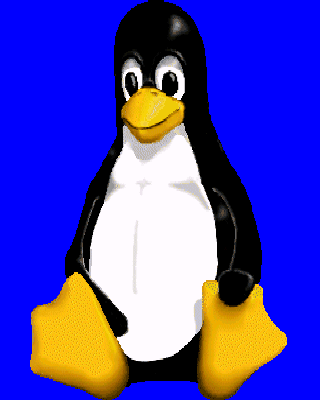
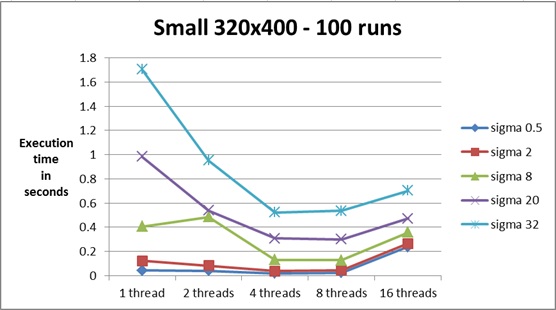
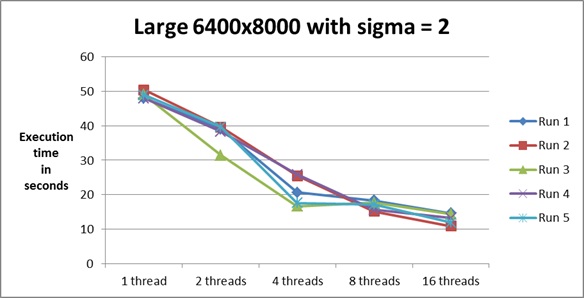
Gaussian Blur

Guide to source code  
  
The source code for Gaussian Blur compiles on the SEASnet linux machines and it correctly blurs the input image in sequential execution and in parallel execution.  The thread code, using OpenMP, is located around each double for loop block where the horizontal and then vertical blur is computed for each pixel.  This means that there is a pragma to launch the threads before the first for loops and then two pragmas, one for each block of for loops to divide up the pixels between the threads.  
  
Project breakdown  
  
The decision to parallelize the code this way was decided on the knowledge that the computation of the blur of each pixel is independent of every other pixel.  This meant that a large amount of independent work was present which is ideal for a data decomposition speed up using OpenMP.  Despite all of this parallelization some synchronization was needed between the two for loop blocks.  Here a barrier statement was inserted to make all threads wait for the intermediary image to be computed. This section of source code was worked on by Andrew Bax.  On the gaussian blur function no completeness features of implementation were sacrificed but it was theorized that a functional decomposition could be implemented  with the computation of the intermediary image and the final result using some synchronization to notify when a particular section is able to be computed.  As the implementation for this proposed addition was very complex and the proof of parallel strength in image computation was shown it was shelved to focus on the presentation and report section of the project.  
  
Bugs & Challenges  
  
There were not very many bugs in the implementation of this function. One of the few bugs was originally perceived as a correctness issue with running the code in parallel on a local linux machine but when moved to the SEAS machines the code crashed and segfaulted immediately.  After a short review of the parallel additions in the pragmas it was realized that an index variable for the  gaussian kernel was not private for each thread.  Once this was fixed the code ran on the SEAS machines and produces identical images to the sequential implementation.  The real challenge with the OpenMP framework is ensuring that each variable is properly shared or set to private as it can greatly affect the execution of the parallel code.  The only difficulty in extracting performance from the threads was when an input image would be too small for the gain in parallelization to be counteracted completely by the overhead to launch the threads.  This was seen on the 400x320 image with 8 or more threads.  
  
  
  
Final Results  
  
The parallelized code showed good gains in execution compared to the sequential version with the exception of small images with a large number of executing threads.  Below is a input image with the produced output and graphs charting the execution time of different size images with a spectrum of execution threads.

Input image is shown on the left and the Gaussian blur of this image with a sigma of 2 is show on the right.

Below are the results for a small 320x400 pixel image with the average of 100 runs for each thread number and sigma value pair. There are good gains in parallelization up to 4 threads on average with speed up being better for higher sigma values where there is more work for each thread. At 8 and 16 threads and likely greater thread counts, the execution slows down and performs worse than with 4 threads. This is believed to be caused by thread overhead being too large for the small amount of work to be done.

Next are the results on an imaged scaled up 20 times, in both width and height. Due to the long read and write times for this image size running the Gaussian blur numerous times was impractical. The noticeable trait of this size image is that for 8 and 16 threads the execution of the blur was faster than small number of threads due to the large amount of independent work able to be done by threads.

Finally, here are some runs of the large image with a sigma of 20. This showed similar performance gains if not better than a sigma of 2 with a same sized image. At 8 threads the execution time of the 5 runs varied greatly which with more time and a little automation could be further tested with many more runs on that sample set.

