MPI+OpenMP Parallel All to All Shortest Distance

ofo5108 – Ömer Faruk Özdemir

Parallelization

In my implementation, I parallelize the given sequential Floyd-Warshall algorithm. I partition the distance matrix linearly and run a part of an iteration on each node.

```
for (int i = myStart; i < myEnd; ++i) {
                for (int j = 0; j < n/2; ++j) {
                    int w;
                    if ((w = local[(i-myStart)*n + k] + kth[j]) < local[(i-myStart)*n +
j])
                        local[(i-myStart)*n + j] = w;
                }
```

I chose the same algorithm with the sequential version to have a fair comparison between parallel and sequential performance.

Optimizations:

A. Each node holds a part of the data. Used uint8_t type to decrease memory usage.

```
1. base = (vertex_count+node_count-1)/node_count;
2. local = (uint8_t *) aligned_alloc(ALIGNMENT, sizeof(uint8_t) * base * n);
3. kth = (uint8_t *) aligned_alloc(ALIGNMENT, sizeof(uint8_t) * n);
```

B. At every iteration the kth distance vector is shared with all of the nodes

C. The vector is shared in two phases, sizes of n/2 and n-n/2, in async way, so that execution can continue running while communication is happening.

D. Openmp parallel for loops are used to parallelize execution in a single node. Memory alignment and chunksizes are selected to be 64B to prevent false sharing between omp threads.

```
#pragma omp parallel for schedule(static,ALIGNMENT) collapse(2)
```

E. If subgraphs are fully connected, Nonblocking Broadcast is used for communication.

F. Otherwise the kth vector is distributed to all nodes with increasing distance order.

Communication algorithm
Assume subgraph connections like this and A is sharing the kth vector with others,
A - B - C - D,, the communication parents are calculated beforehand, and each node receives the data from their comm parent. Also each comm parent shares the vector with their children.

A shares the data to B, B to C, C to D.

Another example:
A — C —- D
| |
B — E

A shares the data with B and C, B shares the data with E, C shares the data with D.

G. Used asynchronous one to one communication for gathering the data on the master node. Called Irecv before Isend via using a Barrier, which enabled writing the data directly to allocated memory instead of an inner buffer in MPI and later copy to allocated memory.

```
if(0 == myRank){
       for(int rank = 1; rank<size; rank++){
           MPI_Irecv(d + (starts[rank]), (ends[rank]-starts[rank]), MPI_UINT8_T, rank,
2*n, MPI_COMM_WORLD, &requests[rank]);
       }
   }
   //This way we won't do copy after Irecv as it will directly write it to d
   MPI_Barrier(MPI_COMM_WORLD);

   if(0 != myRank){
       MPI_Isend(local, (myEnd-myStart)*n, MPI_UINT8_T, 0, 2*n, MPI_COMM_WORLD,
&req1);
       MPI_Wait(&req1, MPI_STATUS_IGNORE);
   }
```

Also started to writing the output data while communications are still going on.

```
for (int rank = 1; rank<size; rank++){
           MPI_Wait(&requests[rank], MPI_STATUS_IGNORE);
           for (int i = starts[rank]/n; i < ends[rank]/n; ++i)
               for (int j = 0; j < n; ++j)
                   fprintf(outfile, "%d%s",
                       (i == j ? 0 : d[i * n + j]),
```

```
                        (j == n - 1 ? " \n" : " ")
                    );
```

System & Compiler Specs

CSE servers

gcc (GCC) 4.8.5 20150623 (Red Hat 4.8.5-44)
CPU(s): 20
Thread(s) per core: 2
Core(s) per socket: 10
Socket(s): 1
Model name: Intel(R) Core(TM) i9-10900X CPU @ 3.70GHz
L1d cache: 32K
L1i cache: 32K
L2 cache: 1024K
L3 cache: 19712K
DRAM: 32 GB

Correctness

$ bash runall.sh

mpirun -np 3 -hostfile hostfile ./apsp on ./dataset/50-1225:

diff cur.out ./dataset/50-1225.out:

mpirun -np 3 -hostfile hostfile ./apsp on ./dataset/100-4000:

diff cur.out ./dataset/100-4000.out:

mpirun -np 3 -hostfile hostfile ./apsp on ./dataset/500-40000:

diff cur.out ./dataset/500-40000.out:

mpirun -np 3 -hostfile hostfile ./apsp on ./dataset/1000-400000:

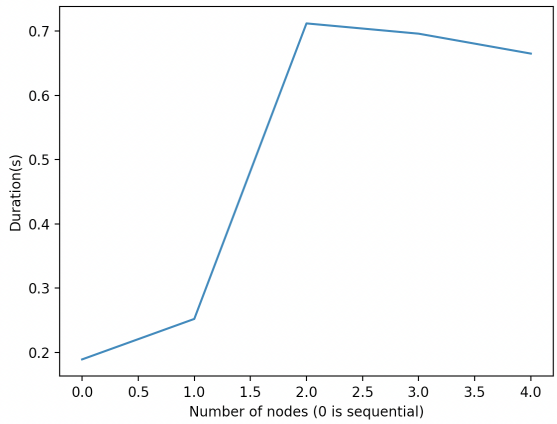
diff cur.out ./dataset/1000-400000.out:

mpirun -np 3 -hostfile hostfile ./apsp on ./dataset/2000-1200000:

diff cur.out ./dataset/2000-1200000.out:

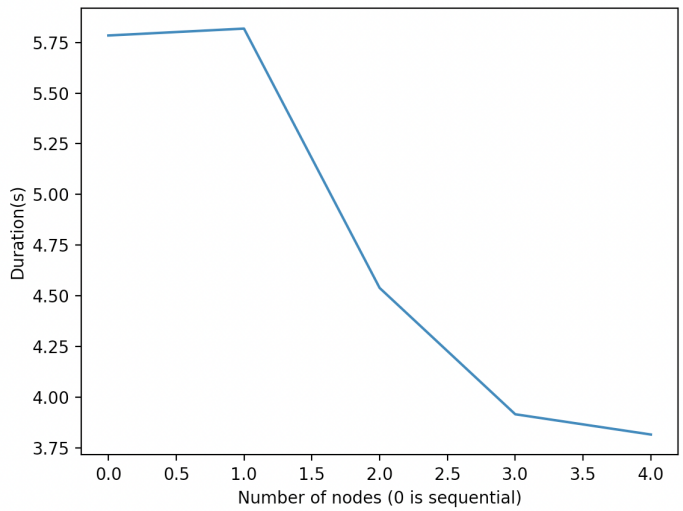
Scalability Plots

CSE Machines 500-40000.in data



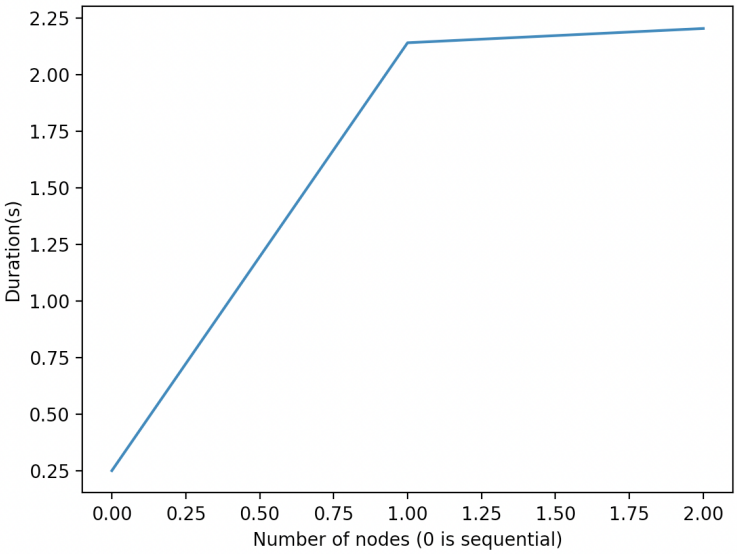
From what we see in the plot, increasing the number of nodes does not cause a speedup, it's the reverse, communication overhead is very significant and slows down the program. The application is not scalable beyond a single node with this data size. Also it seems OpenMP gets a significant speedup in a single node and it is not possible to increase performance with more nodes for this data size with this algorithm.
Warning:MPI_INIT has around 0.1 sec overhead on CSE machines which is important as well.

CSE Machines 2000-1200000.in data



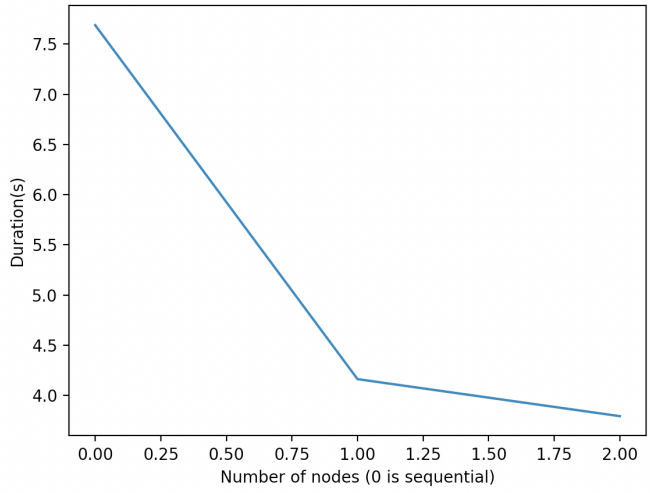
Comparing this and previous plots, we can see that as the data size gets bigger scalability increases. With this dataset, scaling beyond a single node makes sense and increases the performance. However, there is diminishing returns as we increase the number of nodes, and the speedup decreases a lot at 4 nodes, hence the application is not very scalable after 3 nodes with this dataset.

DevCloud Machines 500-40000.in data



Scaling from 1 node to 2 nodes does not increase the performance, it even decreases it. Hence the program is not that very scalable with this dataset size.
Warning: MPI_INIT has an overhead of 2 seconds on devcloud machines, we need to take this overhead into account as well.

DevCloud Machines 2000-1200000.in data



With this dataset, scaling beyond a single node does not make sense as it does not increase the performance that much. It seems OpenMP threads can cover most of the parallelism within a single node at this dataset size, hence the program is still not scalable at this dataset size. **Warning**: MPI_INIT has an overhead of 2 seconds on devcloud machines, we need to take this overhead into account as well.

Final Discussions

With scalability experiments and plots we checked the scalability of the algorithm on different machines and dataset sizes. From the looks of results, it seems the parallel floyd warshall algorithm is not very scalable when the dataset is not very big. It is probably due to the fact that all the work across the nodes is going synchronously. Hence it looks like most of the speedup is coming from OpenMP threads as they can work synchronously better than multiple nodes communicating through the network. At every iteration there is a one to all communication and all the nodes work on the same turn. We can guess that more isolated parallel algorithms could yield better results with multiple nodes, an example could be the parallel dijkstra algorithm.