SOMETHING WITH CONNECTED COMPONENTS

Michael Bernasconi, Roman Haag, Giovanni Balduzzi, Lea Fritschi

Department of Computer Science
ETH Zürich
Zürich, Switzerland

ABSTRACT

1. INTRODUCTION

Motivation.

Related work.

Connected components.

For an unordered graph $G = (V, E)$, the connected components are the ensemble of connected subgraphs, where connected means that for any two vertices, there exist a path along the edges connecting them. The straightforward algorithm to find them is to perform either a breath or depth first search from a starting random vertex in V, and give the same label to all the touched vertices. Then repeat the search from an unlabeled vertex until there are no more left. This has a cost in terms of memory accesses of $\Theta(|E|+|V|)$, which turns to be optimal [1].

2. PROPOSED ALGORITHM

Unfortunately this algorithm does not parallelize straightforwardly. Instead we firstly implemented an algorithm proposed by Uzi Vishkin [**?**] and later described in a class by Pavel Tvrdik [2]. This algorithm casts the problem in terms of the generation of a forest, where the vertices of the same connected component belong to the same three, and its root can be used as the representative. We define a star as a three of height one, a singleton a tree with a single element, and use the variables $n = |V|$ and $m = |E|$. His algorithm can be summarized as:

Algorithm 1 Pavel Tvrdik's Connected components

```
1:  procedure HOOK(i, j)
2:      p[p[i]] = p[j]
3:  end procedure
4:  procedure CONNECTEDCOMPONENTS(n, edges)
5:      p[i] = i   ∀i ∈ {1, ··· , n}.        ▷ Initialize a list of parents.
6:      while Elements of p are changed. do
7:          for ⟨i, j⟩ ∈ edges do            ▷ Execute in parallel.
8:              if i ≥ j then Hook(i, j)
9:              if isSingleton(i) then Hook(i, j)
10:         end for
11:         for ⟨i, j⟩ ∈ edges do            ▷ Execute in parallel.
12:             if isStar(i) and i ≠ j then Hook(i, j)
13:         end for
14:         p[i] = root(i)   ∀i ∈ {1, ··· , n}   ▷ Compress the forest in parallel.
15:     end while
16: end procedure
```

We defer to [2] for a proof of correctness.

After implementing this algorithm we found advantageous to remove the constraint that only singletons and stars can be hooked to another vertex, so that only a single pass through the edge list is required. Extra care is then required during parallel execution: as each vertex has only one outgoing connection, we need to avoid that a process overwrites a connection that has been formed by another one. We therefore need to grow our forest with the following rules:

1. A hook must originate from a vertex id higher than the destination.

2. All edges must generate a connection between the relative vertices, or vertices at an higher level in their three.

3. A hook must originate from a vertex that is currently the root of a tree.

The intuitive proof of correctness follows: rule 1 means that the graph generated by the hooks generate a directed graph with no cycles and with at most a single outgoing connection, therefore it must be a forest. Rule 2 and 3 enforce that after processing an edge between two nodes, they belong to the same tree, and rule 3 guarantees that this connection can not be broken by a different edge. At the end of the algorithm, by following the connections from each vertex to the root, we can find a representative for each connected component.

To implement rule 3 in a multi-threaded environment, we use an atomic compare and swap. We compare the parent of the hook's origin with its id, if they match it means the vertex is still a root and we hook we hook it to its destination. It does not matter for correctness if the destination is a root, but we try without enforcing to hook to a root to minimize the three height. We found empirically that using `std::atomic_compare_exchange_weak`, compared to `std::atomic_compare_exchange_strong` offers better performance, as we anyway need to loop until a hook is successful.

In pseudocode our algorithm is:

Algorithm 2 Single pass connected component.

```
1:  procedure CONNECTEDCOMPONENTS(n, edges)
2:      p[i] = i   ∀i ∈ {1, ··· , n}.
3:      for ⟨i, j⟩ ∈ edges do          ▷ Execute in parallel.
4:          while hook is not successful. do
5:              from = max(root(i), root(j))
6:              to = mint(root(i), root(j))
7:              atomicHook(from, to)
8:          end while
9:          if !isRoot(i) then p[i] = root(i)
10:         if !isRoot(j) then p[j] = root(j)
11:     end for
12:     p[i] = root(i)   ∀i ∈ {1, ··· , n}    ▷ Compress the
    forest in parallel.
13: end procedure
```

While step 9 is not necessary for correctness, we found that reusing the already computed vertex's representative leads to a smaller tree height. This and the parallel compression works and was tested to be efficient only on architectures such as x86, where writes to 32 or 64-bits , used to store a vertex's id, are atomic.

We tried implementing the parallel execution of loop 3 with Boost fibers [3] whose execution is scheduled with a work stealing algorithm, and OpenMP with a dynamic scheduler. OpenMP performed better by a large margin and will be used to acquire the data presented later.

The overall cost of the algorithm is $\Theta((n+m)\langle H\rangle)$, where $\langle H\rangle$ is the average tree height. Therefore $\langle H\rangle = \Theta(1)$ for a sub-critical random graph, and on average (relatively to the execution order of the loop) $\langle H\rangle = \Theta(\log(n))$ for a supercritical one [4].

Multiple compute nodes. Algorithm works only on a single compute node with a shared memory model. Moreover it is efficient only when the graph is relatively sparse so that the chance of a collision between two processors trying to update the same parent is low.

We propose to extend our algorithm by distributing the list of edges evenly among each MPI process, then each one of them computes a forest used only the subset of edges it received. This local computation is followed by a reduction step, where the list of representatives is sent to another process, which confront it with its own. If a discrepancy is detected, a hook is inserted between the two different parents, then the resulting forest is compressed again before the following reduction step.

Using p processes, the total execution time of this extension scales as $\Theta(\frac{(m+n)}{p}\langle H\rangle + n\log p)$.

On top of allowing to scale past a single compute node, this approach is advantageous on dense graphs: if the reduction cost is negligible, the scaling is the same as algorithm 2 executed on a single node, but we can avoid the cost of performing atomic hooks, if a single thread is used, or limit the number of failures if a few threads are used. Therefore a different mixture of MPI ranks and OpenMP threads per rank is advised depending on the density of the graph.

Distributed vertices. While the described approach works on generic graphs, it performs poorly on very sparse graphs using a large number of compute nodes. Moreover the full set of vertices' id must fit in memory, limiting the graph size to 8 billions vertices. If the connectivity of a graph the size of a human brain needs to be studied, we propose do distribute the representation of the vertices as well.

Often, real word graphs are embedded on a space with some metric, and connections are present much more frequently between vertices that are close together. For examples the pixel representing features of a picture, or the roads connecting cities with a known geographical position, posses this property.

We represent this type of graphs with a very simple model: a two dimensional lattice with random connections between nearest neighbors only. We split the lattice in as many square tiles as there are processes. Then each process applies algorithm 2 with the subset of edges connecting two vertices in their own tile. Finally we process the boundary edges, connecting vertices of different tiles, with MPI one sided communication. The list of ids of the local vertices is stored in an MPI window, so that the representative of a remote vertex can be obtained with `MPI_Get`, while an hook can be created with `MPI_Compare_and_swap`. Therefore only two global synchronization points are necessary: after all edges have been processed, and after the final compression of the forest.

Unfortunately, due to time constraint in developing, in our implementation each MPI operation is synchronized locally. This leads to good scaling results only on extremely sparse graphs. Future work should consider batching several MPI requests before synchronization is required.

3. EXPERIMENTAL RESULTS

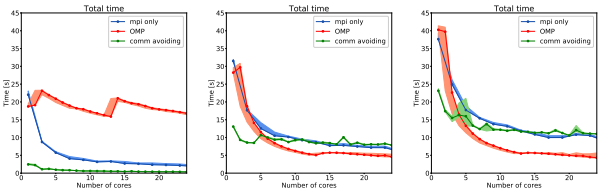
To evaluate our algorithms performance a number of experiments were run on both the Euler cluster and the Piz Daint cluster. In the following paragraphs we will first describe both the Euler and the Piz Daint setups. We will then go on to discussing each experiment.

Euler setup. Each node in the Euler V cluster contains two 12-core Intel Xeon Gold 5118 processors and 96 GB of DDR4 memory clocked at 2400 MHz [5]. We were allowed to use up to two nodes giving us a maximum of 48 cores.

Piz Daint setup. [Insert Daint specs]

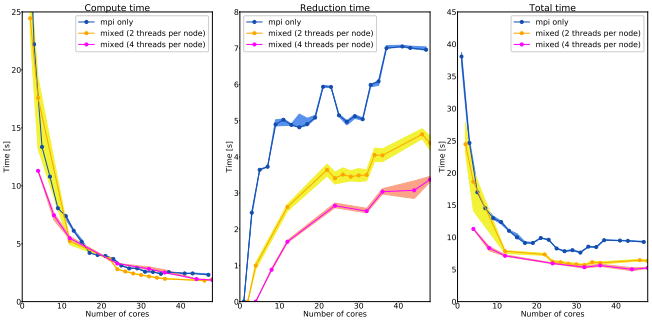
Graph Generation. To evaluate our algorithm we generated undirected, unweighted graphs using [6]. Multiple edges connecting the same two vertices as well as self loops were not allowed.

MPI vs OMP vs Communication Avoiding. The results in Figure 1 show our algorithm compared against the communication avoiding algorithm on three different graphs with the same number of edges but different densities. Our algorithm was run in both the MPI and the OMP only mode. Figure 1a shows the MPI only version outperforming the OMP version by a large margin on the densest graph. This can be explained by the combination of two effects. The first being the fact that a dense graph results in more contention between the different OMP threads during the edge contractions. The second being the fact that the reduction after the contractions scales linearly with the number of vertices. Since the number of vertices is comparatively low in a dense graph the reduction is fast. We further observe a significant increase in the total compute time from one to two cores and from 12 to 13 cores for the OMP only version. The first jump can be explained by the initial overhead of doing the computation in parallel compared to the serial version on one core. Since a single CPU on Euler has 12 cores the second jump is a result of the cache coherency protocol being slow across multiple CPUs.
The results in Figure 1b and Figure 1c were obtained from a much sparser graph compared to Figure 1a. Here, for a large number of cores, the OMP only version is clearly faster than the MPI only version. As can be seen this is the result of the OMP version being faster and the MPI only version being slower compared to the dense graph in Figure 1a. The MPI only version being slower can be explained by the fact that the reduction step increases linearly with the number of vertices. The OMP only version being faster can be explained by the reduced contention between the different



(a) 10^5 vertices (b) 10^7 vertices (c) $2 \cdot 10^7$ vertices

Fig. 1: Total runtime of three different graphs each with 10^5 edges. The experiment was run on the Euler cluster.



(a) $2 \cdot 10^7$ vertices

(b) Total runtime breakdown on graph with 10^5 edges and $2 \cdot 10^7$ vertices. The experiment was run on the Euler cluster.

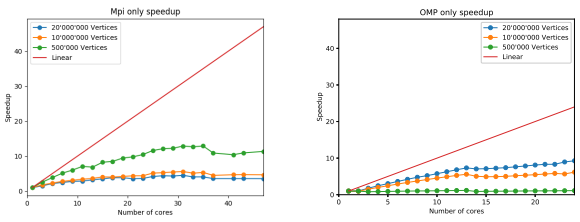
OMP threads due to the sparser graph.
The results in Figure 1 show the communication avoiding algorithm scaling badly with the number of cores. This is expected since the edge contractions are computed on a single node in this algorithm. Since our algorithm does to scale with the number of cores up to some point we manage to outperform the communication avoiding algorithm on each graph.

Mixing MPI and OMP. As seen in Figure 1 there is a distinct difference between the MPI only and the OMP only version of our algorithm. To further investigate this behaviour the algorithm was tested using a mixture of both MPI and OMP.

Figure 2b shows the compute time being largely independent of the mixture. This is a consequence of the graphs sparsity which results in low contention between the OMP threads. The reduction time, however, wildly differs for the different mixtures, each increasing logarithmically with the respective number of MPI ranks. This results in the algorithms performance decreasing as less OMP threads per MPI rank are used.

Speedups.

Figure 3 shows the measured speedups of both the MPI only and OMP only version. As one would expect from the results discussed previously the MPI only version does achieve better scaling compared to the OMP only version



(a) MPI only (b) OMP only

Fig. 3: Speedup of MPI only and OMP only version on three different graphs each with 10^5 edges.

on dense graphs while the OMP only version scales better on sparse graphs.

Results.

4. CONCLUSIONS

[insert conclusion]

5. FURTHER COMMENTS

The main drawback of our algorithm is the reduction steps runtime of $O(n \cdot log(p))$, where p is the number of MPI ranks and n is the number of vertices. For a large number of MPI ranks the $log(p)$ factor becomes a problem.
One approach to solve this problem is trying to reduce the reductions "height" of $log(p)$. In our algorithm this is done by using a mixture of OMP and MPI which reduces the number of MPI ranks. Since OMP does not scale well once the number of OMP threads exceeds the number of cores per CPU this approach is only viable up to a limited number of cores.
Another approach would be reducing the work n done in each reduction step. Due to time constraints we were unable to investigate this approach. One could imagine a more efficient hook tree representation solving this problem.

Another drawback of our algorithm is that in order to achieve satisfying performance one needs to find the right mixture of MPI and OMP. While we analysed the behaviour of different mixtures on graphs with varying density we did not come up with an a priori scheme to determine the right mixture. A good heuristic or even a scheme to find the optimal mixture would be worth exploring.

6. REFERENCES

[1] John Hopcroft and Robert Tarjan, "Algorithm 447: Efficient algorithms for graph manipulation," *Commun. ACM*, vol. 16, no. 6, pp. 372–378, June 1973.

[2] Pavel Tvrdik, "Topics in parallel computing," Lecture, May 1999.

[3] Oliver Kowalke, "Boost fibers library," 2013.

[4] P. Erdös and A. Rényi, "On random graphs i," *Publicationes Mathematicae Debrecen*, vol. 6, pp. 290, 1959.

[5] "Euler cluster," .

[6] Farzad Khorasani, Rajiv Gupta, and Laxmi N. Bhuyan, "Scalable simd-efficient graph processing on gpus," in *Proceedings of the 24th International Conference on Parallel Architectures and Compilation Techniques*, 2015, PACT '15, pp. 39–50.