Final Analysis

In order to assess the performance of this parallel BFS implementation, I conducted a scaling study where each problem is run 10,000 times.

	Num Processes	Avg Time	Parents per sec
64 edges, 4 keys	4	4.79164e-05	333,915
128 edges, 8 keys	4	0.000182053	351,546
256 edges, 16 keys	4	0.000973461	262,462
64 edges, 4 keys	2	6.11174e-05	255,541
128 edges, 8 keys	2	0.000301922	211,975
256 edges, 16 keys	2	0.00175263	136,937
64 edges, 4 keys	1	6.68331e-05	239,402
128 edges, 8 keys	1	0.00050936	125,648
256 edges, 16 keys	1	0.00404472	59,336.6

The first noteworthy observation is the increase in performance when we increase the number of processes from 1 to 2. Looking at the problem of size 256 edges, there was a 231% improvement in terms of parents per second. The change is 442% when we compare the serial version with the same code run with 4 processes. While the problem size stays constant, doubling the number of processes suggests a two-fold improvement in performance. This finding falls in line with expectations prior to implementation: since we are distributing the number of vertices evenly across processes, O(n/p) work can be completed in parallel with some communication overhead.

In the *Alltoallv* stage of the graph search, the size of the input can be *O(E),* the total number of edges, in the worst case. I hypothesized that the communication cost *paN + (m/p) βN,a2a* would become a larger hindrance as the ratio of E/p grows. Looking at the three sets problems summarized, we can see that as the number of edges increases, the parents/sec processed consistently decreases. These findings lead me to reaffirm that as E/p grows, the chance for uneven load balancing across processes increases. As a result, all-to-all blocking communications such as *Alltoallv* experience larger discrepancies in wait time, and take longer to communicate once they are able to proceed. While the problem sizes tested are probably not generalizable, I believe this phenomenon is in part responsible for the decrease in performance as the problem size scales.

To address the issue of load balancing further, I would implement a 2-D partition of the adjacency matrix, as opposed to solely a partition across the rows. In the 1-D partition case, an edge (u,v) is distributed according to the u index. This leaves the distribution of the vertices v highly responsible for the poor distribution of edges: if all v are in the index range owned by processor 1, processing the frontier after all-to-all communication would leave the processes poorly balanced. A 2-D partition would help alleviate this issue by diversifying both u and v vertices across a mesh of processors. The ensuing consequence would be better and more consistent load balancing across problem sizes and E/p ratios, as well as less variance in time spent in all-to-all communications.