**ParConnect Reproducibility Report**

**Supercomputes**

**Introduction**

The paper describes a new, distributed algorithm for finding connected components in graphs, specifically for de Bruijn graphs for metagenomic assembly. A bucketing algorithm based on tuples representing undirected edges in the graph is used to identify connected components. The number of edges considered is initially distributed evenly across the MPI ranks. A sorting and reverse procedure allows the ranks to find nodes that are connected between MPI ranks, by checking the last element after sort. The authors further describe an *active partition* approach where tuples that are part of a component that is not connected to any other nodes are removed. This allows the algorithm to focus on finding new components, while ignoring tuples that are part of a connected component and thereby reducing workload. This is the AP algorithm. A further enhancement of AP is proposed where after each BFS iteration, the tuples are redistributed so that all ranks are load balanced. This is the AP\_LB algorithm. The authors' claims are that the active partition makes the distributed BFS connected-component algorithm faster. By adding load balancing, the algorithm runs faster, as all ranks are working on the same amount of data. We are testing the authors’ assertion that the naive approach is slower than the active partition approach and load balancing further enhances the active partition algorithm. In addition, we are testing the weak scaling behavior of the algorithms.

**Cluster Details**

We have two Dell PowerEdge C1430 nodes. Each node has two Intel Xeon E5-2640 v4 CPUs clocked at 2.40 GHz. Each socket has 64 GB of DDR4 memory clocked at 2133 MHz, with a total of 256 GB of memory across the entire cluster. Our cluster was supplied by Dell with support from Intel and Nvidia. For our interconnect, we use four Intel Omni-Path Fabric 100 series cards. We have one Omni-Path card tied to each socket’s PCIe 3 bus with 8 lanes, this gives a theoretical bandwidth of 7.88 GB/s per connection. The cards are connected directly in a dual-plane configuration between the two nodes, forming a ring topology between the four sockets. The cluster runs CentOS 7 with Linux kernel version 3.10.0-327.36.3.el7.x86\_64.

**Compiling ParConnect**

To compile ParConnect we cloned the repository as indicated in the instructions. We use cmake to configure the build with **-DBENCHMARK\_ENABLE\_CONN=ON** to enable the benchmarking program. We made a modification to the kmer reader to keep track of the number of reads per MPI rank. In benchmark\_parconnect we reduced the measured reads to print out the total reads in case we had to do Figure 1. We made no changes to the core algorithm. We pre-compiled ParConnect with each of the three algorithms in preparation for the competition. During our testing phase, we tried OpenMPI, MVAPICH and Intel MPI. We ultimately settled on Intel MPI because we found it worked best with our Omni-Path interconnect. We encountered no issues when compiling ParConnect on our machine. We used GCC 4.9.2 for our compiler and Intel MPI 5.1.3 Build 20160120 for our MPI runtime.

**Results**

|  |  |  |
| --- | --- | --- |
|  | **Edge Count** | **Connected Components** |
| **small.fastq** | 90665399 | 1929553 |
| **large.fastq** | 442461650 | 9006212 |

Table : Edge counts and connected component counts for the two data sets

|  |  |  |  |
| --- | --- | --- | --- |
| Time (sec) | **Computation and communication runtime along the iterations** | | |
|  |  |  |
| **Iterations** | | |

Figure : Time spent during communication and computation on **large.fastq** for each algorithm on 40 processes

|  |  |
| --- | --- |
| Naïve | /Users/marksb/Desktop/Naive.png |
| AP | /Users/marksb/Desktop/AP.png |
| AP\_LB | /Users/marksb/Desktop/AP_LB.png |

Figure : Allinea MAP compute (green) and communication (blue) times

For computation time and communication time on **large.fastq**, we see results similar to those in the paper. The Naïve algorithm is slowly falling as the number of iterations progresses, and for AP and AP\_LB, we can see slight reductions in times in each iteration. The biggest drop off in times is most apparent in iterations 6-9, but overall LB uses less time per iteration. Comparing Forge results with the measured times from within **benchmark\_parconnect**, we see a big discrepancy in the percentage of time spent communicating versus computing. In particular, the printouts from the tool indicate that MPI takes up approximately 20% of the time, however the profiler indicates that 44% of the time is in MPI calls. From Forge MAP’s Application, timeline we can see pthread activity during some MPI portions of the execution. This indicates that computations are started while the main thread is executing an MPI call. It is also possible that some of the non-communication timer items include MPI calls, and we are counting these as

compute time.

|  |  |  |  |
| --- | --- | --- | --- |
| Count of tuples (millions) | **Balance of work across processors along the iterations** | | |
|  |  |  |
| **Iterations** | | |

Figure : Min, mean and max tuple counts per iteration for each algorithm on 40 processes

For the active tuple count we see behavior consistent with that of the paper. The naïve algorithm maintains the same number of tuples through all BFS iterations. Active partition shows the number of tuples dropping at iteration 5. This is consistent with the algorithm finding connected components that have no out-going edges to other active tuples. Notably, we see an increasing load balance beginning at iteration 5. The mean number of tuples in iteration 6-9 indicate most ranks have almost no tuples while there are a few ranks with about 5 million tuples. Finally, the AP\_LB algorithm shows that all processes maintain the same number of active tuples as indicated by the maximum and minimum. Specifically, the difference between the minimum number of tuples and maximum number of tuples is one throughout all iterations. This is consistent with the algorithm balancing the work between all ranks.

*Figure 4*:Runtime for **small.fastq** on 4, 8, 16, 32 and 40 processes

For the scaling study, we can clearly see that the naïve algorithm takes approximately twice as long as either AP or AP\_LB, consistent with the paper’s claim that removing connected components speeds up the algorithm. A substantial divergence from the paper is apparent from this experiment; AP and AP\_LB perform nearly identically. While not visible in the graph, the maximum runtime for AP\_LB is always about two seconds faster than the minimum runtime of AP. This is not consistent with the paper’s claim for load balancing improving performance: the graph suggests AP\_LB should be about 1.7 times faster than AP. Figure 5 shows an expanded graph for the run times of AP and AP\_LB to show the difference in their run times.

Figure 5: AP and AP\_LB runtimes

**Conclusion**

Many of the claims in the paper were reproduced in this study. The active partition algorithm does reduce the number of work items per process as claimed in the paper. Figure 1 shows that this approach improves performance relative to the naïve algorithm. Figure 3 shows that the algorithm balances the load of tuples between the processes as seen in the AP\_LB plot. We also see in the scaling study that the runtimes decrease as we move from the naïve algorithm to the active partition algorithm, and another improvement when moving to the active partition with load balancing algorithm.

In the scaling study, we did not witness a significant speed up when moving from AP to AP\_LB. This could be caused by a number of factors. It is possible the data set is too small to benefit from the load balancing approach, or the data set’s components are constructed in such a way that load balancing does not help. Another possible cause is the specific configuration of our machine. We have two high bandwidth paths between each socket, requiring at most one hop across the Intel QuickPath Interconnect to get data to a socket on another node. Our machine is not like the machine used in the paper where there were a number of machines used and MPI communication latencies and bandwidth may be constrained. Our cluster could be concealing the cost of the MPI communication versus computation time. For 8 MPI ranks on iteration six, AP has min, mean, max tuples of 5410, 153231, 944395 respectively. For comparison, AP\_LB has 153231 active tuples on iteration 6 with 8 MPI ranks. The time to get from iteration 6 to iteration 7 is .4s and .08s respectively. Consistent with the maximum tuple load in AP being approximately 5 times higher than the maximum load of AP\_LB. From iteration 7 the algorithm quickly ends, and this is likely why we are not seeing the promised speed up from load balancing.