

Parallel Horizontal Cell Segmentation

**Nick Larusso
Brian Ruttenberg
Mike Crowell**

Final Project Report
CS 240A
Prof. John Gilbert
Spring 2007

Introduction

In the eye, horizontal cells exist only in a thin layer of the retina called the outer plexiform layer and provide horizontal connections between groups of photoreceptors. These cells consist of a round cell body with many processes extending outward which are called neurites. These neurites are what provide the connections to photoreceptors, as illustrated below in Figure 1.

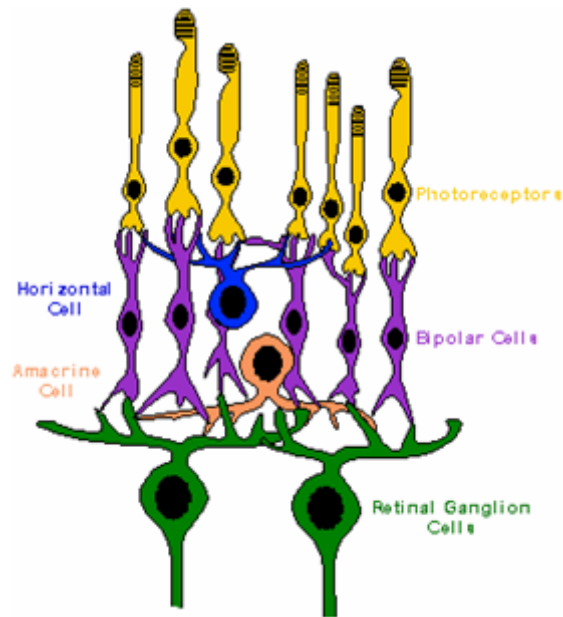


Figure 1: The Retinal Layers

We have been working in collaboration with Steve Fisher's Retinal Cell Biology Laboratory here at UCSB[1]. They have been collecting, preparing, and imaging wholemount retinal tissue samples and providing us with the data to segment and analyze. Preparation consists of injecting the tissue sample with a fluorescent protein that binds only to a particular cell type (horizontal cells). Imaging is done with a confocal light microscope which captures reflected light from the fluorescent proteins which are bound to the cells of interest. This method of imaging faces some challenges and does not always provide images with low signal-to-noise ratio.

As you can see from the image shown in Figure 2, picking out all of the neurites of a specific cell is a very difficult and time consuming task which is the main motivation for an automated segmentation method. Also, imaging methods are becoming more common in the biological sciences. This is especially true with the Center for Bioimage Informatics at UCSB which was initiated specifically to build an accessible database of biological images for large scale analysis. The goal of this project is to mine this database to identify interesting patterns among thousands of images that can aid in our understanding of biology and help test and formulate hypotheses. Large scale automated methods for image segmentation and analysis are a crucial step in this realization.

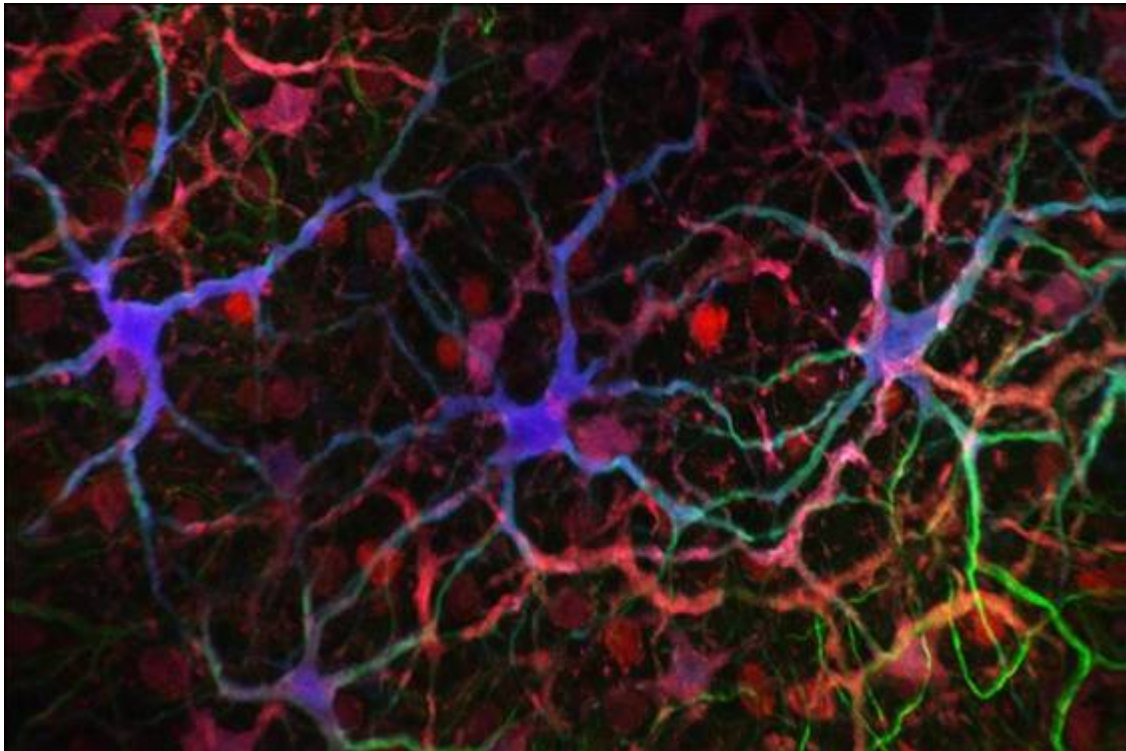


Figure 2: Example Horizontal Cell Image. The horizontal cells are dyed with anti-calbindin (blue) and anti-neurofilament (green)

Because the process of dying the cells and capturing two dimensional images provides us with uncertain data, we would like an algorithm that conveys that uncertainty through segmentation and analysis. Many image segmentation techniques work by thresholding some values so an answer is always given, but no measure of confidence in the algorithm is given. This is problematic because the uncertainty of how well the algorithm segmented the given region propagates through to the analysis of the cell, which makes it difficult to gauge how much confidence we should have in our results.

1. Algorithm

1.1. Original Work

The segmentation algorithm we use is built on work done by Ljosa and Singh[2] in which they use a Markov Random Walk with restarts for each cell they wish to segment. The image is treated as a graph where each pixel (except for those located on an edge of the image) has ten connections: its eight neighbors, the restart position and itself, as shown in Figure 3. The pixel intensities (we only process in gray-scale) act like edge weights such that a pixel with a higher intensity will have a higher probability of being visited than dull pixels. There is also a very small probability that we will return to the restart position at each pixel.

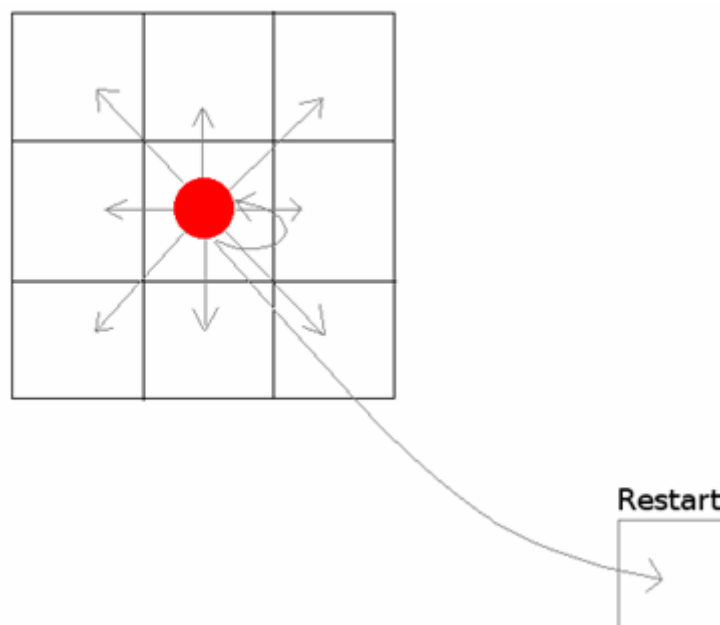


Figure 3: Possible next states for a given pixel

We would like to continue this process long enough so that we converge to the steady state of the random walk. To ensure steady state is reached consistently, it is possible to reformulate the problem in terms of solving for the largest eigenvector of the column normalized transition matrix. This algorithm works especially well for segmenting horizontal cells and, as described by Ljosa and Singh, it outperforms the typical watershed segmentation method[2][3]. We have taken the algorithm one step further to significantly increase the accuracy (up from 65% to as much as 90% on some images), however, this was done at the cost of severely increasing the problem size.

1.2. Improvements

Our first improvement to the segmentation algorithm was to simply take the location of the other cells in the image into account while doing the random walk. To do this we define a distance D from the cell center that is slightly less than the radius of the typical cell body. Then we can say anytime we are less than D pixels from a given cell center, we are definitely inside that cell. Since we know the locations of each cell center in the given image, we can say that if we are within D pixels of another cell, we should set our restart probability to 1 because we have wandered into another cell. We can go as far as to say, the farther you are from your cell center, the more likely it is that you've stepped into another cell, and thus scale the restart probability to slightly increase the chance of restarting. Instead of having a static restart probability across every pixel, we now change it depending on where we are in the image. This change, while extremely helpful in images with many cells close together, adds only a small constant time calculation to each step in building the transition matrix.

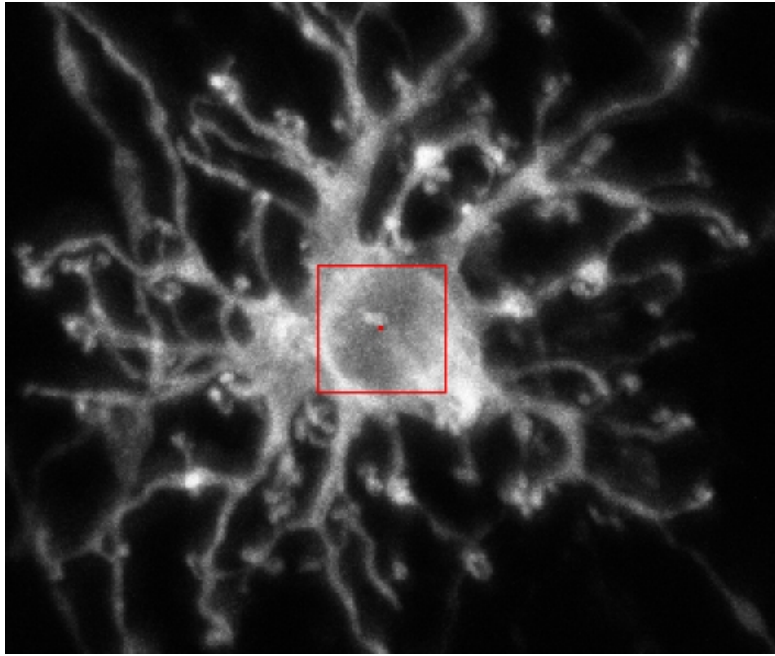


Figure 4: The cell center with zero restart probability box

For the second improvement, we again take knowledge from the domain and incorporate it into the algorithm. Here we try to solve the problem that occurs when neurites from two cells overlap. In this case (see Figure 5 below) we greatly increase the probability that we'll jump into the neurite of another cell because there are high pixel intensities all around. This can be avoided because neurites, in general, are relatively straight. So if we have knowledge of where we came from we can increase the probability of moving in a straight line and lower the probability of making any kind of sharp turn.

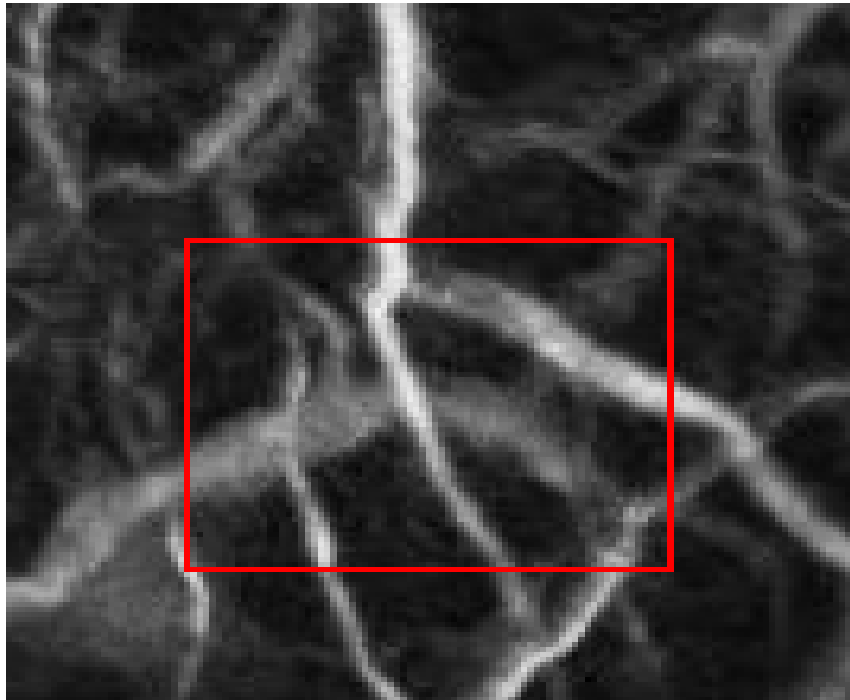


Figure 5: Example of two neurites overlapping

Since we are still solving an eigenvector problem, we are forced to increase the size of our transition matrix to add this sense of memory into the random walk. This is done by adding an additional eight rows and eight columns for each pixel in our transition matrix where each new column will represent being at that pixel having come from one of its neighboring pixels.

For example, let us consider the nine-pixel stencil below in Figure 6 and the columns of the transition matrix that would correspond to pixel five. The fifth column corresponding to pixel five would represent being in pixel five and having come from itself. The sixth column corresponding to pixel five would represent being in pixel five having come from pixel six, etc.

P1	P4	P7
P2	P5	P8
P3	P6	P9

Figure 6: 9-pixel stencil

2. Mathematical Formulation

2.1. Transition Matrix Characteristics

As was noted earlier, the transition matrix is formulated such that the eigenvector associated with its largest eigenvalue is the probabilistic solution to the cell segmentation problem. If we let A be the transition matrix for a given image we intend to analyze, then A is an n by n square matrix where n is equal to nine times the total number of pixels in the image. Recall that the dimension of A would simply be equal to the total number of pixels in the image if we were not using the "momentum" formulation described above in order to improve our segmentation results. Typical images that we want to analyze contain almost 400,000 pixels so even without the momentum formulation A could have 160 billion potential entries. With momentum A now typically has 13 trillion potential entries, so A is large by any standard.

Of course A is also quite sparse because from any given pixel there is only the possibility of moving to 10 other pixels in one step, as described earlier. Therefore each column of A has at most 10 nonzero entries. This also holds true for the extra columns added to include the momentum formulation. Thus a typical A has approximately 35 million nonzero entries out of a possible 13 trillion entries. Clearly it will still be difficult to even store A on a single computer and potentially quite computationally expensive to directly compute the eigenvector of A we are interested in. Even worse, we know that A is non-symmetric both due to the momentum formulation and the possibility of restarting to the center of the cell being segmented. The eigenvalue problem for non-symmetric matrixes is in general more difficult than for symmetric matrixes. Shown below in Figure 7 is a Matlab spy plot of an example of A for a small image. Note the fairly dense diagonal band and the single (completely) dense row corresponding to the restart probability each pixel in the image has to the cell center pixel.

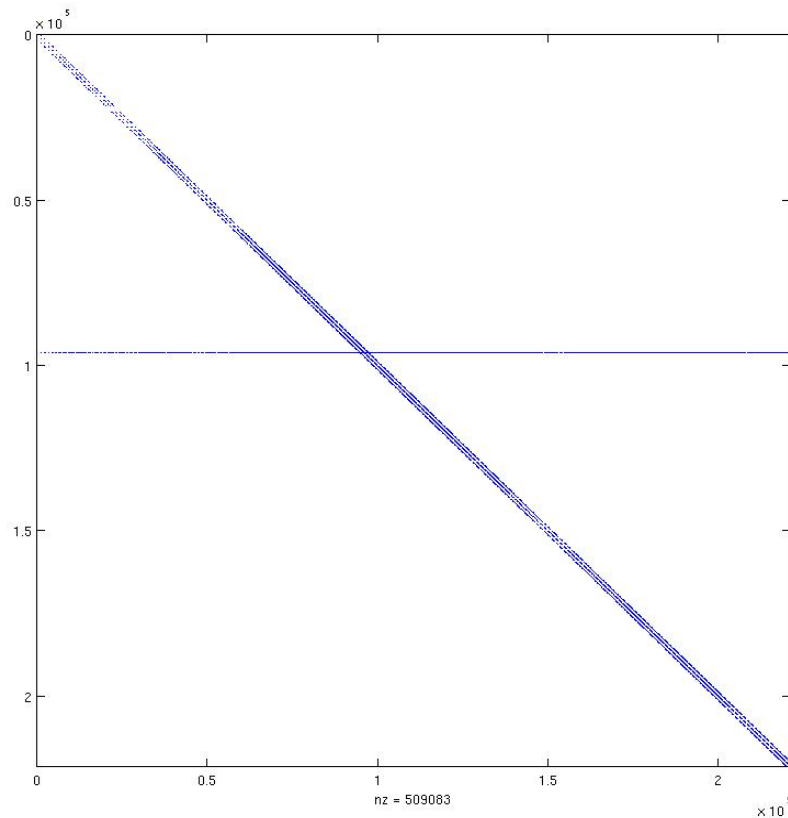


Figure 7: Spy plot of a transition matrix

2.2. Reformulation of the Eigenvector Calculation

To solve an eigenvector problem, we need to solve the matrix equation:

$$Ax = \lambda x$$

for x where λ is the largest eigenvalue of A and x is the corresponding eigenvector. Since A is column normalized to 1 we know that the largest eigenvalue, λ , is equal to 1. So we may rewrite the above equation as:

$$Ax = x$$

or equivalently:

$$(A - I)x = 0$$

Thus instead of solving the eigenvalue/vector problem we now need only solve the homogeneous linear equation above for x .

While this does simplify our problem, $(A - I)$ is still a very large non-symmetric matrix and solving the homogenous linear equation associated with it is highly nontrivial. The size of $(A - I)$ and its lack of symmetry dictate the choice of an iterative Krylov subspace based algorithm. This size also

dictates that the algorithm be implemented in parallel to obtain solutions with a reasonable run-time for our intended application. There are at least two such methods widely implemented in parallel that would appear to be suitable for our needs: GMRES[4] and BiCGSTAB[5]. Both methods can solve large, sparse, non-symmetric linear systems.

GMRES stands for the "generalized minimal residual method." It is based on the Arnoldi method which is itself a generalization of the Lanczos method to non-symmetric matrixes. The Lanczos method is based on the Power method but incorporates the idea of iteratively searching through orthogonal Krylov subspaces. BiCGSTAB stands for the "stabilized biconjugate gradient method." The Biconjugate Gradient method is a generalization of Conjugate Gradient for non-symmetric matrices. Conjugate Gradient is another method based on performing the Power method through a series of orthogonal Krylov subspaces, but with more stringent requirements on the matrix A than Lanczos. Interestingly, the "stabilization" part of BiCGSTAB is provided by using the GMRES algorithm locally between BiCG steps.

In the end there appear to be two choices when applying iterative Krylov subspace methods to large non-symmetric matrices. GMRES is an example of the first choice where you realize that guaranteeing the orthogonality of all search directions is prohibitive both in storage and computation, so you only store up to a certain number of directions and then you start over. BiCGSTAB is an example of the other approach where you do not guarantee orthogonality of the search directions and consequently do not have to store and reference them all at each step. In general the convergence behavior of each algorithm is not well understood for any given A , so in practice both approaches are usually tried to see which one performs best. In our case, BiCGSTAB provided the best and most consistent results. Therefore, we chose to use BiCGSTAB for our eigenvector solver.

3. Parallelization Techniques

3.1. Image and Matrix Decomposition

There were several ways in which it is possible to break this problem up into parallel processes. Ideally, each processor would receive an equal number of non-zeros to process, with as many of those values clustered around the portion of the resulting vector a processor owns in order to reduce the communication necessary in the BiCGSTAB portion of the solution. Determining the total number of non-zeros in the matrix is a non trivial task, and dividing those up equally among all the processors could be difficult since the matrix is not evenly distributed. However, without regard to the restart row which will be discussed later, dividing the image (not the matrix) up by rows leads to each processor having a roughly equal amount of non-zeros in the transition matrix. Therefore, we chose to implement a simple row distribution algorithm. This translates to each processor owning a block of rows of the transition matrix.

At runtime, the image is evenly divided into blocks of rows and each processor is assigned a block of rows that it owns. Since calculating the column values in the transition matrix only requires a pixel and its immediate 8 neighbors, each processor can construct its portion of the transition matrix without needing the rest of the image (except for 2 ghost rows that we add to each processor for the border cases). One caveat to the transition matrix construction is that the matrix is column normalized to 1. This means that the pixels around the edge of the block of rows will need to set values in the matrix that are outside of the portion of the matrix that it owns. This is generally not a major issue, as before the BiCGSTAB algorithm begins, all the processors exchange values that do not lie within their portion of the matrix.

3.2. Load Balancing

As was shown in the previous sections, the transition matrix has a horizontal restart row that extends the entire dimension of the matrix. Since we divide the image and matrix up by rows, one processor would have to handle the entire restart row, which is roughly one-tenth of the number of non-zeros in the entire matrix. Obviously, this presents some load balancing problems. We solved this problem by actually changing the structure of the matrix depending on the number of processors that the segmentation is run on.

We implemented a method that evenly divides the restart row onto each processor. In our random walk model, we can restart back to the center of the cell in one time step. However, if we slightly alter the random walk and restart back to the center of the cell in two time steps (with the same probability), we do not change the long term behavior of the walk. Therefore, when we divide up the matrix among processors, we can add an intermediate step in the restart path of the pixels on a given processor, without affecting the long term behavior of the walk.

Each processor adds one row and column to its own portion of the matrix. This is used as the “local” restart row. Every pixel in a processors portion of the matrix restarts back to this extra row. As every processor does this, we are increasing the size of the matrix by the number of processors. On the processor that actually contains the real or “global” restart, we add paths from each of the local restarts back to the global restart (with a probability of 1). This results in the processor with the restart pixel owning an additional (number of processors) non-zero entries in its portion of the matrix. Using this method, we are able to almost evenly distribute all the non-zeros over the processors.

The BiCGSTAB method was implemented using HYPRE, a high performance sparse linear solver using MPI. Almost all the communication was a result of the BiCGSTAB algorithm, with a very small amount coming from a small amount of overhead in building the transition matrix and reconstructing the final image.

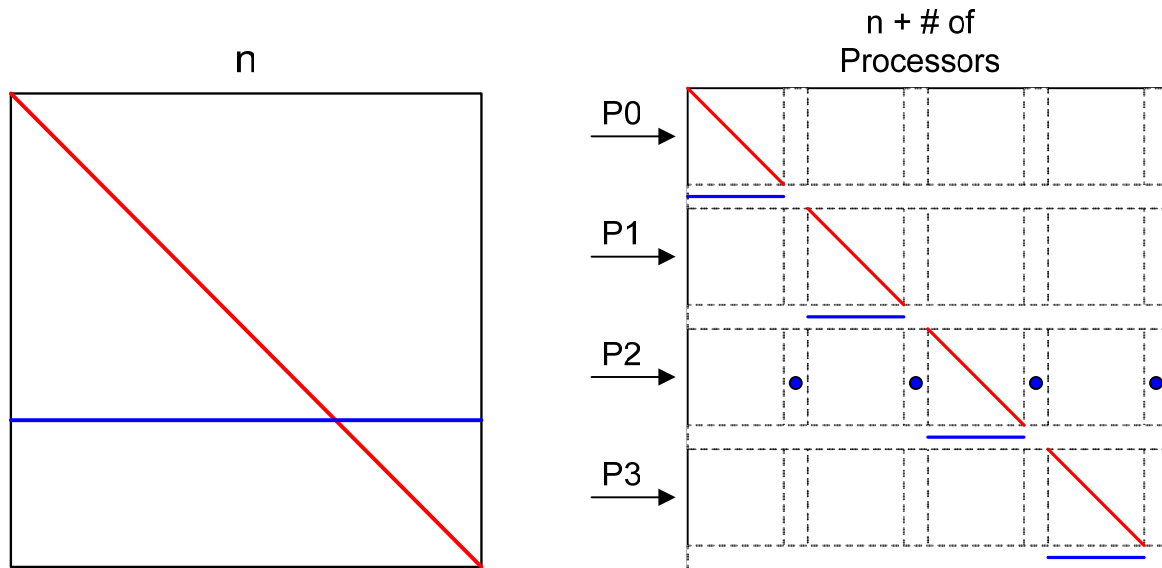


Figure 8: Original transition matrix and the load balanced transition matrix. The matrix on the left is the normal form of the transition matrix, with the restart row in blue. On the right is the processor balanced restart row for 4 processors. Each processor now owns a portion of the restart. The extra rows and columns that were added to the matrix can also be seen. The blue dots represent paths from the local restarts back to the global restart.

4. Results and Analysis

4.1. Experimentation Environment

All of the testing and analysis of the parallel probabilistic segmentation was performed on the Center for BioImage Informatics cluster. The cluster consists of 16 primary nodes and 16 secondary nodes. Each primary node consists of two dual core Intel processors running at 2.3 GHz, each with 4GB of RAM. Each secondary node consists of a single dual core Intel processor running at 3 GHz, with 2 GB of RAM. For consistency, most of the analysis and testing was performed on the primary nodes, with usage of the secondary nodes occurring when testing required more than 64 processors.

4.2. Previous Work

The original probabilistic segmentation algorithm was written and developed for Matlab. While Matlab certainly had programmatic advantages (such as sparse eigenvector commands), it also had severe limitations when running with extremely large matrices. Even with a compressed storage format, a full sized image (768 x 512 pixels) would still require over 500 MB to store in memory. In addition to the heavy memory usage, computing the eigenvector of the transition matrix proved to be computationally challenging for Matlab. On a serial version of Matlab, segmenting one cell took 110 minutes. On a parallel version of Matlab using Star-P, it took 30 minutes on 4 processors¹.

¹ Thanks to Viral Shah for testing on Star-P

This probabilistic segmentation is geared to be a practical tool for biologists to use. It is quite unreasonable then to expect a biologist to wait up to 2 hours for the results of segmenting one cell (especially when images typically have several cells). Clearly, there was a real and practical need to parallelize this algorithm in order to bring its runtime down to a reasonable level.

4.3. Speedup and Efficiency Analysis

We broke up our analysis of speedup and efficiency into two different metrics. First, we looked at the speedup and efficiency of building the transition matrix. Separately from that, we analyzed the speedup and efficiency of the BiCGSTAB algorithm. The primary reason this is done was due to the restart row load balancing discussed in the previous section.

Run time balancing of the restart row actually changes the structure of the transition matrix, and hence can (and usually does) affect the convergence behavior of BiCGSTAB. As such, in some cases increasing the number of processors would also increase the number of iterations BiCGSTAB needed to converge. Therefore, in order to account for the differences in runtime due to varying convergence times, we normalized the BiCGSTAB time by the number of iterations performed. This gave us a rate of time per BiCGSTAB iteration, which we could then compare across different configurations of processors.

Figures 9 and 10 show the speedup and efficiency of the transition matrix and the BiCGSTAB eigenvector calculation. Because of the sheer memory usage and processing requirements of this algorithm, our results look exceptionally good when compared with one processor. Given how prohibitive the running time is on one processor, we don't expect this algorithm to be run on less than four processors.

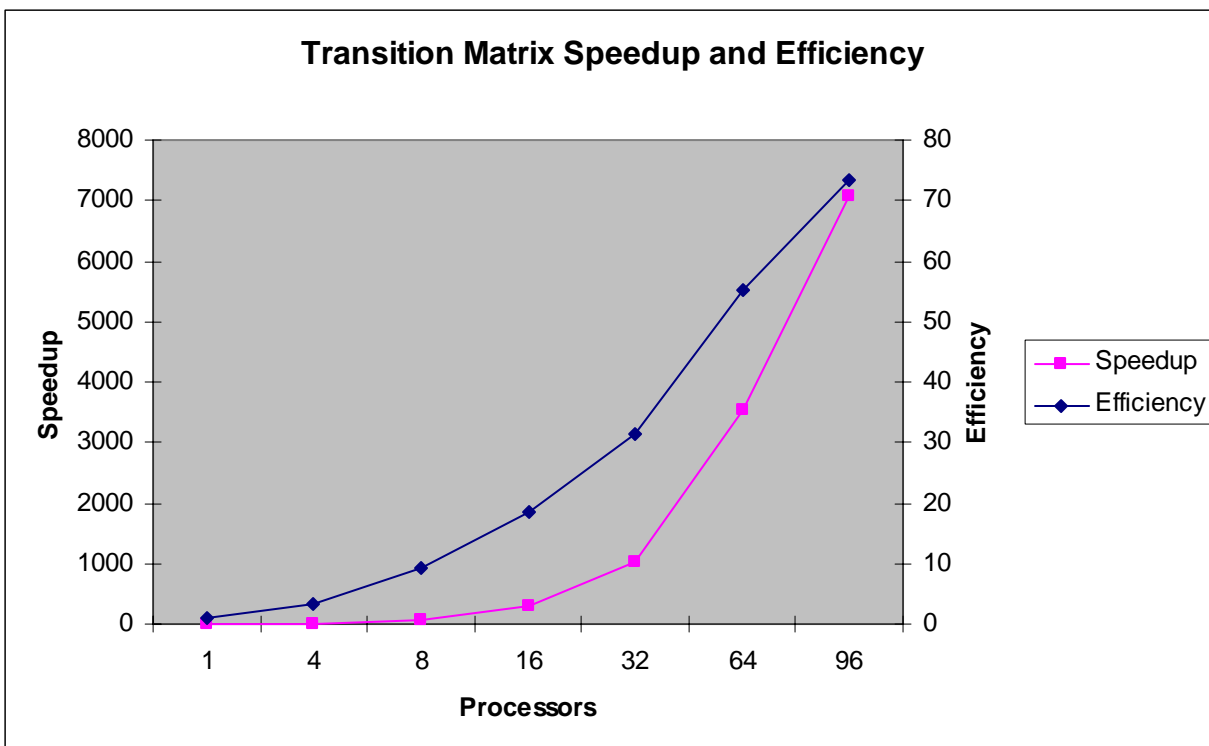


Figure 9: Transition matrix Speedup and Efficiency

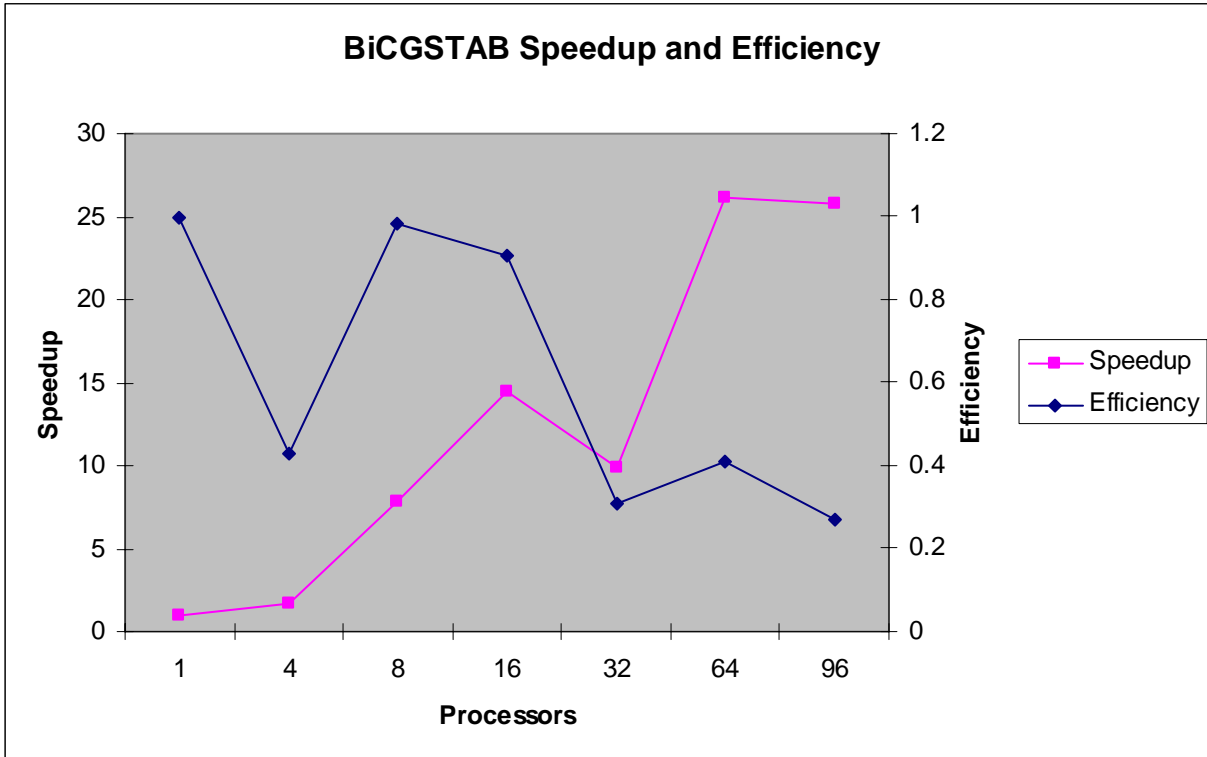


Figure 10: BiCGSTAB Speedup and Efficiency

In the figures, it can be seen that the speedup and efficiency of the transition matrix scales extremely well even up to 96 processors. Building the transition matrix is a highly parallel operation, since it requires almost no communication.

The BiCGSTAB speedup and efficiency is slightly different. Not surprisingly, the speedup scales very well with the number of processors. The efficiency however, clearly drops after 8 processors. This is expected as the structure of our transition matrix predicates communication among processors between BiCGSTAB iterations. Figure 11 shows the amount of communication per BiCGSTAB iteration under different configurations. As we increase the number of processors, the communication starts to dominate the efficiency. In the 96 processor configuration however, we are forced to use the secondary cluster nodes as well, so some overhead might be incurred by different architectures and networks interacting.

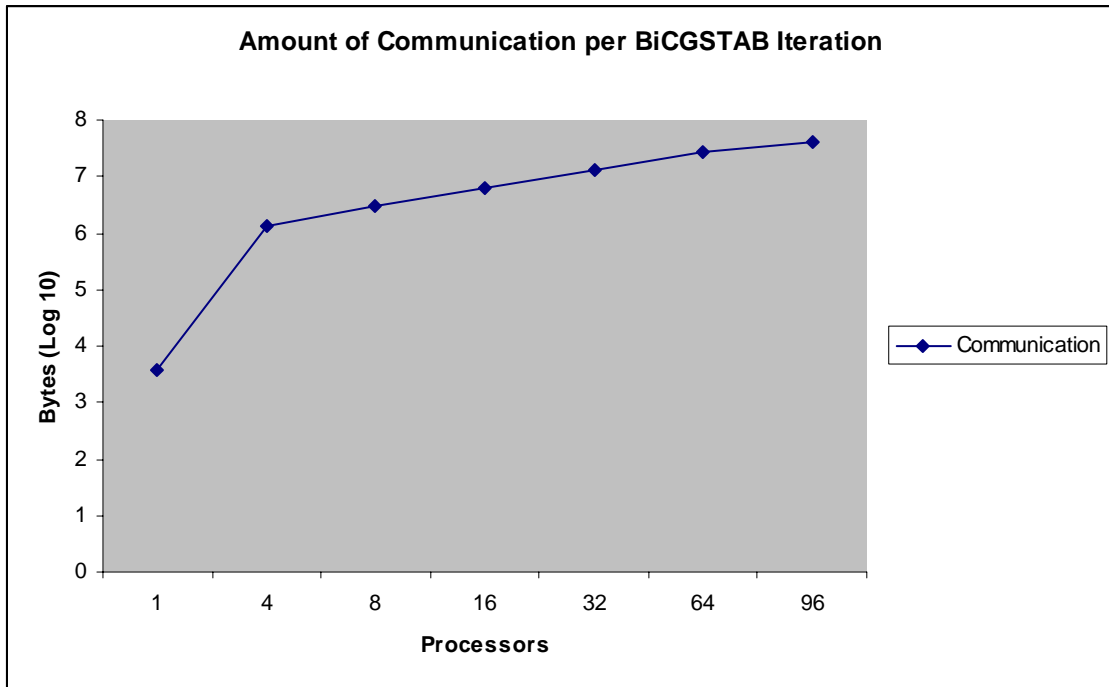


Figure 11: Communication per BiCGSTAB iteration

4.4. Scaled Speedup and Efficiency

In order to gain a better understanding of how our algorithm scales, we conducted a scaled speedup and efficiency analysis. Determining how to do a scaled speedup experiment for this problem is not necessarily straightforward. As one increases the size of the image, the size of the matrix and the number of non-zeros in the transition matrix increase quadratically. Unfortunately, it is not possible to scale the number of processors we have quadratically, therefore we had to take a different approach.

We decided to base our scaled speedup analysis on the number of non-zeros in the transition matrix. The size of the image is held constant, and a fixed sized vertical stripe of white pixels is added to the image each time the number of processors increases. This allows us to scale the number of processors linearly as the number of non-zeros increases linearly.

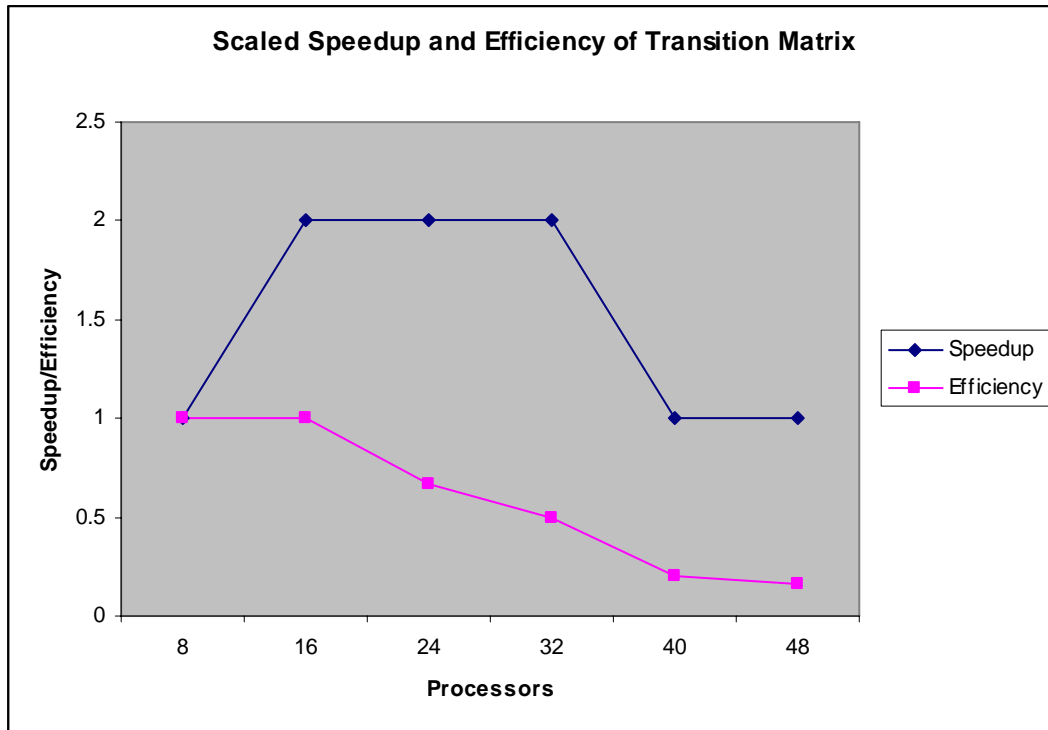


Figure 12: Transition matrix scaled speedup and efficiency

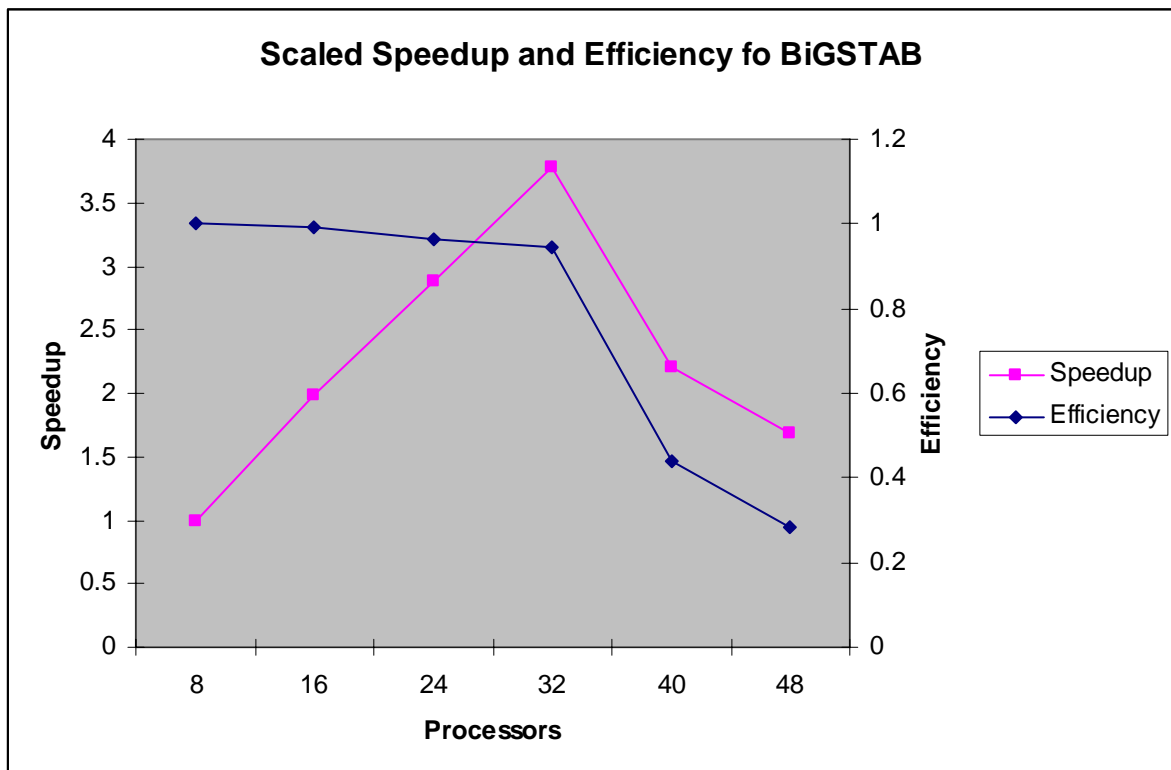


Figure 13: BiCGSTAB scaled speedup and efficiency

The results of our scaled experiments show that the performance of our segmentation does not scale linearly. It is certainly expected that the transition matrix speedup and efficiency start dropping. Construction of the transition matrix is subject to diminishing returns as creating the transition matrix incurs a certain amount of overhead which does not scale with the number of processors (such as opening the image).

The BiCGSTAB plot on the other hand starts to sharply drop off after 32 processors. We do not exactly know why this is the case, especially when looking at the plot of the communication per BiCGSTAB iteration. The communication overhead scales very well.

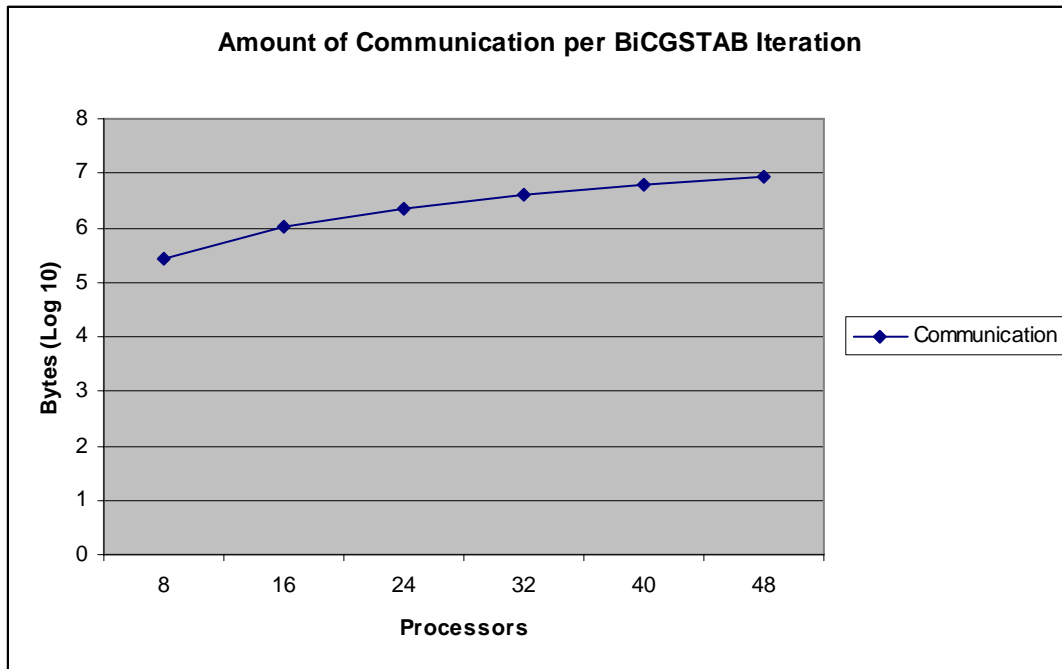


Figure14: Communication per BiCGSTAB iteration scaled

4.5. Optimal Configuration

Certainly running the probabilistic segmentation in the shortest amount of time is the optimal scenario. As seen from the figures, running with 96 processors results in the shortest wall clock time to perform the segmentation. However, in order to use our computing resources more responsibly, one of our goals was to find the optimal number of processors that would balance speedup, efficiency and wall clock time.

To accomplish that we generated kiviart charts for different configurations. The kiviart chart plots the amount of time each processor is blocked waiting for communication as well as the amount of time spent in non-blocking processes. The kiviart charts for four configurations are shown below in Figure 15.

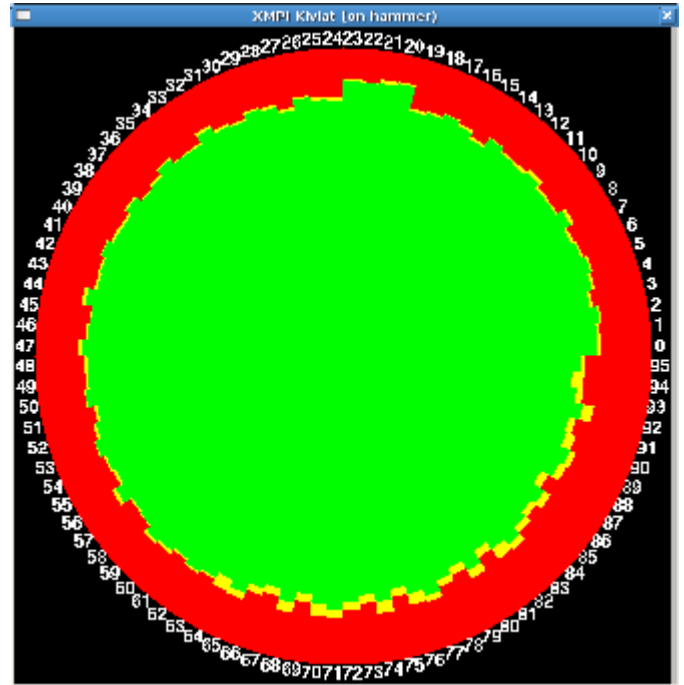
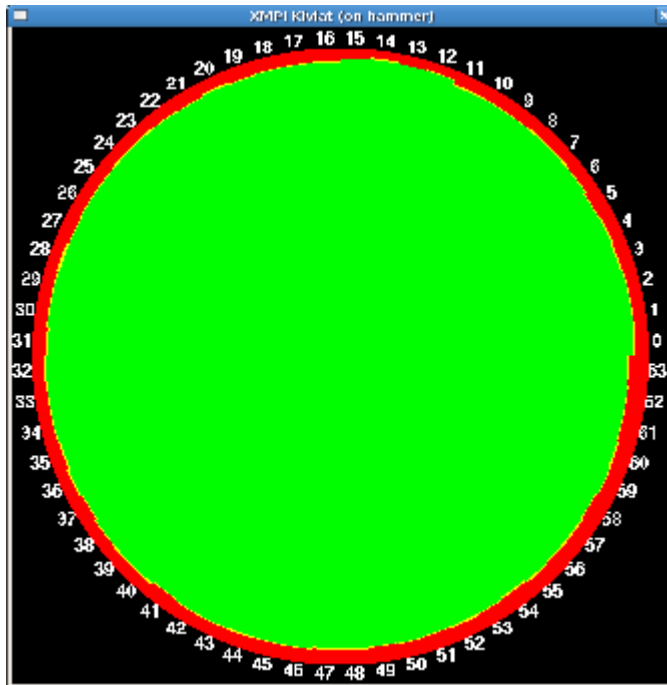
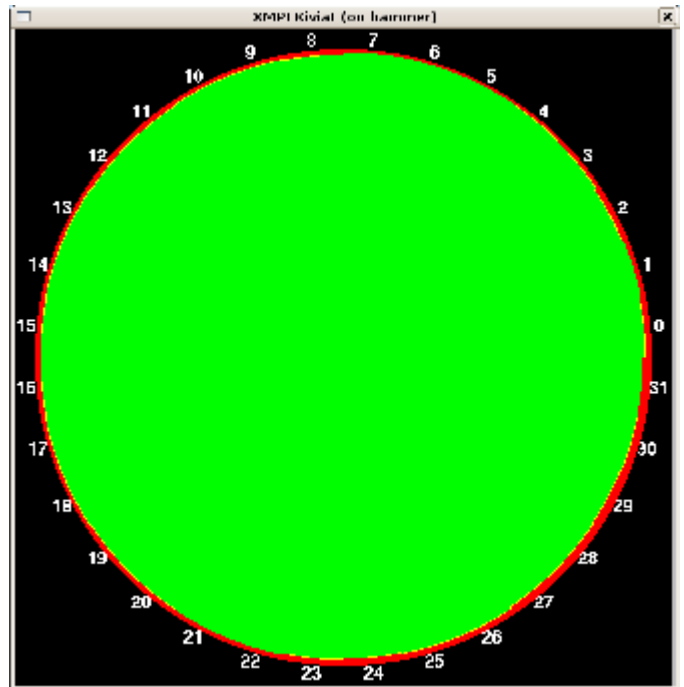
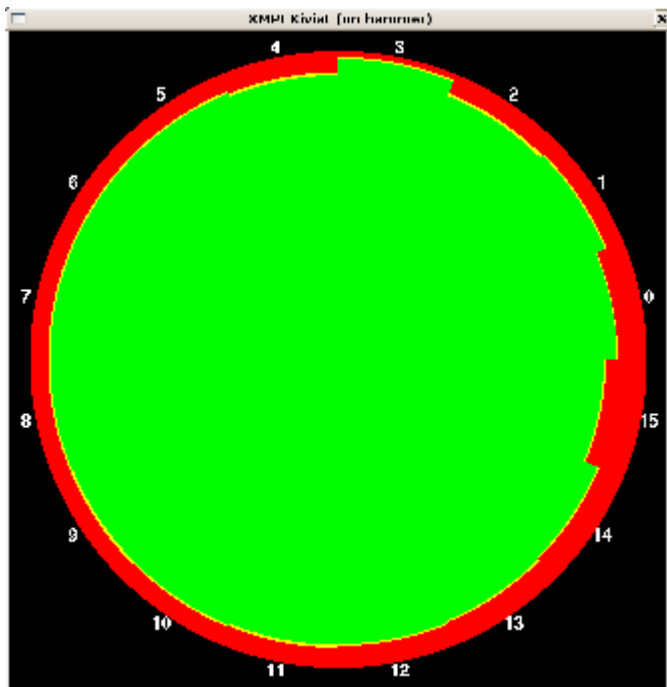


Figure15: Kiviat charts for 16, 32 (top row), 64 and 96 (bottom row) processors

The kiviats charts show very clearly which configurations are the most efficient. While the 96 processor configuration was fastest in terms of wall clock time, it is highly inefficient. Some of the poor performance of the 96 node case is certainly due to the use of two different machines. The 32 or 64 node configurations on the other hand, produced little communication overhead while distributing the blocking time almost equally among all processors. Given that the running time difference between the 64 and 96 node case is only approximately 16 seconds, it appears that the most efficient configuration to run this segmentation algorithm on is 64 processors.

5. References

- [1] Fisher, S. K, Lewis, G. P., Linberg, K. A., Barawid, E., and Verardo, M. R. (2005). Cellular Remodeling in Mammalian Retina Induced by Retinal Detachment. Retrieved on June 6, 2007, from WebVision website: <http://webvision.med.utah.edu/Fisher.html#Introduction>
- [2] Ljosa, V. and Singh, A.K., (2006) Probabilistic Segmentation and Analysis of Horizontal Cells. Proceedings of the 2006 IEEE International Conference on Data Mining (ICDM), 980-985.
- [3] L. Vincent and P. Soille. Watersheds in digital efficient algorithm based on immersion simulations. IEEE Trans. PAMI, 13:583–598, 1991.
- [4] Y. Saad and M.H. Schultz, "GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems", *SIAM J. Sci. Stat. Comput.*, **7**:856-869, 1986.
- [5] van der Vorst, H.A., "BI-CGSTAB: A fast and smoothly converging variant of BI-CG for the solution of nonsymmetric linear systems," *SIAM J. Sci. Stat. Comput.*, March 1992, Vol. 13, No. 2, pp. 631-644.