# Parallel Algorithms for Motion Panorama Construction

Yong Wei, Hongyu Wang, Suchendra M. Bhandarkar and Kang Li
*Department of Computer Science, The University of Georgia*
*Athens, Georgia 30602-7404, USA*
*E-mail: {yong, suchi, kangli}@cs.uga.edu*

## Abstract

*A motion panorama is an efficient and compact representation of the underlying video. However, the motion panorama construction process is computationally intensive and hence extremely time consuming. Addressing this issue is crucial when one considers using motion panoramas in a real-time environment such as live video transmission. We present two parallel algorithms for motion panorama construction, namely, the shared memory parallel algorithm (SMPA) that uses POSIX threads and the distributed memory parallel algorithm (DMPA) that uses MPI. The parallel algorithms are tested on real videos. Experimental results show that the SMPA achieves linear speedup in most cases whereas the DMPA suffers from reduced efficiency when the number of processors exceeds 8.*

## 1. Introduction

Motion panoramas are an efficient representation of the underlying video and are based on *image mosaicking* [1] where a set of small images are combined into a larger composite image. The basic idea is to extract several key frames from a video and *stitch* them together into a single wide-angle panoramic image. The *stiching* is done by aligning all frames of that sequence to a fixed coordinate system and then integrating the aligned images into a mosaic image.

In a static panorama, the input video sequence is segmented into contiguous scene subsequences, and the mosaic image is constructed for each scene subsequence to provide a snapshot view of the scene subsequence. The static panorama exploits long-term temporal redundancies over the entire scene subsequence and spatial correlations over large portions of the image frames, and therefore constitutes an efficient scene representation. It is ideal for video storage and retrieval, especially for rapid browsing in large digital libraries and for enabling efficient random access to individual frames of interest [3]. However, the changes in the scene with respect to the background (termed as *residuals*) caused by a moving object are not captured by a static mosaic image and need additional representation. On the other hand, a motion panorama is a sequence of evolving mosaic images [1] where the contents of each new mosaic image are updated using the current information derived from the most recent frame. In general, since the residuals are relatively small, the motion panorama is an ideal tool for low bit-rate transmission [2]. However, due to the incremental nature of frame reconstruction, motion panoramas, lack the important capability of random access to individual frames of interest.

Several applications of video-based static and motion (dynamic) panoramas have been described in the literature. Irani, Hsu, and Anandan [2] describe a technique for video compression using dynamic mosaics. Teodosio and Bender [3] introduce the notion of Salient Stills (static panoramas) for the purpose of video visualization. Irani et al. [4] demonstrate the use of static and dynamic panoramas in scene change detection, video search and indexing, and video editing and manipulation. Shum and Szeliski [5] show how motion panoramas can be used to create virtual reality environments to enable virtual travel. Irani and Peleg [12] show how motion panoramas can be used to improve the resolution of the underlying video data resulting in a super-resolution mosaic. Static and dynamic video mosaics have also been used as the basis of generative video models [15]. More recently, Pan [14] has shown how motion panoramas can be used for video transcoding in low-power mobile multimedia devices. Some applications of video-based static and motion panoramas have also been commercialized [6]–[8].

In spite of several algorithmic advances, construction of video-based static and motion panoramas is a computationally intensive process. For example, a serial implementation of the recent motion panorama reconstruction algorithm proposed by Bartoli et al. [1] was observed to take more than 9 minutes to

process a 12-frame video sequence on a Sun UltraSPARC workstation with a 1.05 GHz CPU and 1.0 GB of RAM. Note that at a normal sampling rate, a 1-second video sequence usually consists of 30 frames. Therefore, most algorithms for generation of video-based panoramas are suitable only for off-line applications. For applications with real-time constraints, the need for faster processing capability is critical. Even in the case of applications where the panorama generation could be done off-line, faster processing capability is clearly desirable.

In light of the above, we present two parallel algorithms for motion panorama construction, namely, a shared memory parallel algorithm (SMPA) that uses POSIX threads (Pthreads) [18] and a distributed memory parallel algorithm (DMPA) that uses the Message Passing Interface (MPI) [17]. The SMPA is conceptually and practically simpler since it does not require the user to explicitly specify the communication of data between tasks. However, the fact that the data locality and data placement are transparent to the programmer, could have a negative impact on performance and scalability of the SMPA. The DMPA offers performance and scalability but compromises the ease of use since the programmer is typically responsible for sending and receiving data amongst the processors. In addition, the DMPA might not scale well on problems that require a lot of data communication amongst the tasks since the complexity of the inter-processor communication overhead typically scales super-linearly with respect to the number of processors in the networked cluster.

## 2. Sequential Algorithm for Motion Panorama Construction

The sequential motion panorama construction algorithm [1] consists of three major phases: static background generation, background-foreground segmentation (i.e., extraction of moving objects) and final panorama composition. During the first phase, the homographies corresponding to the motion of the camera are computed for certain frames. The static background for the entire scene underlying the video sequence is generated by stitching the individual frames into a single large wide-angle panoramic image using these homographies. In the second phase, the dynamic foreground, which includes regions corresponding to both, moving objects and false detections in the scene, is segmented by warping together three consecutive frames in the video sequence and consequently detecting the intensity discrepancy at each pixel location. The noise in the dynamic foreground is smoothed using a Gaussian filter. The moving objects are detected using a connected components labeling (CCL) algorithm. Regions corresponding to false motion are deleted using a size filter applied to the output of the CCL algorithm. Finally, the foreground objects are pasted back onto the static background using the location information from the homographies computed in the first phase and the position coordinates computed during the second phase. For further details on the sequential motion panorama construction algorithm, the interested reader is referred to [1].

## 3. Parallel Algorithms for Motion Panorama Construction

Each of the aforementioned steps in the sequential algorithm for motion panorama construction is parallelized as described in the following subsections:

### 3.1. Homography Estimation

The Homography Estimation procedure in the case of the sequential motion panorama construction algorithm consists of four computational tasks that are performed in order: (a) Interest Point Detection where interesting feature points (typically corner points) are detected in each video frame; (b) Correspondence Matching where for feature points detected in a given frame, the corresponding matching feature points in the reference frame are identified; (c) RANSAC-based Estimation where false matches or outliers are removed; and (d) Optimal Transformation Estimation where the homography transformation between the current frame and the reference frame is computed.

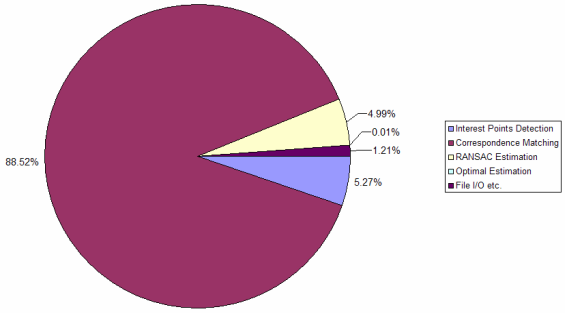



**Figure 3.1 CPU time distribution for various tasks in sequential homography estimation**

Figure 3.1 shows the CPU time distribution of the aforementioned tasks in the Homography Estimation process. As can be seen from Figure 3.1, the tasks of Interest Point Detection, Correspondence Matching and RANSAC Estimation require the most computation and

collectively account for 98% of the total execution time of the sequential Homography Estimation algorithm. Hence, an efficient parallel algorithm for each of these three steps is crucial to the final outcome. On the other hand, Optimal Transformation Estimation requires little computation and has limited parallelization potential. The remaining time is spent mostly on file I/O which, in the current implementation, is not parallelized.

**3.1.1. Interest point detection.** Since the interest value of a pixel only depends on the color values of neighboring pixels in the original image frame, no data dependence exists between any two computations. Thus parallelization can be achieved by partitioning the original image frame, consisting of 480 rows and 720 columns in our case, into as many blocks as the number of processors, such that each block has an almost equal number of pixels, and assigning one block to a processor for computation. Once the above process is completed, the resulting matrix of interest points is also divided into as many blocks as the number of processors. Each processor searches for the pixel that has the largest interest value within each neighboring and non-overlapping $30 \times 30$ window within its assigned block. These pixels are then marked as interest points.

In the SMPA, no synchronization is required during the entire interest point detection process because the block partition remains the same. The overhead for the SMPA, if any, is due to memory contention. The DMPA does not suffer from memory contention, however, it requires explicit communication amongst the nodes at the end of the computation so that all nodes can obtain a local copy of the matrix of interest points. The simplest approach is to communicate the entire matrix. However, because there is only one interest point within a $30 \times 30$ region, communicating the entire matrix is extremely inefficient. Thus, in order to reduce the redundancy during the communication, an array of size (480×720)/(30×30)×2 is used in the communication instead of the entire matrix. Each node copies the coordinates of all interest points within its assigned block into the array before performing an all-to-all communication over the entire array. At the end of the communication, the array in each node, which contains the coordinates of all the interest points, is used to mark the local interest point matrix. As a result, the size of data used in the communication is reduced by a factor of 2/(30×30) = 1/450.

**3.1.2. Correspondence matching.** After having determined the interest points in a given frame, the next step is to search the following frame in the video sequence for the corresponding matching interest points [10]. Again, there is no data dependence between any two distinct searches for the corresponding matching interest points. Hence, the key issue in the parallelization is how to assign the interest points evenly to each processor so that the workload is balanced amongst all the available processors. Our implementation is similar to a multiple server queue, where several servers simultaneously serve a group of clients. A client waits in the queue until a server has finished its assigned task. Thus all servers are kept busy until each server finishes its assigned task and there is no client in the queue.

In the SMPA, we use a global cursor to identify the first interest point in the queue. When a processor is available, it first locks the cursor from being accessed by another processor. After it gets the information about the first interest point in the queue, it updates the cursor's position, releases the lock attached to the cursor and begins to search for the corresponding point in the next frame. If there is no interest point left in the queue, the processor releases the lock attached to the cursor and waits for other processors to finish their assigned tasks. The overhead incurred during this step is the time spent in synchronizing the value of the global cursor. An alternative approach is to statically assign the interest points to the processors based on a certain criterion. For example, we can partition equally the set of interest points and assign each subset to a processor. Such an approach can eliminate the synchronization overhead, but may also suffer from an unbalanced load. This is so because the computation required for each single search may vary and with static assignment, the worst case could occur where all the interest points that require less computation are assigned to one processor and all interest points that require extra computation are assigned to the others.

We have tested the performance of the SMPA for both, dynamic and static assignments of interest points to processors. In the case of static assignment, we employ the same scheme used in the DMPA, discussed in the following paragraph. Experimental results show that there is almost no difference between the performance of the SMPA with dynamic and static assignments when using 2 or 4 processors. When using 8 processors, our dynamic assignment scheme performs on average 2%-3% better than the static version in terms of total execution time. This is expected since the probability of a load imbalance increases as the number of processors increases.

In the case of the DMPA, we did not employ the above dynamic assignment scheme. The reason is that the use of a global cursor would entail a costly

overhead of explicit inter-processor communication over the cursor. Any performance gain would most probably be offset by the inter-processor communication overhead. Hence, in the DMPA, the interest points are assigned based on the following scheme. In our implementation, we first number the interest points from 0 to P. The interest points with the numbers 0, $N_n$, $2N_n$, $3N_n$…, where $N_n$ is the total number of nodes (processors), are assigned to the node with id 0, interest points with the numbers 1, $N_n+1$, $2N_n+1$, $3N_n+1$… are assigned to the node with id 1, and so on. Although this static assignment may suffer from a load imbalance problem, the experiment results discussed in Section 4, however show that the resulting performance is still quite satisfactory.

**3.1.3. RANSAC estimation.** The RANSAC algorithm [16] basically repeats a statistical sampling process for a certain number of iterations in order to remove outliers from the input data. Each sampling process is independent. Also, the computation entailed by each sampling process is roughly the same. Therefore, the RANSAC procedure can be easily parallelized by sharing the total number of sampling iterations amongst all the available processors. Regardless of whether the SMPA or DMPA is used, the total number of sampling iterations is divided by the total number of available processors. The result is the number of sampling iterations a processor should repeat. If there is a residual $R$ (where $R > 0$), processors with an *id* from 0 to $R$-1 will perform an additional iteration.

Each processor computes the homography from its local consensus set with the largest number of inliers. The local largest number of inliers amongst all the processors is finally compared and the homography computed from the largest number of inliers among all the processors is selected. In the SMPA, this is done by using *mutex_lock()* & *mutex_unlock()* operations so that the homography and the largest number of inliers are updated by each processor one by one. In the DMPA, all child nodes send their local copies of homographies along with the largest number of inliers to the master node. The master node computes the result based on the data it receives.

***Algorithm for Homography Estimation:***

1: Partition the frame into a several blocks, assign each block to a processor and let it calculate the interest values of all pixels within that block.

2: Within each 30 × 30 window, find the pixel with the maximum interest value and mark it as an interest point.

3: In the case of the SMPA, do the following:

Let the global cursor $C = 0$
Let the total no. of interest points = $P$
**for all** interest points from $i = 1$ to $P$ **do**
mutex_lock()
**if** $i > C$ **then**
$C = i$
mutex_unlock()
search for the corresponding point of interest point $i$
**else**
mutex_unlock()
**continue**
**end if**
**end for**

In the case of the DMPA, do the following:
Let the total no. of interest points = $P$
**for all** *interest points* from $i = 1$ to $P$ **do**
**if** ($i$ **mod** *total_number_of_processors*) = processor's ID **then**
search for the corresponding point of interest point $i$
**else**
**continue**
**end if**
**end for**

4: Let $S_n =$ *total _ sampling _ iterations* / *total _ number _ of _ processors*

5: Randomly select 4 pairs of corresponding points and estimate the homography based on these points, use the homography to compute the number of inliers.

6: Each processor repeats step 5 $S_n$ times.

7: The master processor uses the homography with the largest number of inliers among all processors to estimate the final optimal transformation.

## 3.2. Background Mosaic Generation

Background mosaic generation consists of two tasks: (a) estimating the homographies for a series of frames in the video sequence, and (b) mapping these frames onto the final static background mosaic using the estimated homographies. The background mosaic generation procedure is parallelized by first running the parallel algorithm (SMPA or DMPA) for homography estimation on a single frame and then using a single processor to execute the mapping process. This procedure is repeated on a series of frames in the video sequence.

## 3.3. Dynamic Foreground Segmentation

The sequential algorithm for dynamic foreground segmentation consists of three major processes performed in order: (a) generation of the probability image, (b) Gaussian filtering of the probability image, and (c) connected component labeling (CCL) and

segmentation of the foreground objects from the background. Figure 3.2 depicts the CPU time distribution for the three major procedures involved in the sequential algorithm for dynamic foreground segmentation. The processes of generating the probability image and then performing Gaussian filtering on the probability image together take up over 99% of the total execution time. Our works focuses mainly on the parallelization of these two tasks.

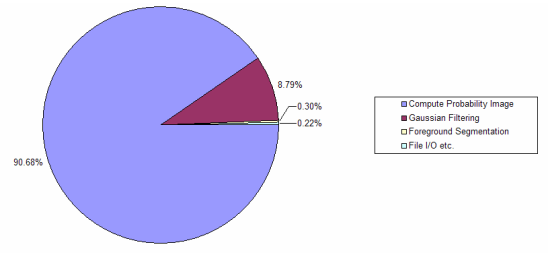



**Figure 3.2 CPU time distribution for various tasks in dynamic foreground segmentation**

**3.3.1. Computation of the probability image.** The two major steps in this task are: estimating the homography between two consecutive frames and computing the Mahalanobis distance matrix for all pixels [1]. We use the same parallel algorithm for homography estimation as the one used in static background mosaic generation. To parallelize the computation of the Mahalanobis distance matrix, we partition the matrix into several blocks and share the computation amongst all the available processors. This matrix computation depends only on the estimated homography and the color values of all pixels in a single frame. So once the homography is obtained, no further data synchronization or communication is required.

**3.3.2. Gaussian filtering of the probability image**. A two-dimensional Gaussian filter is applied on the probability image. Since the two-dimensional Gaussian filter is separable, the filtering operation is tantamount to applying the one-dimensional Gaussian filter along the *x*-axis followed by the application of the same filter along the *y*-axis. When applying the filter along the *x*-axis, the value of each pixel is updated based on the values of its neighbors on the same row. On the other hand, when applying the filter along the *y*-axis, the value of each pixel is updated based on the values of its neighbors in the same column.

In the SMPA, the changes in the data dependence pattern require the data to be synchronized before a different filter is applied. This can be easily done by placing a *barrier()* before applying the Gaussian filter along the second dimension. In the DMPA, the need for explicit inter-processor communication amongst all the processors has a significant negative impact on performance. Our original implementation used the *MPI_Allgather()* operation to communicate the entire probability image matrix. This approach is proven to be very inefficient. In Section 4.3, we provide an optimized version of the inter-processor communication process. After the probability image has been processed by the Gaussian filter masks along each dimension, it is partitioned into a several blocks such that each processor works on only one block for the remainder of the computation.

***Algorithm for Foreground Segmentation:***

1. Use the parallel algorithm to estimate the homography $H_p$ between the current and the previous frame.
2. Let each processor compute a portion of Mahalanobis Distance matrix $MD_p$ based on $H_p$.
3. Use the parallel algorithm to estimate the homography $H_n$ between the current and the next frame.
4. Let each processor compute a portion of Mahalanobis Distance matrix $MD_n$ based on $H_n$.
5. The matrix *MD* is computed as the sum of $MD_p$ and $MD_n$.
6. Partition the matrix *MD* and let each processor apply the *x*-mask of the Gaussian filter on a portion of the matrix.
7. Perform barrier synchronization.
8. Partition the matrix *MD* and let each processor apply the *y*-mask of the Gaussian filter on a portion of the matrix.
9. Partition the matrix *MD* and let each processor find the maximum value within its assigned partition of the matrix.
10. Let the master processor gather the results and compute the threshold as 1/5 of the maximum value of the whole matrix.
11. Partition the matrix *MD* and let each processor mark the pixels within its assigned partition whose values are greater than the threshold as foreground.
12. Let the master processor run the procedures for dynamic component labeling and segmentation.

## 4. Experimental Results

The experimental results for the SMPA were obtained using a Sun Fire 880 server with 8 UltraSPARC III 750MHz processors, where each processor has 8MB internal cache and shares a total of 16GB of global memory. Experimental results for the DMPA were obtained on a cluster of 16 Sun Blade 1500 workstations, each equipped with an UltraSPARC IIIi 1.05Ghz processor, 1MB Cache and 1GB local memory.

We measured the speedup for each parallelizable task during static background generation and dynamic foreground segmentation. As mentioned before, once the static background generation and dynamic foreground segmentation are performed, the final panorama composition does not involve intensive computation and is, in fact, intended to be performed on the client side on a device with limited CPU resources. Therefore, the final panorama composition is not discussed in this section. However, the overall performance, which includes both static background generation and dynamic foreground segmentation, is included for comparison. In the SMPA, the total number of processors used in the test ranged from 2 to 8, covering each power of 2. In the DMPA, we used the maximum available 16 processors. All tests were performed ten times and the averages of the results were computed. The input video sequence consists of 14 image frames, each frame of size 480 rows × 720 columns. Note that the problem size does not depend on the number of input frames, it only relates to the dimensions of a single frame.

## 4.1. Performance of the SMPA

Figure 4.1 shows the SMPA speedup results. It is quite obvious that SMPA performs very well. The overall CPU efficiency is shown in Table 4.1. Such high efficiency is primarily due to the fact that the motion panorama construction algorithm does not require too many data communication operations. The shared memory architecture with a high bandwidth memory bus helps to reduce the time for data synchronization. In both procedures, Correspondence Matching and RANSAC Estimation, the dynamic work load distribution technique proved to be extremely effective in terms of load balancing. Both tasks achieve nearly 100% CPU efficiency when using 4 or fewer processors. When using 8 processors, the CPU efficiency was reduced to approximately 89% due to the fact that the data synchronization overhead typically scales super-linearly with respect to the number of processors.

Figure 4.2 shows the proportion of the total execution time taken by each task. It was observed that the time spent in both file I/O and the inter-processor communication overhead did not increase significantly with the number of processors, thus proving that the parallel algorithm does very well in balancing the load. However, the proportion of total execution time taken by non-parallelizable tasks such as file I/O and dynamic component segmentation increases with an increasing number of processors. With 8 processors, these two tasks take up around 5% of the total execution time. From the data tabulated in Figure 4.2, we predict that the SMPA will work very well on at least 16 processors.

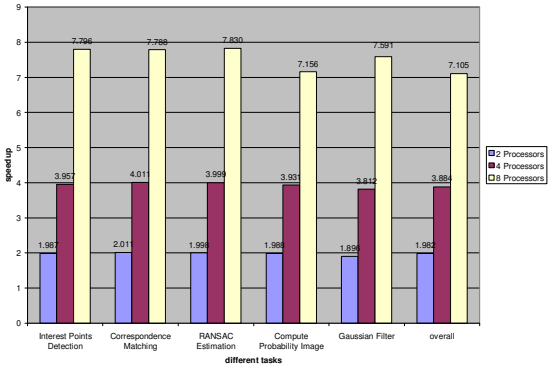
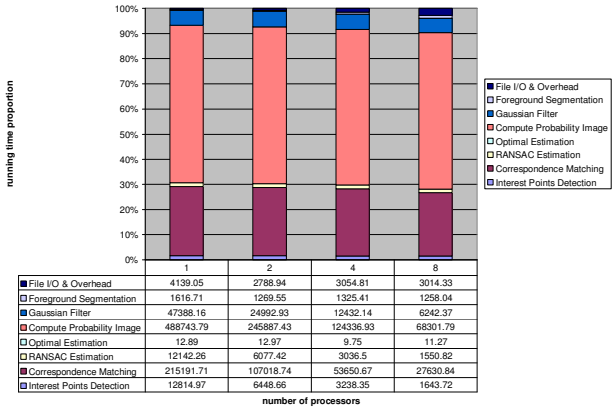



**Figure 4.1 Speedup results for the SMPA**




| | 1 | 2 | 4 | 8 |
|---|---|---|---|---|
| File I/O & Overhead | 4139.05 | 2788.94 | 3054.81 | 3014.33 |
| Foreground Segmentation | 1616.71 | 1269.55 | 1325.41 | 1258.04 |
| Gaussian Filter | 47388.16 | 24992.93 | 12432.14 | 6242.37 |
| Compute Probability Image | 488743.79 | 245887.43 | 124336.93 | 68301.79 |
| Optimal Estimation | 12.89 | 12.97 | 9.75 | 11.27 |
| RANSAC Estimation | 12142.26 | 6077.42 | 3036.5 | 1550.82 |
| Correspondence Matching | 215191.71 | 107018.74 | 53650.67 | 27630.84 |
| Interest Points Detection | 12814.97 | 6448.66 | 3238.35 | 1643.72 |

**Figure 4.2 Proportion of the total execution time taken by various tasks in the SMPA**

**Table 4.1 CPU efficiency results for the SMPA**

| # processors | 2 | 4 | 8 |
|---|---|---|---|
| CPU efficiency | 99.1% | 97.1% | 88.8% |

## 4.2. Performance of the DMPA

The DMPA for video-based motion panorama construction does not perform as well as the SMPA. This is mostly attributed to the overhead arising from explicit data communication amongst processors. The parallel algorithm does work well on the first three tasks, where little inter-processor communication is entailed. The overall CPU efficiency is shown in Table 4.2. It is quite obvious that the $O(n^2)$ communication overhead has a significant negative impact on the performance of the DMPA. The extreme case occurs during the process of Gaussian filtering, where the

speedup for 8 processors exceeds the speedup for 16 processors. With 16 processors, the DMPA achieves a CPU efficiency of only 62%. From Figure 4.4, we can see that file I/O and Gaussian filtering have a significant negative impact on the performance of the DMPA. Optimization of the file I/O is beyond the scope of this paper. In the next section, we describe the optimization of Gaussian filtering in case of the DMPA.

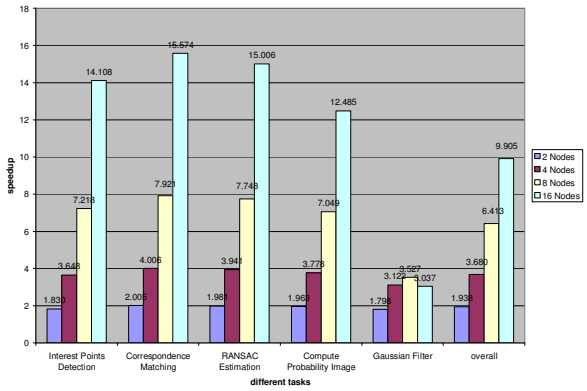
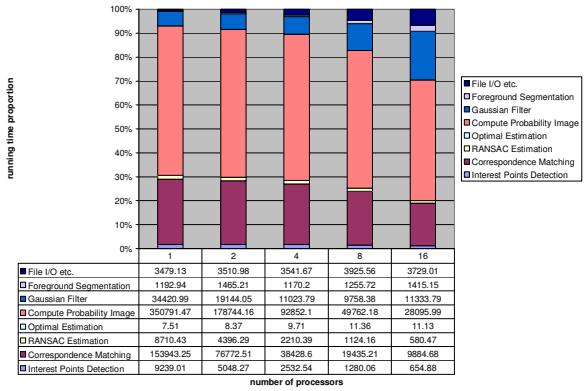



**Figure 4.3 Speedup results for the DMPA**




| | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|
| File I/O etc. | 3479.13 | 3510.98 | 3541.67 | 3925.56 | 3729.01 |
| Foreground Segmentation | 1192.94 | 1465.21 | 1170.2 | 1255.72 | 1415.15 |
| Gaussian Filter | 34420.99 | 19144.05 | 11023.79 | 9758.38 | 11333.79 |
| Compute Probability Image | 350791.47 | 178744.16 | 92852.1 | 49762.18 | 28095.99 |
| Optimal Estimation | 7.51 | 8.37 | 9.71 | 11.36 | 11.13 |
| RANSAC Estimation | 8710.43 | 4396.29 | 2210.39 | 1124.16 | 580.47 |
| Correspondence Matching | 153943.25 | 76772.51 | 38428.6 | 19435.21 | 9884.68 |
| Interest Points Detection | 9239.01 | 5048.27 | 2532.54 | 1280.06 | 654.88 |

**Figure 4.4 Proportion of the total execution time taken by various tasks in the DMPA**

**Table 4.2 CPU efficiency results for the DMPA**

| # processors | 2 | 4 | 8 | 16 |
|---|---|---|---|---|
| CPU efficiency | 96.9% | 92% | 80.2% | 61.9% |

## 4.3. Optimization

For reason of simplicity, we used the operation *MPI_Allgather()* in the DMPA implementation of the Gaussian filtering when the direction of data dependences changes from along the rows to along the columns. However, only 2 × 48 rows of data for each node are actually useful, whereas a total 480 – (480/NODES) – (2×48) rows of data per node during the communication are redundant. Note that the value of 48 rows is determined by the size of the Gaussian filter mask. To eliminate this redundancy, we use the following technique:

- Each node sends 48 rows of data to its *upper* and *lower* neighbors. The *upper* neighbor for the node 0 is the last node, and the *lower* neighbor for the last node is node 0.
- If a node has fewer than 48 rows of data, 2 or more nodes will be involved in sending these 48 rows data to the nodes *upper* and *lower.*
- All sending and receiving are non-blocking. This process is blocked until all the *receives* are completed.

In the original implementation, *MPI_Allgather()* is used at the end of the Gaussian filtering operation so that each processor will have a local copy of the filtered probability image. However, this is unnecessary because only the master node performs the dynamic foreground segmentation. Therefore, we further optimize the algorithm by only gathering the data in the master node. Figure 4.5 shows a significant improvement in reduction of the communication overhead, and Figure 4.6 shows a noticeable improvement in overall performance.

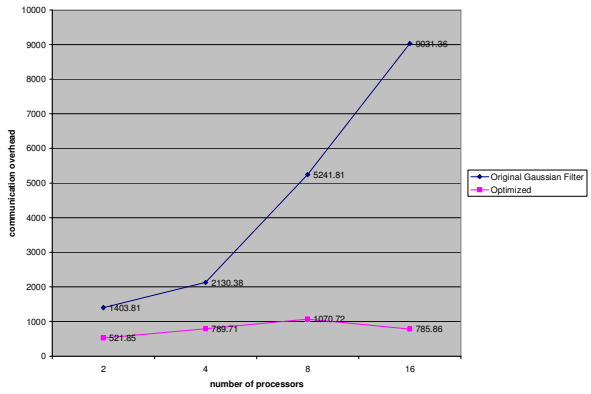



**Figure 4.5 Communication overhead for original and optimized Gaussian filtering**

## 5. Conclusions

Motion panorama construction is commonly viewed as an efficient representation for a video sequence for a wide range of applications. Although sequential algorithms for motion panorama construction yield very good results, they are computationally infeasible for real time applications. Parallelization is proposed as a means to speed up the motion panorama construction

process thus making it suitable for real time applications.

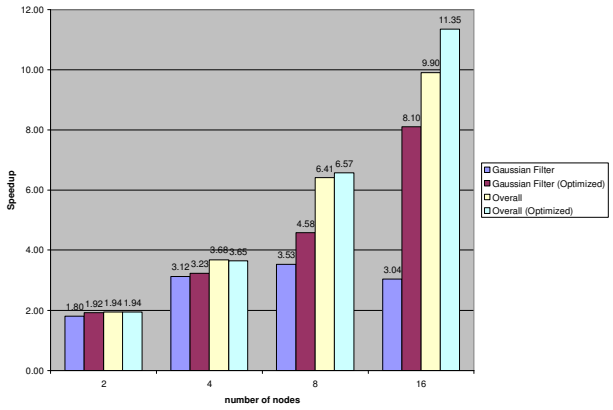



**Figure 4.6: Performance comparison for original and optimized Gaussian filtering**

In this paper, we have compared the SMPA and the DMPA, for motion panorama construction. In the SMPA, we proposed a dynamic workload distribution mechanism which leads to an algorithm with balanced workload. Good speedup results were achieved when the number of available processors ranges from 2 to 8. We predict that the SMPA will continue to work well on shared memory systems with up to 16 processors. In the case of the DMPA, we devised an optimized scheme for inter-processor communication when the Gaussian filter is applied on the probability image, which resulted in a noticeable improvement in terms of overall performance of the DMPA. Nonetheless, the DMPA was observed to have lower speedup and efficiency than the SMPA on account of the need for explicit inter-processor communication in case of the DMPA.

In terms of future work, we are currently working on a hybrid parallel algorithm, targeted for a cluster of shared-memory symmetric multiprocessors (SMPs) which provides a combination of both shared memory and distributed memory architectures.

## 6. References


[1] A. Bartoli, N. Dalal, B. Bose, and R. Horaud, From video sequences to motion panoramas, *Proc. IEEE Wkshp. Motion Video Comp.*, Dec. 2002, pp. 201- 207.

[2] M. Irani, S. Hsu, and P. Anandan, Mosaic based video compression, *Proc. SPIE Conference on Electronic Imaging, Digital Video Compression: Algorithms and Technologies*, Feb. 1995, vol. 2419, pp. 242-253.

[3] L. Teodosio, and W. Bender, Salient video stills: Content and context preserved, *Proc. ACM Intl. Conf. Multimedia*, 1993, pp. 39-46.

[4] M. Irani, P. Anandan, J. Bergen, R. Kumar, and S. Hsu, Efficient representations of video sequences and their applications, *Signal Processing: Image Communication*, May 1996, vol. 8(4), pp. 327-351.

[5] H.Y. Shum and R. Szeliski, Panoramic image mosaics, *Microsoft Research Tech. Report*, 1997, vol. MSR-TR-97-23.

[6] DARTFISH Ltd, DartTrainer Software, 2001/2002, http://www.dartfish.com/.

[7] Salient Stills Inc, VideoFOCUS Software, http://www.salientstills.com/.

[8] S.E. Chen, QuickTime VR - An Image-based Approach to Virtual Environment Navigation, *Proc. ACM SIGGRAPH*, 1995, pp. 29-38.

[9] S. Hsu, and P. Anandan, Hierarchical Representations for Mosaic Based Video Compression, *Proc. Picture Coding Symp.*, Mar. 1996, pp. 395-400.

[10] P.H.S. Torr. and A. Zisserman, Feature based methods for structure and motion estimation, *Vision Algorithms: Theory and Practice*, July 1999.

[11] M. Irani, and S. Peleg, Motion analysis for image enhancement: Resolution, occlusion, and transparency, *Jour. Visual Comm. Image Representation*, Dec. 1993, vol. 4, pp. 324–335.

[12] M. Irani, and S. Peleg, Improving resolution by image registration, *CVGIP: Graphical Models and Image Processing*, May 1991, vol. 53, pp. 231-239.

[13] J.Y.A. Wang and E.H. Adelson, Representing moving images with layers, *IEEE Trans. Image Processing,* September 1994, vol. 3, no. 5, pp. 625-638.

[14] X.Y. Pan, Motion panorama construction from streaming video for power-constrained mobile multimedia environments, *MSAI Thesis*, University of Georgia, Athens, Georgia, 2004.

[15] M. Pedro, Q. Aguiar, R. Jasinschi, José M. F. Moura, and C. Pluempitiwiriyawej, Content-based Image Sequence Representation, (ed. Todd Reed), *Digital Image Sequence Processing: Compression and Analysis*, CRC Press Handbook, Boca Raton, FL, 2004. Chapter 2, pp. 5-72.

[16] M. A. Fischler, and R. C. Bolles, Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography, *Comm. ACM,* 1981, vol. 24, pp. 381-395.

[17] P.S. Pacheco, *Parallel Programming With MPI,* Morgan Kaufmann, San Francisco, CA, 2003.

[18] G. Andrews, *Foundations of Multithreaded, Parallel, and Distributed Programming*, Addison Wesley, Reading, MA, 2000.