, the SPMD model communication must be entirely regular, otherwise one is forced to use loops and barriers to ensure that all communication is

[6]Note that a local point is different from a local point set. A local point is one that is owned by the processor which is accessing it. The opposite of a local point is a remote point.

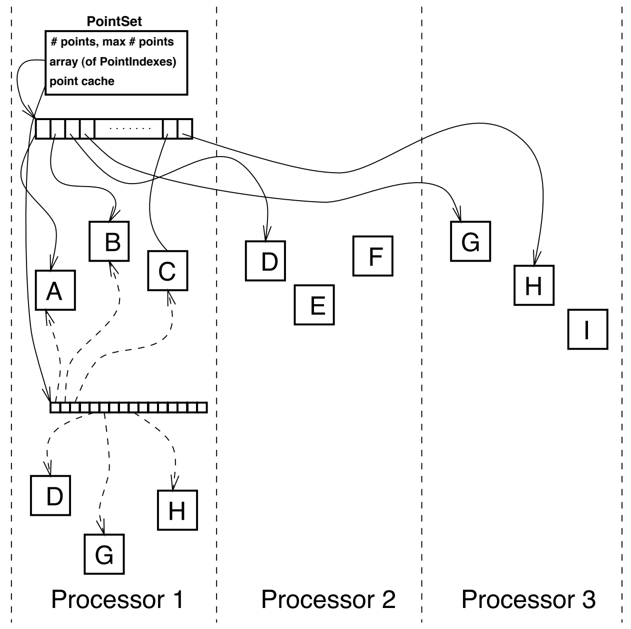


Figure 10: Diagram of a local point set with copies of remote points.

complete. The algorithm structures computation in the match phase as a traversal over a distributed point set (*i.e.*, all the points in an image) with traversals over local point sets nested inside. Hence, communication is predictable if all the processors have exactly the same number of local points in the distributed point set (which is typically not the case). Augmenting *correspondence finder* with the appropriate loops and barriers overly complicates the code. Finally, because communication is performed on a per point set basis and a single point is a member of many point sets, the same point is likely to be communicated to the same processor many times which leads to substantially wasted communication.

3.3.3 Match Phase

The match phases consists of several steps (see Figure 5 or steps 4–9 in Section 2). In our parallel version, we only implement the step which creates the matches from the set of points (called PtMatch).

PtMatch iterates over each point from one of the images. Each processor operates on the points created by the processor in the point phase. For example, if processor 1 creates 40 points in the point phase, then PtMatch iterates over 40 points on processor 1. For each point, PtMatch also iterates over the neighboring points from the other image[7]. However, the processor may not have local copies of the neighboring points so communication must occur to obtain the points. At the beginning of each iteration of PtMatch, we need to communicate with the other processors to obtain local copies of the neighboring points (Figures 8, 9, and 10 illustrate the parallel point data structure).

PtMatch also operates on a small portion of each image which is centered around each point (a radius of 7 pixels). Recall that each processor only maintains part of the image. For local points, we need to communicate with neighboring processors to obtain rows on the border. However, for non-local points we need to communicate with the other processors to obtain the whole window.

Our implementation sends parts of image 1 and image 2 to the other processors. For image 1, each processor sends the top 6 rows to the previous processor and bottom 6 rows to the next processor. The

[7]The set of neighboring points are computed during the point phase.

communication for image 2 is more complex. Each processor sends all of its portion of image 2 to the other processors. The number of processors is $\lceil numrows/numtasks/radius \rceil$, where $radius$ is the number of pixels in which a neighbor point may be found. In our implementation, we end up copying the image to all the other processors since the radius is so large[8].

Evaluation

As we mentioned, we have only parallelized the PtMatch module. Unfortunately, this module does not work correctly. We did not continue with parallelizing the rest of the match phase since we found that this parallelization strategy does not work. We address the problems in *Serial1* and *Parallel1* by redesigning *correspondence finder*.

One problem we address in our second parallel version is that PtMatch needs to read part of the images in order to compute matches. This implementation relies upon a lot of unnecessary communication to pass portions of the image to all the processors. One solution is to have each processor read the entire image (but each processor only operates on part of the image during the image phase).

3.4 The Bit Vector Version– Serial *(Serial2)*

As noted in Section 3.2, our first implementation has several problems. In our first version, operations such as union and intersection are very inefficient. Also, our first parallelization strategy is not very efficient. We design and implement a second version to address these problems. We use bit vectors to improve the performance of the union and intersection operations and to save space. We also implement decentralized point and match data structures which facilitate parallel operations. We discuss our new decentralized data structures in Section 3.5.1.

3.5 The Bit Vector Version– Parallel Decentralized*(Parallel2)*

In this section, we discuss the implementation of our second parallel version. We base the parallel version on the serial version, *Serial2*. This parallel implementation addresses some of the inefficiencies in the first version. The main change is that we implement decentralized data structures for the points and matches. We discuss the decentralized data structures in more detail below.

One benefit of the bit vector representation is that we have been able to eliminate the append tokens part of the image phase (*i.e.*, the last box, AppendTks, in Figure 4). Although we eliminate append tokens in the serial version, the parallel version benefits the most from this change. Eliminating AppendTks improves overall performance because the code is sequential and has a negative impact on speedup. Another benefit is that we save heap space and reduce the amount of data involved in communication operations.

3.5.1 Decentralized Data Structures

Based upon our experience from our first parallel version, we rarely need to pass an entire point or match to another processor. The decentralized data structures enable the processors to pass only essential information to other processors. In our first parallel version, if a processor needed to send only one field from the point data structure to another processor, it had to send the whole point. We have broken up the point and match data structures so that we only send the essential parts.

Figure 11 illustrates the decentralized data structure for points. We only show a few of the fields that we associate with points. We maintain a similar structure for the matches. As we have mentioned, we represent token sets as bit vectors. Our program creates many different instances of the token sets so the space saving is substantial because we only need to create a bit vector for each token set. The "fields" representing a

[8]The radius is half the image.

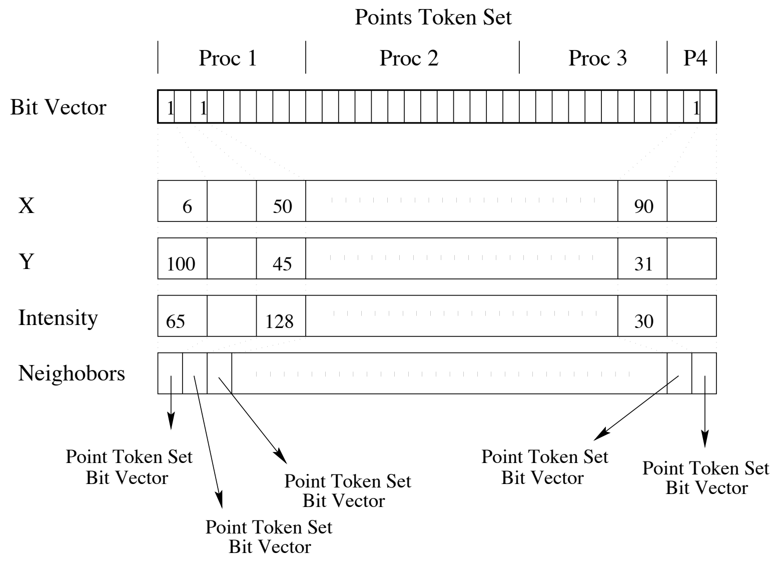

Points Token Set
Proc 1
Proc 2
Proc 3
P4
Bit Vector
1
1
1
X
6
50
90
Y
100
45
31
Intensity
65
128
30
Neighobors
Point Token Set Bit Vector
Point Token Set Bit Vector
Point Token Set Bit Vector
Point Token Set Bit Vector
Point Token Set Bit Vector

Figure 11: Decentralized Data Structure

point are the other structures in Figure 11 (*i.e.*, X, Y, Intensity, and Neighbors). We represent each field as an array, where the size of each array is the number of points or matches. We use the position in the bit vector to index into the field arrays. For example, if bit number 10 is 1, then the X and Y values for the point are X[10] and Y[10].

The fields of the data structures are independent which means that we can manipulate the X field without worrying about the other fields. The independence property is useful when we need to pass information among the processors. For example, we can send the neighbors field to other processors without having to send any other field associated with a point. This reduces the amount of communication the algorithm performs during runtime.

3.5.2 Image Phase

A significant change in the image processing phase is that each processor reads the complete image. However, as with our first version, each processor only operates on $1/P^{th}$ of the image. Having each processor maintain the whole image reduces the complexity in the match phase.

Reading the whole image only slightly reduces the amount of communication during the image phase. We save the initial communication step which sends the rows on the border to the other processors. However, Gaussian Smoothing updates the pixels in the image and each processor must send the new values to the neighboring processors. Both versions of the parallel algorithm perform this update.

We have an interesting implementation detail to note in this phase. We initially used `MPI Send()` to send data to other processor. For large images, our program halts in the `MPI Send()` call while passing messages in the corner detection code. The reason the program halts is because the buffers in `MPI Send()` become full. We solve this problem using `MPI_Bsend()` which uses a programmer allocated buffer. Another possible solution is to use `MPI_Isend()`, the non-blocking send routine.

3.5.3 Point Phase

The point phase consists of two routines and the program calls each routine twice, once for each image. For each processor, both routines iterate over its set of local points (*i.e.*, the points that were computed on that processor). If the points are evenly distributed among the processors, then each processor operates on P/C points and performs $1/P^{th}$ the work compared to the serial version, where C is the number of points (corners) and P is the number of processors. In practice, the points are not evenly distributed and the amount of work each processor performs is uneven. At the end of the points phase, each processor copies its neighbor and the support neighbor information to the other processors using two `MPI Allgatherv()` calls.

3.5.4 Match Phase

We parallelize most of the match phase (steps 4–7 in Figure 2, but we did not parallelize the final two steps in the algorithm which are responsible for calculating the strength of each match, selecting the best matches, and discarding inconsistent matches (steps 8 and 9 in Figure 2 and the `for` loop in Figure 5). We believe this part of the algorithm will not benefit from parallelization since it is inherently sequential. In the rest of this section, we discuss the details of the parts we parallelized.

The initial step of the match phase iterates over the set of local points and creates matches. For each local point, the algorithm iterates over the set of neighboring points in the other images. The point phase computes the set of neighboring points. In order for our algorithm to work, we must have a local copy of every point from image 2. After the point phase computes the set of local points, it also copies the points to the other processors using `MPI_Allgatherv()`.

Similar to the serial algorithm, each processor preallocates space for the match token set. We combine the bit vectors created by each processor after this initial step using the `MPI Allgatherv()` command. A match consists of a point from image 1, a point from image 2, and a correlation value. We issue three `MPI_Allgatherv()` commands; one for each point and one for the correlation value. After the `MPI_Allgatherv()` commands complete, each processor contains a copy of the entire match data structure. We make a copy on each processor for the later phases in the algorithm.

One change in the parallel version is that we create a match and associate the match with two corresponding points during two separate passes. The serial version is able to do this in a single pass; create the match, assign the match to point 1, and assign the match to point 2. Unfortunately, on the parallel version, another processor may own point 2. Recall that our parallel program follows the *owner computes rule* which means that a processor cannot alter a point or match that it does not own (*i.e.*, a processor only changes points and matches that it created). We overcome this limitation by updating the match field for each point after we copy the match data structure to each processor. After we update each point's match field, we copy the match field information to each of the processors using `MPI Allgatherv()`.

Identifying supporting match pairs and computing a cost for each pair (steps 6 and 7) requires that we iterate over the set of local matches (*i.e.*, the set of matches created by a processor). Similar to the point phase, if the matches are evenly distributed among the processors, then each processor operates on M/P matches, where M is the number of matches and P is the number of processors. Our experiments show that, in general, the matches are not evenly distributed which means each processor performs a different amount of work.

The algorithm to compute supporting match pairs iterates over the supporting neighbors for each local match. We compute the supporting neighbors in the point phase (step 2), and we copy the information to the other processors. We copy the support match pair data to the other processors so we can compute the supporting match scores. The algorithm to compute the support match score iterates over the set of supporting matches for each local match. The parallel code for computing supporting match scores is straightforward and does not require any new communication. By this stage, each processor maintains local copies of each point.

Image	Size	Points (left)	Points (right)	Matches
Baballe	768x576	34	33	41
SmRub	768x576	180	142	147
Sport	768x576	117	105	116
Hpair	230x260	306	298	203
Inria	512x512	356	563	680
Office	(l) 512x257 (r) 512x256	307	306	962
Pair	(l) 386x644 (r) 403x630	1015	657	359

Table 1: Test Images

4 Experiments

In this section, we evaluate the performance of *correspondence finder*. We have a test suite of seven image pairs (see Appendix A). Our test suite contains images with varying size and content. Our image sizes range from 59800 to 442368 pixels per image (see Table 1). The image pairs *Hpair* and *Office* have a small displacement between the cameras that took the image. On the other hand, *SmRub*, *Sport*, and *inria* have a relatively large displacements between cameras. The image pair *Baballe* has an object in one image that is not in the other as well as a large displacement. The *Pair* image pair has a small displacement, but the images are of different sizes and brightness.

Table 1 lists the 7 images we use to evaluate *correspondence finder*. The name of each image is in Column 1. Column 2 lists the width and height of each image. Note that the left and right images for Office and Pair are different sizes. Columns 3 and 4 show the number of points (corners) we generate for the left and right images, respectively. Finally, we list the number of matches in Column 5.

We run our experiments on a four-node IBM SP. We run our experiments over relatively idle processors, however, we expect small perturbations in execution times compared to single user mode.

4.1 Execution Times

Tables 2–4 list the wall-clock execution times for three of our implementations: the serial bit vector implementation (serial2), the parallel index list implementation (parallel1), and the parallel bit vector implementation (parallel2).

Table 2 lists the wall-clock execution times for our serial bit vector implementation (Serial2). The total execution time is not a simple function of any one parameter. Corner dominates the time for larger image pairs, but only if they have few points and matches. Images with the most points and matches have the longest execution times. Our smallest images, *Hpair*, have the lowest execution times despite having a relatively large number of points. The *Hpair* images have the same number of points as *Office* but save time in PtMatch and the final matching phase because *Hpair* has far fewer matches. Hence, execution time depends on the number of matches. Yet, the *Office* images have nearly three times the matches as the *pair* images, but *Office*'s execution time is only 27% that of *Pair*'s. The difference between these image pairs occurs in PtMatch and is attributable to the difference in the number of points. Hence, the number of points is also an important factor in the overall execution time. The *Inria* images have a high number of points and matches and achieve the second highest execution time.

Our first parallel implementation only correctly executes until the CandMatch module. Hence, Table 3 lists the wall-clock execution time up to and including CandMatch. The times in this table are dominated by the time to read images (ReadImg) and identify corners (Corner).

Image	ReadImg	Corner	PtNbr	CandMatch	PtMatch	SupMatch+Match	Total
Baballe	0.63	14.78	0.01	0.01	0.21	0.07	15.71
SmRub	0.63	14.88	0.19	0.2	2.86	0.67	19.43
Sport	0.63	14.9	0.08	0.1	1.9	0.18	17.79
Hpair	0.09	2.06	0.43	0.69	8.04	0.44	11.75
Inria	0.36	8.58	1.54	1.64	33.73	6.32	52.17
Office	0.18	4.45	0.62	0.76	12.29	9.73	28.03
Pair	0.36	8.5	3.52	5.25	81.05	2.35	101.03

Table 2: Execution times for various modules in Serial2

The execution times for our parallel bit vector implementation (Table 4) has much lower overall execution times than our previous parallel implementation despite doing more work (*i.e.*, the entire *correspondence finder* algorithm). In fact, every module runs more quickly in our second parallel implementation. ReadImg runs faster because we modified the way images are read into each processor. Corner runs much faster because we have eliminated the AppendTks module and many "malloc" calls. PtNbr and CandMatch benefit from the use of bit vectors to represent point sets.

In comparing the serial and parallel bit vector implementations, the execution time for each module is comparable except for Corner. For a single processor, the parallel version of Corner is roughly twice as slow as the serial version.

4.2 Speedup Graphs

Tables 12–16 shows the speedup graphs for the parallel index list version (parallel1), and Tables 17–23 shows the graphs for the parallel bit vector version (parallel2).

The overall speedup of our initial parallel implementation is fairly good, with the exception of the *Pair* image. An examination of *Pair* reveals that the distribution of points when we use three or more processors is very poor. The distribution of points on each processor for the left image is 166, 708, and 141 points using three processors and 75, 400, 430, and 100 points using four processors. The distribution is similar for *Pair*'s right image.

Figure 13 shows the times for ReadImg are erratic. There are several reasons for the variations. First, the image files are located on two different disks, one installed in the SP's control workstation and one internal to one of the SP nodes. These disks are mounted on all of the SP nodes via NFS. The time to read the images is such a small number (varying between a half and three seconds) that it is quite likely that network traffic and contention for the disk between processors and other users causes the unpredictable behavior.

Figure 14 shows encouraging speedups in Corner for all images which is a result of locality and regular communication patterns. However, performance does degrade as we add processors. The speedup values in PtNbr are erratic (Figure 15), but are very good for several images (*HPair*, *Inria*, and *office*). The distribution of the points affects the effectiveness PtNbr. Poor speedups occur when the points are not evenly distributed among the processors. Lastly, CandMatch has relatively low speedup and performance degrades on several images. Again, we attribute these problems to poor point distribution.

The overall speedup of our parallel bit-vector version looks very similar to the speedup in our initial parallel implementation but the problem with Pair's distribution of points remains. We do achieve a slightly higher degree of speedup on each image in comparison to the parallel index list version.

The speedup graphs (Figures 18– 23) for the modules in Parallel2 are not regular. ReadImg suffers from the same problems as the original parallel version. Corners exhibits excellent speedup because it processes image data. PtNbrs and CandMatch are very sensitive to processor load and network traffic

Image	Num procs	ReadImg	Corner	PtNbr	CandM	Total
Baballe	1	1.98	29.42	0.06	0.06	31.36
	2	1.62	15.00	0.04	0.05	16.74
	3	1.94	13.41	0.03	0.04	15.43
	4	1.12	11.64	0.03	0.04	12.85
SmRub	1	2.06	29.56	5.11	5.28	41.97
	2	1.78	15.33	3.69	3.22	24.83
	3	1.85	11.84	2.43	2.02	18.72
	4	1.25	10.77	2.20	2.04	16.96
Sport	1	2.23	29.62	1.77	1.79	35.27
	2	1.56	15.93	1.19	1.32	20.26
	3	1.74	13.29	0.85	0.90	17.05
	4	1.39	11.57	0.82	0.97	15.23
Hpair	1	0.29	4.09	21.94	34.68	60.96
	2	0.34	2.16	10.24	22.91	38.33
	3	0.44	1.41	7.09	15.82	27.80
	4	0.41	1.24	5.69	14.18	25.25
Inria	1	1.16	17.83	133.18	115.08	267.05
	2	1.04	8.99	77.02	119.99	233.86
	3	0.84	7.84	45.81	102.35	168.71
	4	0.99	6.95	36.97	77.85	135.77
Office	1	0.80	8.89	31.61	36.26	76.16
	2	0.57	4.60	20.09	28.57	60.18
	3	0.60	4.32	13.26	20.17	41.49
	4	0.57	3.69	11.05	16.30	35.83
Pair	1	1.28	17.07	524.98	713.34	1234.98
	2	1.09	8.83	260.27	388.16	664.74
	3	1.09	5.98	309.13	345.92	802.88
	4	1.31	4.39	224.22	285.31	569.31

Table 3: Execution times for various modules in Parallel1

because of their extremely short running times. PtMatch does not achieve a very high speedup because of poor point distribution. Our test images do not exhibit many matches, so by the time matches are distributed among the processors the supporting match has relatively few matches to work with. A point load balancing scheme might correct the behavior of images like *Pair*.

Image	**Num procs**	**ReadImg**	**Corner**	**PtNbr**	**CandM**	**PtMatch**	SupM	**Total**
Baballe	1	0.655	29.383	0.011	0.01	0.211	0.044	30.317
	2	0.649	15.996	0.008	0.008	0.18	0.036	16.88
	3	0.651	11.93	0.004	0.004	0.14	0.018	12.785
	4	0.647	9.598	0.004	0.004	0.137	0.012	10.445
SmRub	1	0.654	29.636	0.181	0.196	2.846	0.425	33.941
	2	0.648	16.343	0.138	0.143	2.436	0.404	20.116
	3	0.654	12.329	0.091	0.095	1.489	0.327	14.596
	4	4.633	9.622	0.053	0.055	1.614	0.197	12.681
Sport	1	2.027	29.653	0.091	0.101	1.925	0.118	33.825
	2	0.659	16.099	0.072	0.078	1.596	0.097	18.753
	3	0.655	11.965	0.027	0.031	1.275	0.093	14.162
	4	3.565	10.703	0.059	0.072	1.4	0.095	12.316
Hpair	1	0.099	4.103	0.417	0.686	8.277	0.262	13.849
	2	0.101	2.236	0.144	0.222	5.781	0.141	9.027
	3	0.172	1.413	0.201	0.35	3.88	0.109	6.083
	4	0.438	1.223	0.179	0.32	3.467	0.096	5.26
Inria	1	1.137	17.859	0.577	1.595	33.765	3.846	58.789
	2	0.44	9.2	1.411	1.433	30.19	3.104	45.798
	3	0.412	6.285	1.034	0.992	19.041	2.304	30.131
	4	0.41	4.669	0.796	0.741	16.434	2.442	26.212
Office	1	0.214	8.861	0.592	0.724	12.235	6.027	28.657
	2	0.252	4.507	0.492	0.611	10.781	3.879	20.684
	3	0.208	3.033	0.329	0.405	6.831	2.489	13.459
	4	0.219	2.722	0.218	0.268	6.13	1.778	11.834
Pair	1	1.203	17.107	3.382	5.039	80.785	1.019	108.56
	2	1.167	8.741	1.537	2.269	42.924	0.467	58.054
	3	0.464	5.854	2.47	3.298	57.005	0.772	69.933
	4	0.565	4.434	1.651	1.691	37.261	0.543	47.103

Table 4: Execution times for various modules in Parallel2

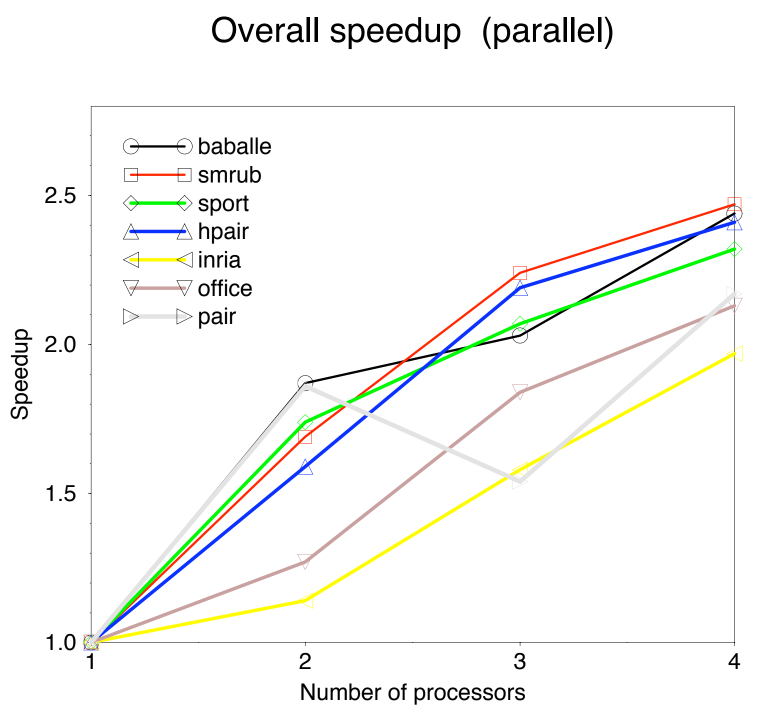


Figure 12: Overall speedup (Parallel)

Speedups for individual stages in *Parallel*

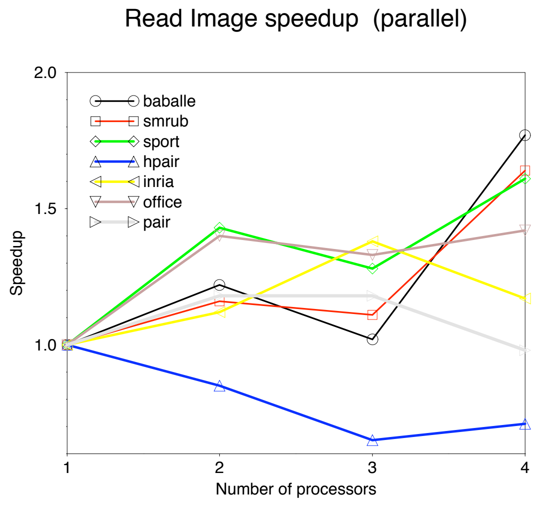


Figure 13: Speedup for Read Image

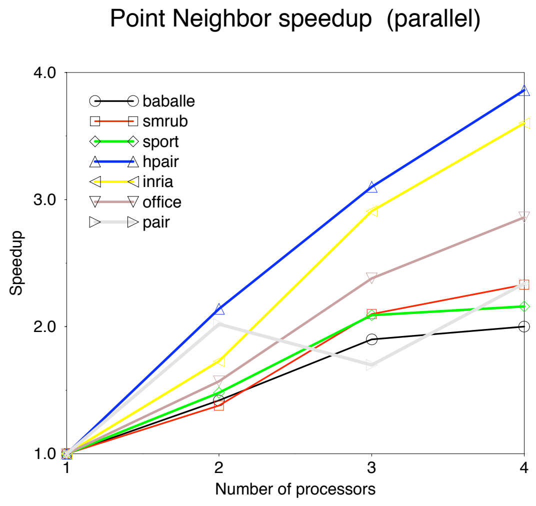


Figure 15: Speedup for point neighbors

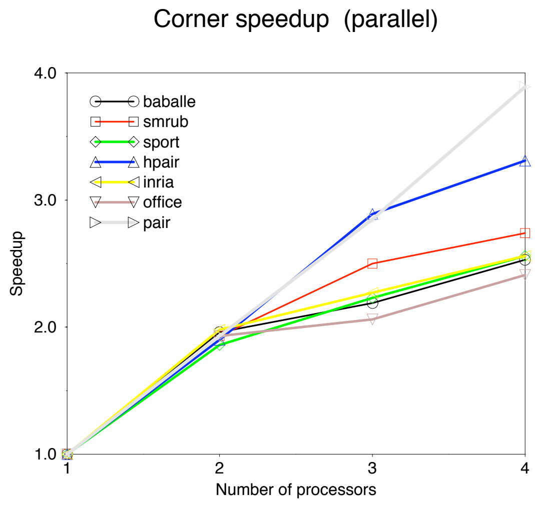


Figure 14: Speedup for corners

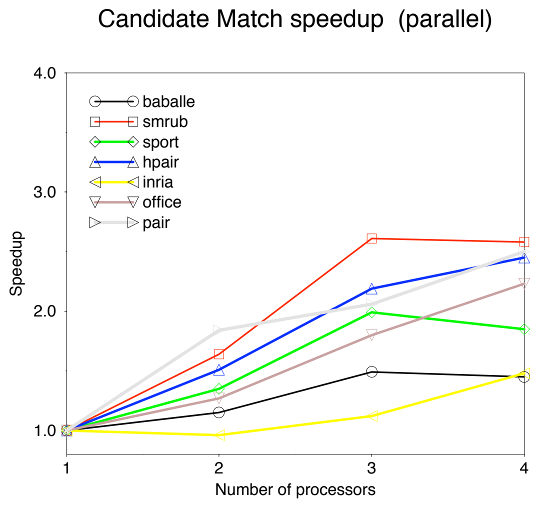


Figure 16: Speedup for candidate matches

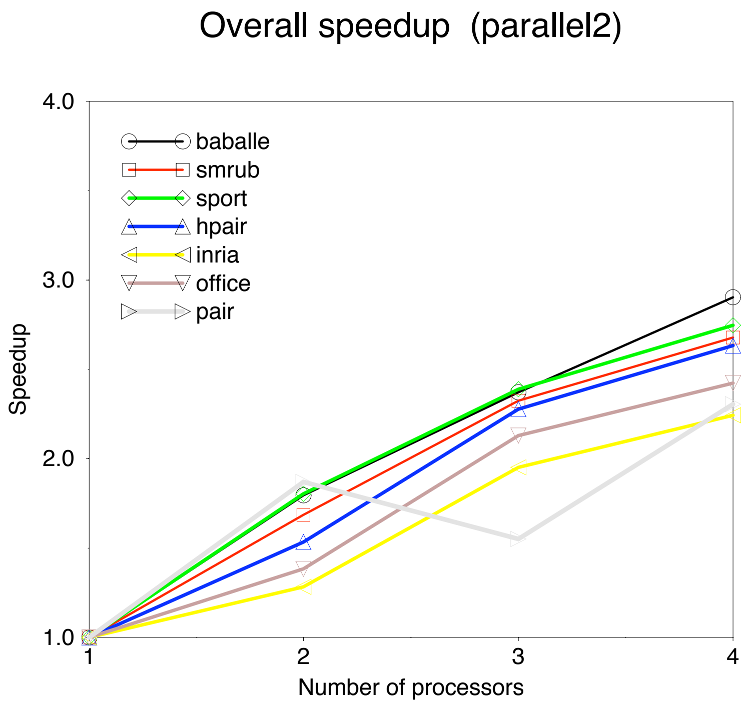


Figure 17: Overall speedup (Parallel2)

Speedups for individual stages in *Parallel2*

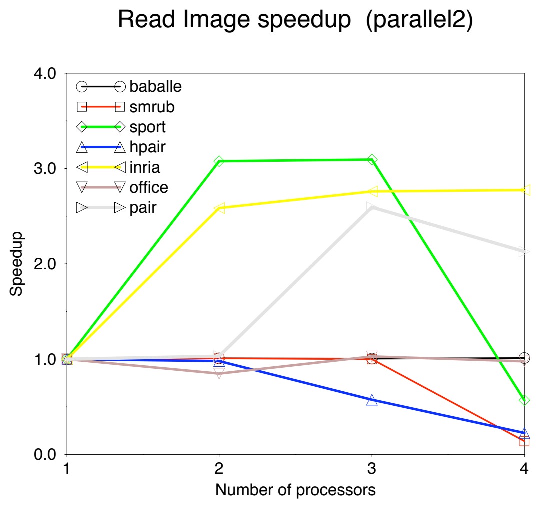


Figure 18: Speedup for Read Image

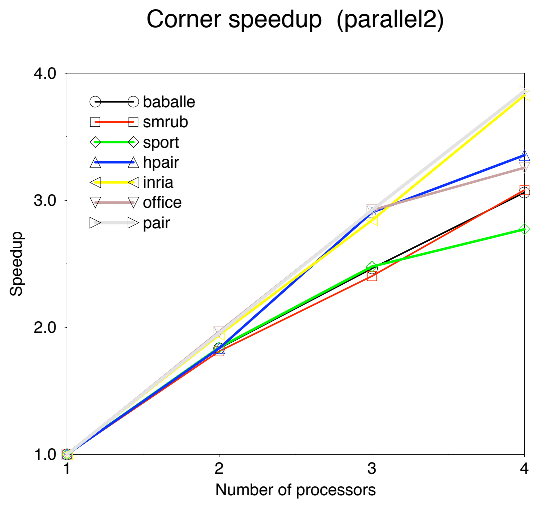


Figure 21: Speedup for corners

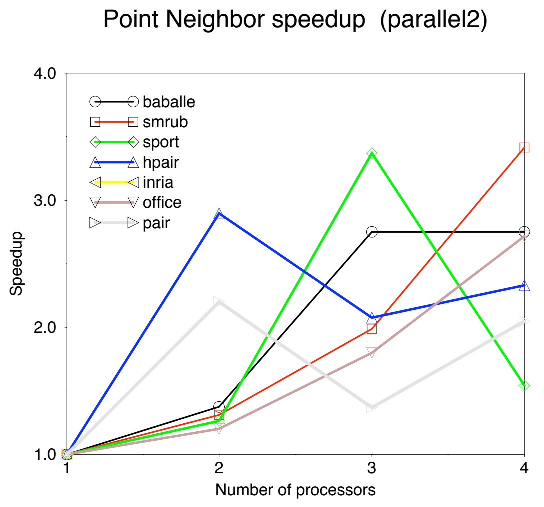


Figure 19: Speedup for point neighbors

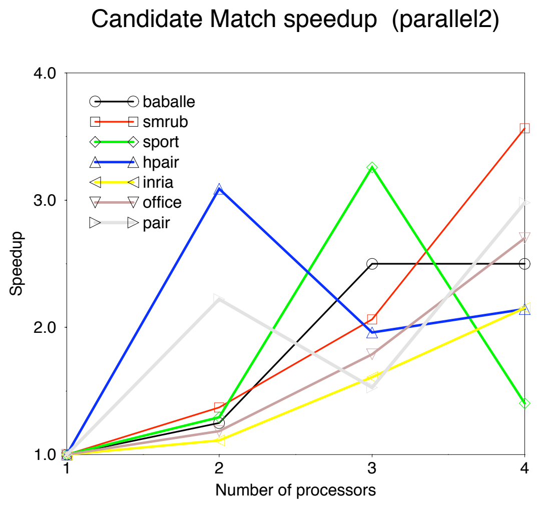


Figure 22: Speedup for candidate matches

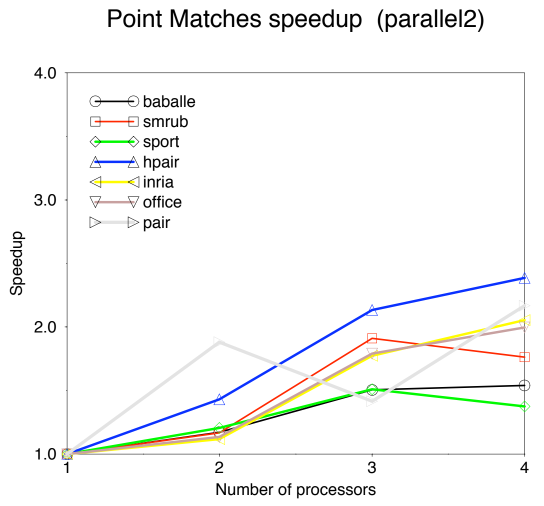


Figure 20: Speedup for Point matches

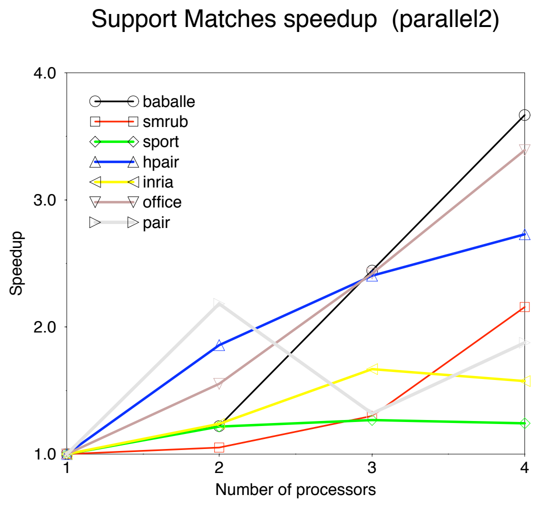


Figure 23: Speedup for Supporting matches

5 Evaluation of our Implementations

This section contains an evaluation of our parallelization effort and suggestions for additional work to be done.

5.1 Evaluation

Even though we obtain good speedups across a small number of processors, *correspondence finder* is not the best candidate for parallelization because

- *Correspondence finder* has little locality (or regularity) of reference for the point and match records,
- *Correspondence finder* has a high ratio of communication to computation, and
- the final relaxation step is inherently sequential.

We mitigate the severity of these problems by aggregating communication and maintaining local copies of remote data. However, we sacrifice generality (in the sense that our point sets may not work well for another application) and scalability.

The intermediate-level vision code in *correspondence finder* does not exhibit the locality of communication that the image processing portion exhibits. Interestingly, points appear to have locality because they have a position in image space. However to process a point, *correspondence finder* must access "neighboring" points and the search window for these points is quite large. In contrast, the image processing portion of match uses search windows that are three to five pixels wide. Figure 24 depicts the search windows for various stages of match. To build Q's list of neighbors in its own image, *correspondence finder* uses a search window with a radius of 1/4 of the image dimensions. To build Q's list of potentially matching points, *correspondence finder* uses a search window with a radius of 1/2 of the image dimensions. To find supporting matches for Q, *correspondence finder* uses a search window with a radius equal to the sum of the other two radii. It is clear from Figure 24 that accessing the points required to process Q requires substantial communication with most, if not all, other processors. Therefore, with the current window sizes, *correspondence finder* does not benefit from spatial locality. We do not include the time for the relaxation step in our timings in Section 4.

The relaxation step, which selects the best couple of matches in each iteration, is inherently sequential. The determination of which matches are best depends on the strength of the other matches in which the endpoints participate. Once a match is selected as best, all other matches which contain this point are invalidated, which in turn changes the best match for other points. These interdependencies effectively serialize the relaxation algorithm. This problem could be remedied by perhaps implementing a completely different, parallel relaxation algorithm, however, we did not explore that possibility.

5.2 Future work

This section discusses our experiences and ideas about how the parallelization of this algorithm can be improved.

- Our current parallel implementations leave points and matches on the processor on which they are created. This approach avoids the overhead of redistributing these data structures at the cost of causing load imbalance. In Figure 25 we observe that for the office image pair the distribution of points per processor is very skewed. We see the impact of this imbalance on the speedup numbers for the *pair* images.

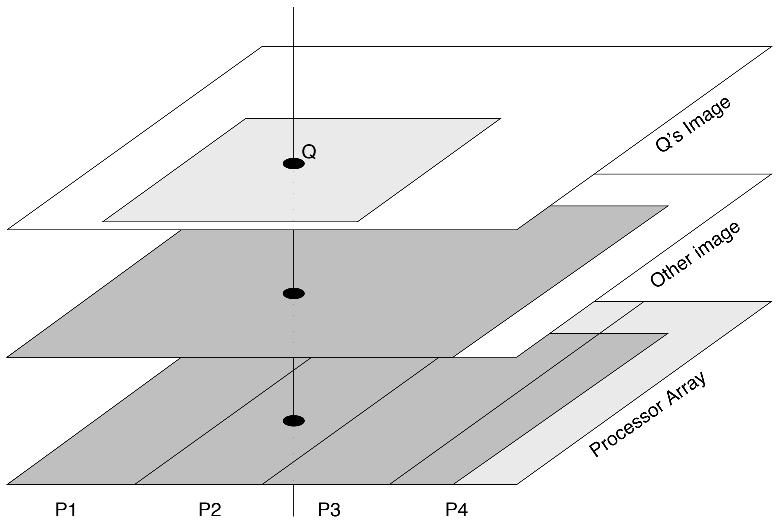


Figure 24: The search window for point Q's neighbors in its own image has a radius that is 1/4 of the image's dimensions, and the search window for Q's potential matches in the other image has a radius that is 1/2 of the image's dimension. Moreover, supporting matches can be found in an area whose radius is the sum of the other two radii.

A communication step should be inserted after the image phase and after PtMatch to redistribute points and matches, respectively. We designed our indexing scheme in both our parallel implementations to support redistribution, so this should be straight forward. Care must be taken, if the algorithm is to retain what little locality it has.

- The functional flow diagrams in Figures 3–5 indicate that functional parallelism is available. However, our implementation currently does not take advantage of any functional parallelism. We do not have enough processors to justify extracting more parallelism and adding functional parallelism might aggravate communication issues because the results of work done by separate processors would have to be combined.

  Functional parallelism may prove useful even if the number of processors is not increased. Much of our communication uses collective communication operations which causes an implicit barrier and puts a lot of data in the network at once. Having additional processes per processor may allow the processors to remain busy during communication operations. However, it is not clear how well, if at all, MPI supports this approach.

- Not all of the modules in the match phase take full advantage of the bit vectors in the bit vector implementation. Our bit vector manipulation code only works on bit vectors of equal size, but module PtMatches builds bit vectors of varying lengths because it does not know how many matches will be generated. Therefore, later modules (especially GetSupMatch) are not able to use bit vector operators. Correcting this problem will result in faster serial and parallel implementations.

- Our implementations are currently limited to reading images from KBV's proprietary format. This limitation has made collecting test data difficult and reduced the number of available images. Several additional experiments are possible:

  - Use larger images. Larger images should result in better speed-up by amortizing the cost of

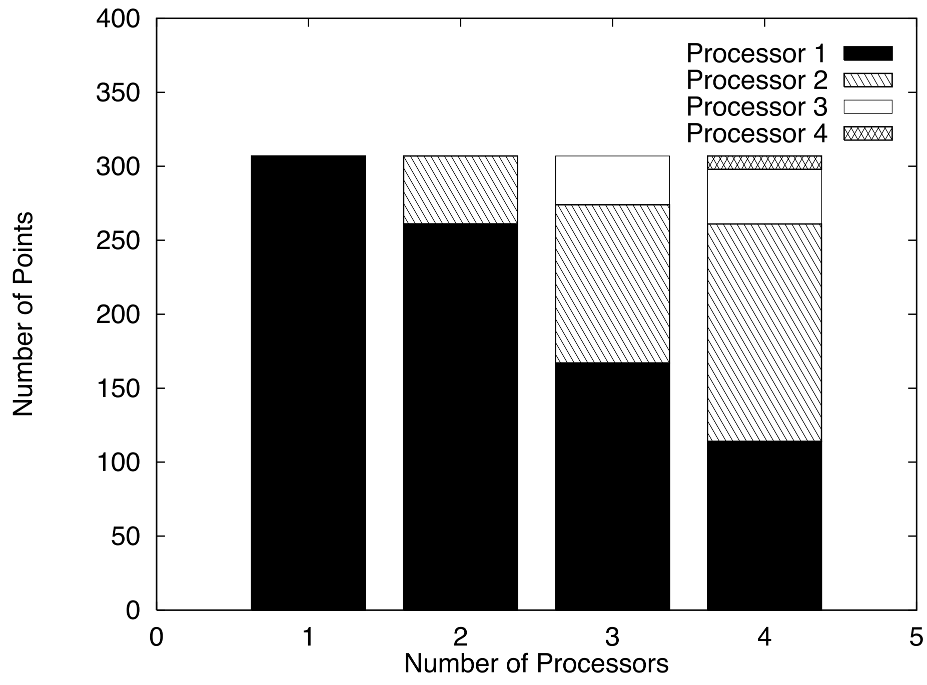


Figure 25: This graph shows the load imbalance caused by allowing points to remain on the processor on which they are created. The data for graph is taken from the office images (Figure 31).

accessing distributed data.

- Compare the speedup on images of the same size with a wider variation in the number of corners and matches.
- Use an image sequence of a camera moving towards an object with many corners against a featureless background. This test may demonstrate the effects of locality, though redistributing points and matches may obviate this test.
- Run our bit vector parallel implementation on a machine with more nodes. It would be interesting to determine where speedup tapers off and why[9].

6 Conclusions

For the portion we parallelize, we are able to obtain good speedups on a computer vision application that includes intermediate-level, as well as low-level, vision code. The intermediate-level vision code does not exhibit the high regularity and local communication of most image codes, however, with careful design, we are able to mitigate the communication costs. To find a good design, we leverage the lessons learned from developing our initial version to the design of the second version.

Our experiments show that the performance of *correspondence finder* is sensitive to the number of points and matches and that parallel versions of *correspondence finder* are in turn sensitive to the distribution of points and matches. Hence, any future parallel implementations should be careful to evenly distribute these data structures.

[9]Note that the current implementation may need some modifications in its handling of match bit vectors to handle processors with zero matches.

We also find that mapping an application with data dependent communication patterns onto the SPMD model is quite difficult. The SPMD model requires that the sender and receiver anticipate each other's actions or that they continually poll for messages. MPI does not provide architecture independent polling support so we do not use this approach in *correspondence finder*. We engineer around the problem in our second parallel implementation by communicating data fields for all of the points' (or matches') at once.

References

[Ame] Amerinex Applied Imaging, Inc., Amherst, MA. *The KBVision System Programmer's Reference Manual*.

[Fos95] Ian T. Foster. *Designing and Building Parallel Programs*. Addison-Wesley Publishing Company, Inc., 1995.

[Pre86] K. Preston, Jr. Benchmark results: The abingdon cross. In L. Uhr, K. Preston, Jr., S. Levialdi, and M.J.B. Duff, editors, *Evaluation of Multicomputers for Image Processing*, pages 23–54, Orlando, FL, 1986. Academic Press. (Proceedings of the 1984 Multicomputer Workshop, Tucson, AZ).

[Ros87] A.R. Rosenfeld. A report on the DARPA image understanding architectures workshop. In *Proceedings of the 1987 DARPA Image Understanding Workshop*, pages 298–302, Los Angeles, CA, February 1987. Morgan Kaufmann Publishers.

[UPLD86] L. Uhr, K. Preston, Jr., S. Levialdi, and M.J.B. Duff. Preface. In L. Uhr, K. Preston, Jr., S. Levialdi, and M.J.B. Duff, editors, *Evaluation of Multicomputers for Image Processing*, pages ix–xiv, Orlando, FL, 1986. Academic Press. (Proceedings of the 1984 Multicomputer Workshop, Tucson, AZ).

[Wer96] Dawn Werner. Personal communication, August 1996.

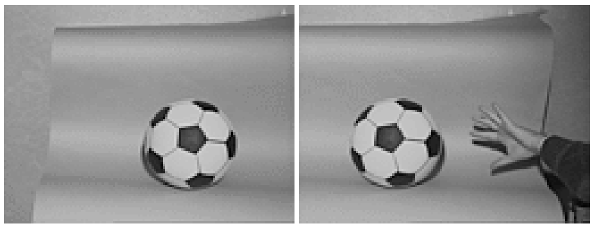
[Wre96] David Wren. Correspondence-finder. Technical report, Amerinex Applied Imaging, Inc., 1996.

[WRHR91] C. Weems, E. Riseman, A. Hanson, and A. Rosenfeld. The DARPA image understanding benchmark for parallel processors. *Journal of Parallel and Distributed Computing*, 11:1–24, 1991.

[ZDFL94] Zhengyou Zhang, Rachid Deriche, Olivier Faugeras, and Quang-Taun Luong. A robust technique for matching two uncalibrated images through the recovery of the unknown epipolar geometry. Technical report, INRIA, 1994.

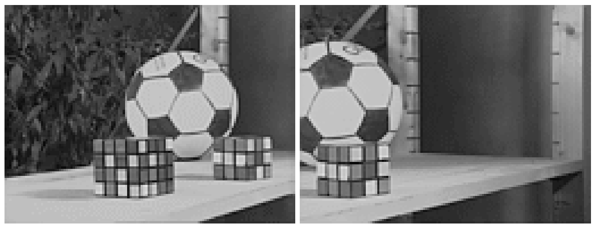
A Images

This section presents the image data we used as test cases for our experiments in Section 4.



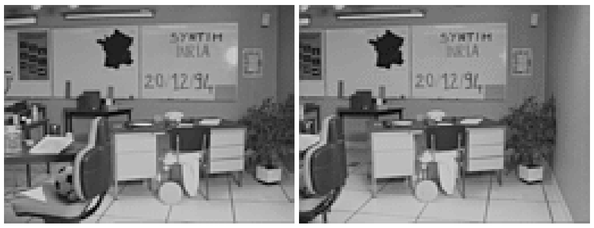
Left Image– Baballe0 Right Image– Baballe1

Figure 26: Baballe stereo image pair



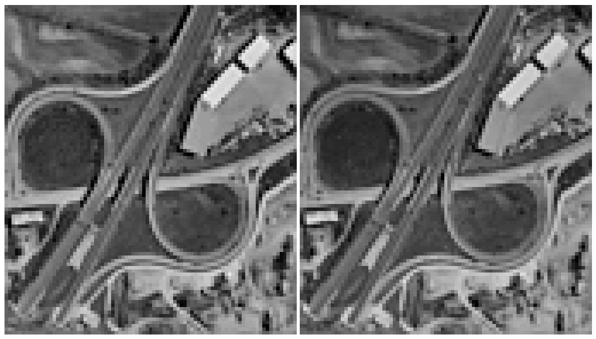
Left Image– SmRub0 Right Image– SmRub1

Figure 27: SmRub stereo image pair



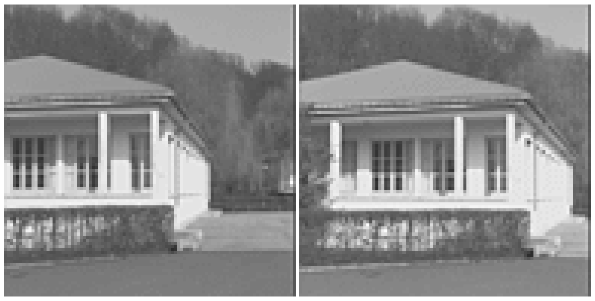
Left Image– Sport0 Right Image– Sport1

Figure 28: Sport stereo image pair



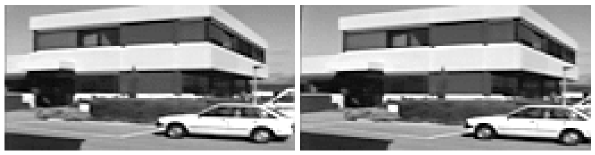
Left Image– hpairl Right Image– hpairr

Figure 29: hpair stereo image pair



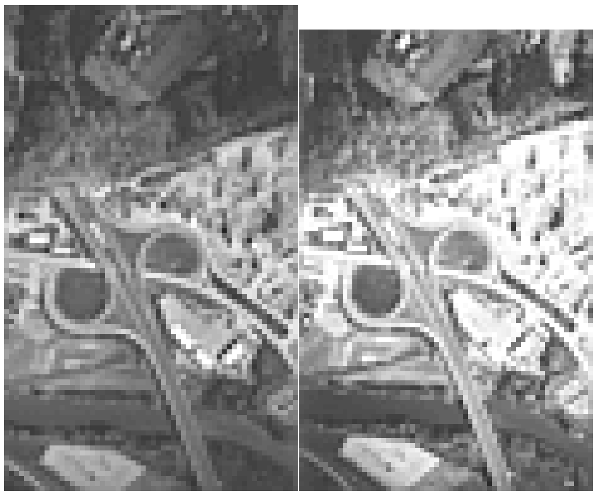
Left Image– inrial Right Image– inriar

Figure 30: inria stereo image pair



Left Image– officel Right Image– officer

Figure 31: office stereo image pair



Left Image– pairl Right Image– pairr

Figure 32: pair stereo image pair