Parallel Jaccard

Team Musilaj
Hasim Sait Goktan hasimsait@alumni.sabanciuniv.edu
Efe Oztaban efeoztaban@sabanciuniv.edu
Burakcan Kazim Yesil burakcan.yesil@boun.edu.tr

Single Node, 40 CPU Threads

The main improvement on the OMP code over the provided sample is to get rid of the linear search on the neighbors of the first node for each neighbour. At each iteration, start by marking the neighbours of the first node (convert the to naive representation) , then for all its neighbours, use that array as below. This provides ~20X speedup over the sample code.

```
for (int i = xadj[adj[v_ptr]]; i < xadj[adj[v_ptr] + 1]; i++) {
      // for every neighbour i of v
      if (u_neighbours[adj[i]])
        num_intersections++;
```

Single Node, 40 CPU Threads

For the next iteration, we need the u_union array to be memset to false, but memset on an array of n booleans is slow. The solution is marking the neighbours back to false. This provides a 12X speedup with 10 threads on com-dblp.

```
for (int u = 0; u < n; u++) {
  for (int v_ptr = xadj[u]; v_ptr < xadj[u + 1]; v_ptr++) {
    uv_union[adj[v_ptr]] = true;
    // set every neighbour of u to true.
  }
  /*THE CALCULATION*/
  for (int v_ptr = xadj[u]; v_ptr < xadj[u + 1]; v_ptr++) {
    // set every neighbour of u back to false for the next node.
    uv_union[adj[v_ptr]] = false;
  }
}
```

Single Node, 40 CPU Threads

The edges are symmetrical, we can trim some iterations by setting edge 2-1's value while calculating edge 1-2.

```
if (adj[v_ptr] > u) {
    // do not waste time with 2-1, 1-2 calculates that.
    for (int i = xadj[adj[v_ptr]]; i < xadj[adj[v_ptr] + 1]; i++) {
      // for every neighbour i of v
      if (uv_union[adj[i]])
        num_intersections++;
      if (adj[i] == u)
        // find v-u edge
        symetric_v_ptr = i;
    }
    jaccard_values[v_ptr] = num_intersections / card_u + card_v - num_intersections;
    jaccard_values[symetric_v_ptr] = jaccard_values[v_ptr];
}
```

Single Node, 40 CPU Threads

Parallelization is very easy with OpenMP, the only difference is #pragma omp for.

```
#pragma omp parallel
  {
    // instead of unordered set, keep an array of size n
    bool *uv_union = new bool[n];
    memset(uv_union, false, n * sizeof(bool));
#pragma omp for schedule(dynamic)
    for (int u = 0; u < n; u++) {
      /*The calculation for the edges of u.*/
    }
}
```

CPU Results on Truba, Single Node

graph	OMP t=10	OMP t=20	OMP t=30	OMP t=40
com-dblp	0.02582	0.03861	0.05826	0.04880
youtube	0.10182	0.08554	0.11841	0.10434

Multiple GPU Single Node

Somehow there is an invalid memory access when I split the work for multiple gpus. Couldn't run cuda-memcheck properly on truba and my machine has a single gpu. It is single gpu, 1 node per thread instead of 1 edge per thread. Performance is bad, it is correct.

Multiple GPU Kernel

```
int u = blockDim.x * blockIdx.x + threadIdx.x + chunk_start;
if (u < *nov) {
  for (int v_ptr = xadj[u]; v_ptr < xadj[u + 1]; v_ptr++) {
    int v = adj[v_ptr]; // v is a neighbor of u
    int num_intersections = 0;

    for (int u_nbr_ptr = xadj[u]; u_nbr_ptr < xadj[u + 1]; u_nbr_ptr++) {
      // Go over all neighbors of u
      int u_nbr = adj[u_nbr_ptr];
      for (int v_nbr_ptr = xadj[v]; v_nbr_ptr < xadj[v + 1]; v_nbr_ptr++) {
        // Go over all neighbors of v
        int v_nbr = adj[v_nbr_ptr];
        if (u_nbr == v_nbr) {
          // Neighbors of u and v match. Increment the intersections
          num_intersections++;
        }
      }
    }
    int num_union = xadj[u+1]-xadj[u] + xadj[v+1] - xadj[v] - num_intersections;
    // ||X U Y|| = ||X|| + ||Y|| - ||X ∩ Y||
    jaccard_values[v_ptr] = float(num_intersections) / float(num_union);
  }
}
```