SOMETHING WITH CONNECTED COMPONENTS

Giovanni Balduzzi, Michael Bernasconi, Lea Fritschi, Roman Haag

Department of Computer Science
ETH Zürich
Zürich, Switzerland

ABSTRACT

We present an improved parallel implementation of a tree based connected components algorithm originally introduced by U. Vishkin[~~?~~]. To improve the algorithm's performance the graph's edges are distributed across multiple cores. Each core~~process~~ computes in a lock-free manner a local forest representing the different connected components. The resulting forests are then combined using a reduction. A comparison of our algorithm against a communication avoiding algorithm proposed by Lukas Gianinazzi et al.[~~?~~] on a set of large graphs (5×10^8 Edges with up to 2×10^7 Vertices) is presented. Results where our algorithm uses a mixture of both MPI and OMP are also shown.

1. INTRODUCTION

Motivation. Problems in computer science are often modeled as graphs. Therefore, graph algorithms are ubiquitous. One of these graph problems is finding connected components. It is a well understood problem in graph theory with a variety of applicable domains. Computer vision tasks, such as pattern recognition and image segmentation [**?**] can make use of connected components [**?**]. Other fields are medical imaging [**?**] and image processing [**?**]. The related problem of strongly connected components will not be discussed in this paper.

Related Work. The first sequential algorithm to solve the connected components problem goes back to [**?**]. Parallel approaches were presented in [**?**] and [**?**]. Recently [**?**] was published, where a communication-avoiding approach was discussed. A communication-avoiding algorithm uses asymptotically less communication. By doing so [**?**] sacrifices some computational efficiency as the root node does most of the work. In this paper we present an algorithm which distributes the work while still avoiding as much communication as possible. This is achieved by distributing the list of edges evenly among different MPI ranks. These then locally compute their corresponding connected components which are represented as a forest. In a next step the algorithm reduces these forests in a binary manner. Two MPI ranks compare and merge their results and then compress them. This step is repeated until the final result is propagated to the root process.

Connected components.

For an undirected graph $G = (V, E)$, the connected components are the ensemble of connected subgraphs, where connected means that for any two vertices there exists a path along the edges connecting them. A straightforward algorithm to find the connected components is to perform either a breath or depth first search starting from a random vertex in V, and give the same label to all vertices reached. The search is then repeated, starting from an unlabeled vertex. This has a cost in terms of memory accesses of $\Theta(|E|+|V|)$, which turns out to be optimal [**?**].

2. PROPOSED ALGORITHM

Unfortunately this algorithm does not parallelize in a straightforward way. Instead we first implemented an algorithm proposed by Uzi Vishkin [**?**] and later described in a class by Pavel Tvrdik [**?**]. 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 tree, and its root can be used as the representative.
We define a star as a tree of height one, a singleton as a tree with a single element, and use the variables $n = |V|$ and $m = |E|$. Thealgorithm 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: $p[i] = i \quad \forall i \in \{1, \cdots, n\}$. ▷ Initialize a list of parents.
6: **while** Elements of p are changed. **do**
7: **for** $\langle i, j \rangle \in$ edges **do** ▷ Execute in parallel.
8: **if** $i \geq j$ **then** Hook(i, j)
9: **if** isSingleton(i) **then** Hook(i, j)
10: **end for**
11: **for** $\langle i, j \rangle \in$ edges **do** ▷ Execute in parallel.
12: **if** isStar(i) **and** $i \neq j$ **then** Hook(i, j)
13: **end for**
14: $p[i] =$ root(i) $\quad \forall i \in \{1, \cdots, n\}$ ▷ Compress the forest in parallel.
15: **end while**
16: **end procedure**

We defer to [**?**] 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 only has~~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 larger than the destination.
2. All edges must generate a connection between the relative vertices, or vertices at an higher level in their tree.
3. A hook must originate from a vertex that is currently the root of a tree.

An intuitive proof of correctness follows: rule 1 means that the graph generated by the hooks is a directed graph without cycles and at most with a single outgoing connection. Therefore, it must be a forest. Rules 2 and 3 enforce that after processing an edge between two nodes, they belong to the same tree. Rule 3 guarantees that this connection cannot 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 it to its destination. For correctness it does not matter if the destination is a root, but doing so minimizes 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: $p[i] = i \quad \forall i \in \{1, \cdots, n\}$.
3: **for** $\langle i, j \rangle \in$ 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) $\quad \forall i \in \{1, \cdots, n\}$ ▷ 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 [**?**] 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 [**?**].

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 par-

ents, 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 algorithm's performance a number of experiments were run on both the Euler 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 [**?**]. We were allowed to use up to two nodes, giving us a maximum of 48 cores.

Piz Daint setup. Each of the utilized XC40 nodes on Piz Daint contains two Intel Xeon E5-2695, each with 18 hardware threads, and

Graph Generation. Our algorithm was evaluated on undirected, unweighted graphs. Multiple edges connecting the same two vertices and self loops were not allowed. All graphs were generated using [**?**].

MPI vs OMP vs Communication Avoiding. The results in Figure 1 show our algorithm compared with the communication avoiding algorithm [**?**] on three different graphs with the same number of edges but different densities. Our algorithm was run in MPI and OMP only mode. Figure **??** shows the MPI only version outperforming the OMP version on the densest graph. This can be explained by the combination of two effects: the first being that a dense graph results in more contention between the OMP threads during the edge contractions, and the second being 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.
The results in Figure **??** and Figure **??** were obtained using sparser graphs compared to Figure **??**. Here, for a large number of cores, the OMP only version is clearly faster than the MPI only version. Figure 1 shows a trend of the OMP only version speeding up as the graph becomes sparser, while the MPI only version slows down. The OMP only version's speed up can be explained by the reduced contention between the OMP threads due to the sparser graph. The MPI only version's slow down is a result of the increased reduction time due to the increasing number of vertices.
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. To further investigate the distinct difference between the MPI only and the OMP only version the algorithm was tested using a mixture of 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. For each mixture it increases logarithmically with the number of MPI ranks. This results in the algorithm's performance decreasing as less OMP threads per MPI rank are used.

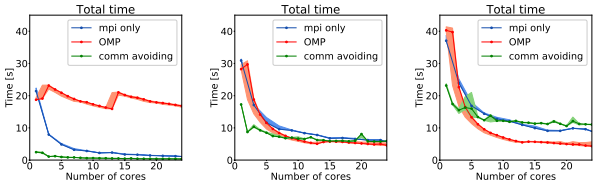
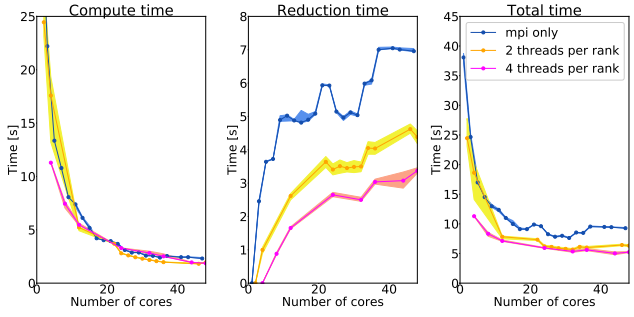
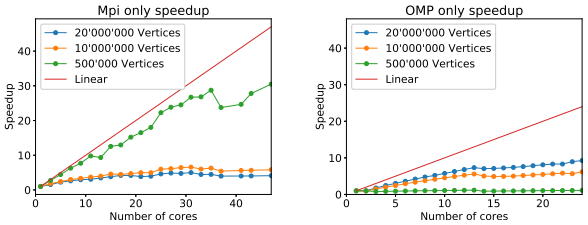


Fig. 1: Comparison of the total runtime of our algorithm with the communication avoiding algorithm [**?**] over three different graphs each with 5×10^8 edges and $5 \times 10^5, 1 \times 10^7, 2 \times 10^7$ vertices. The experiment was run on the Euler cluster.



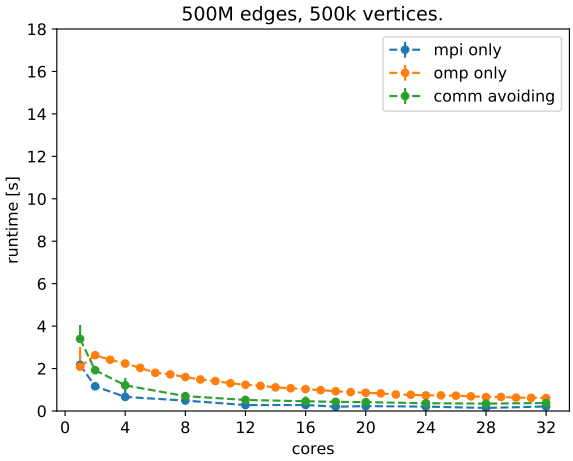
(a) 2×10^7 vertices

(b) Total runtime breakdown on graph with 5×10^8 edges and 2×10^7 vertices. The experiment was run on the Euler cluster.



(a) MPI only (b) OMP only

Fig. 3: Speedup of MPI only and OMP only version on three different graphs each with 5×10^8 edges.



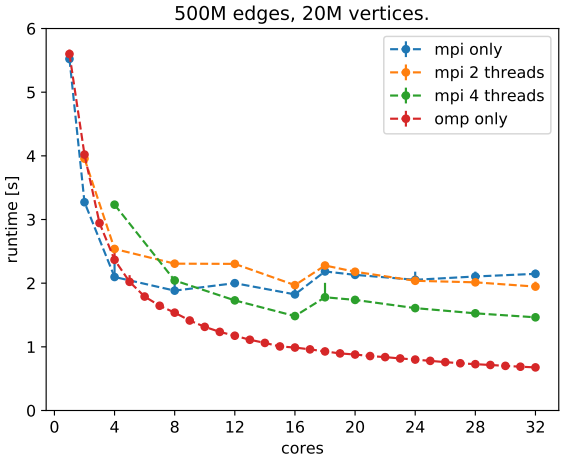
Speedups.

Figure 3 shows the measured speed up of the MPI only and OMP only version. As one would expect from the results discussed previously the MPI only version achieves better scaling compared to the OMP only version on dense graphs while the OMP only version scales better on sparse graphs.

Results on Piz Daint. In addition to testing our algorithm on the Euler cluster it was also tested on the Piz Daint cluster. The graphs used were the same as in the experiments run on Euler. We will now discuss interesting differences in the results.

The OMP only version's behaviour, shown in **??**, is very different to its behaviour on Euler. We do still observe a decrease in performance when going from one to two cores. The decrease in performance, however, is not as severe as on Euler. The performance decrease as we go from one to multiple CPUs is no longer present. This can be explained by Daint's cache coherency protocol being more efficient, especially across multiple CPUs.

Results.



4. CONCLUSIONS

The proposed algorithm manages to take advantage of the parallelism available within shared memory units while avoiding excessive communication between them. By choosing the right number of OMP threads per MPI rank the algorithm achieves good scaling across a variety of graphs. While on dense graphs the Communication Avoiding algorithm [**?**] yields better results, it is outperformed by our algorithm on sparser graphs.

5. FUTURE WORK

The main drawback of our algorithm is the reduction step's 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 an issue.
Reducing the reductions "height" addresses this problem. 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.