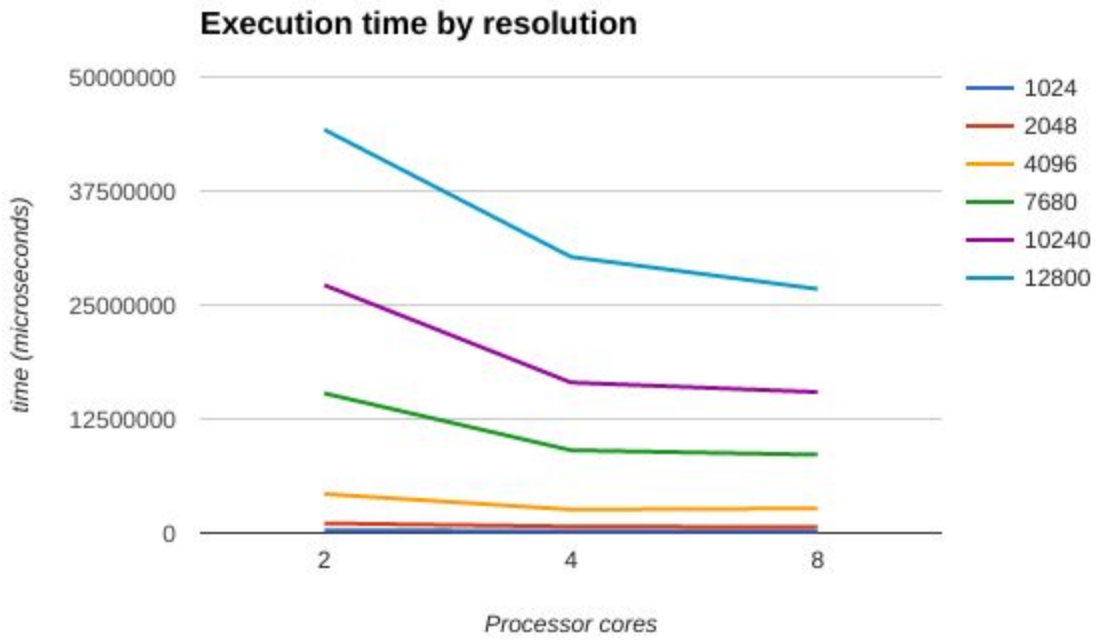
Cameron Franke
4/1/17
High Performance Computing



*The statistics in the chart above are average execution times after 10 iterations of each test.

Abstract

The purpose of this project is to evaluate the parallel efficiency of an edge detection algorithm. The edge detection algorithm in question consists of 5 steps at the end of which an edge image is created. The algorithm is parallelized with the Message Passing Interface (MPI) multiprocessing framework.

Introduction

The first step of the edge detection algorithm is to create horizontal and vertical gradient images. This is done by convolving the images with a gaussian filter. The second step is to use the horizontal and vertical gradient images, along with the provided mathematical equation, to produce magnitude and phase images. The third step is to us the magnitude and phase image to create a suppression image. From the suppression images a hysteresis image can be derived. This images involves amplifying some pixels and blacking out others. Finally, from the Hysteresis image, an edge image can be created.

Implementation

The first step in my parallelization implementation is to broadcast some basic information to all workers. The initial broadcasts contain the image dimensions, gaussian filter dimensions and the gaussian filters themselves. After that the initial image is scattered to all of the workers. After receiving their chunks the workers proceed to swap ‘ghost rows’ with each other. This is done so that the root process is not overloaded and so that the vertical gradient images can be created properly. After the swapping takes place the workers create their horizontal and vertical gradients on their own and then all of the chunks of both images are gathered by the root process. The next few images are created in the same manner; each process computes its chunk and then the root gathers the pieces. Magnitude, phase, hysteresis and suppression images are created with this pattern. After these images are created the processes once again swap ghost rows. This is done because the final edge image is created by referencing the neighbors of each pixel. This necessitates the swapping of the top and bottom rows of each processes chunk. After each process swaps rows and creates its final edge image, there is one last gather and the final images are written to files by the root process.

Results and Analysis

As you can see in the graph above the speedup of MPI at this scale is fairly poor relative to other multiprocessing frameworks like Pthreads and OpenMP. The 2-worker run of the algorithm showed a 1.65x speedup over the serial version at 1024x1024 and at 4096x4096 the 2-worker execution showed a 1.8x speedup. From just this example we can see that the relative cost of communication decreases as the problem size increases. At 4096x4096 the 8-worker execution performed 2.9x faster than the serial implementation.

Conclusion

MPI is clearly meant for large computing systems like clusters or supercomputers. This obvious from the fact that is is a distributed memory framework but it is made more obvious by the enormous communication overhead displayed by this algorithm. Although the speedups achieved here to not rival the speedups previously seen by other multiprocessing frameworks, MPI is clearly the best choice for large problems being solved on large clusters.