PARALLEL 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 in 1984 by U. Vishkin. To improve the algorithm's performance the graph's edges are distributed across multiple cores. Each 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 recently developed at ETH, 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 [3] can make use of connected components [4]. Other fields are medical imaging [5] and image processing [6]. 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 [7]. Parallel approaches were presented in [8] and [9]. Recently a communication-avoiding algorithm was published [2]. This algorithm uses asymptotically less communication, but 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 [7].

2. PROPOSED ALGORITHM

Unfortunately this algorithm does not parallelize in a straightforward way. Instead we first implemented an algorithm proposed by Uzi Vishkin [1] and later described in a class by Pavel Tvrdik [10]. 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 $m = |E|$. The algorithm can be summarized as:

Algorithm 1 Pavel Tvrdik's Connected components

```
procedure Hook(i, j)
    p[p[i]] = p[j]
end procedure
procedure connectedComponents(n, edges)
    p[i] = i   ∀i ∈ {1, ··· , n}.        ▷ Initialize a list of parents.
    while Elements of p are changed. do
        for ⟨i, j⟩ ∈ edges do        ▷ Execute in parallel.
            if i ≥ j then Hook(i, j)
            if isSingleton(i) then Hook(i, j)
        end for
        for ⟨i, j⟩ ∈ edges do        ▷ Execute in parallel.
            if isStar(i)  and i ≠ j then Hook(i, j)
        end for
        p[i] = root(i)   ∀i ∈ {1, ··· , n}      ▷ Compress the forest in parallel.
    end while
end procedure
```

We defer to [10] 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. This allows the algorithm to terminate after a single pass through the edge list. Extra care is then required during parallel execution: as each vertex only has one outgoing connection, we need to avoid the scenario where a thread overwrites a connection that has been formed by another one. Therefore whenever we need to generate a hook from $v1$ to $v2$ we choose the following rules:

1. $\text{index}(v1) > \text{index}(v2)$.
2. $v1$ must be a root.

An intuitive proof of correctness follows: rule 1 means that the graph generated by the hooks is a directed graph without cycles and each vertex has at most one outgoing connection. Therefore, it must be a forest. Rule 2 guarantees that a connection cannot be broken by a different edge. After the algorithm terminates all vertices in a tree belong to the same connected component. The connected component can be canonically represented by the tree's root.

To implement rule 2 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 a root, and this status was not modified by another thread. For correctness it does not matter if the destination is a root, but doing so minimizes the tree height. Empirically we found that, using the standard library, weak atomics offer better performance compared to strong atomics.

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. We avoided using atomic directives for this step and the parallel compression 12 therefore our implementations works only on architectures such as x86, where writes to 32 or 64-bits variables, used to store a vertex's id, are atomic.

We tried implementing the parallel execution of loop 3 with Boost fibers [11] whose execution is scheduled with a work stealing algorithm, and OpenMP with a dynamic scheduler. OpenMP performed better by a large margin and thus was used to acquire the data presented in Section 3.

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 (over the execution order of the loop) $\langle H\rangle = \Theta(\log(n))$ for a supercritical random graph [12].

Multiple compute nodes. The algorithm works only on a single compute node with a shared memory model. Moreover, it is efficient only when the graph is sparse enough that the probability 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 the MPI processes. Then each one of them computes a forest using only the subset of edges it received. This local computation is followed by a reduction step, where pairs of MPI ranks combine their respective forests. The resulting forest is then compressed 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 offers better results on dense graphs, where the relative cost of the reduction is small, and multiple threads would have a high probability of collision during the atomic update. Therefore a different mixture of MPI ranks and

OpenMP threads per rank is advised depending on the density of the graph.

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 separately. The errorbars in all plots refer to a 95% confidence interval.

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 [13]. We were allowed to use up to two nodes, giving us a maximum of 48 cores. The algorithm was compiled and run using gcc 6.2.3 and Open MPI 3.0.0.

Piz Daint setup. Each of the utilized XC40 nodes on Piz Daint contains two Intel Xeon E5-2695, each with 18 hardware threads, and 32 GB of DDR4 memory. We used both gcc and the Cray compiler, and observed marginally faster run times with the latter.

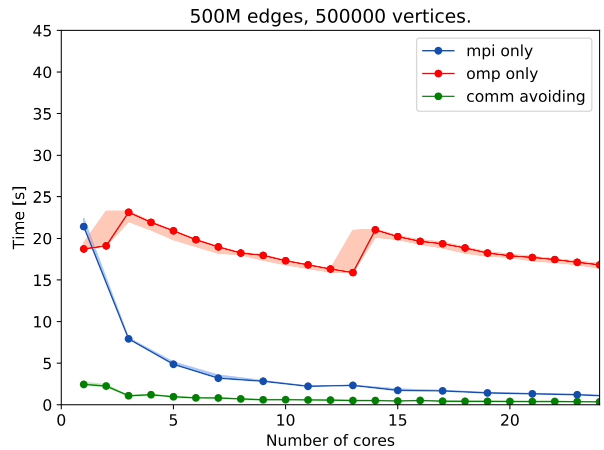
Graph Generation. Our algorithm was evaluated on undirected, unweighted graphs. Multiple edges connecting the same two vertices and self loops were not allowed. All graphs were generated using [14]. The same tool was used in [2]

MPI vs OMP vs Communication Avoiding.

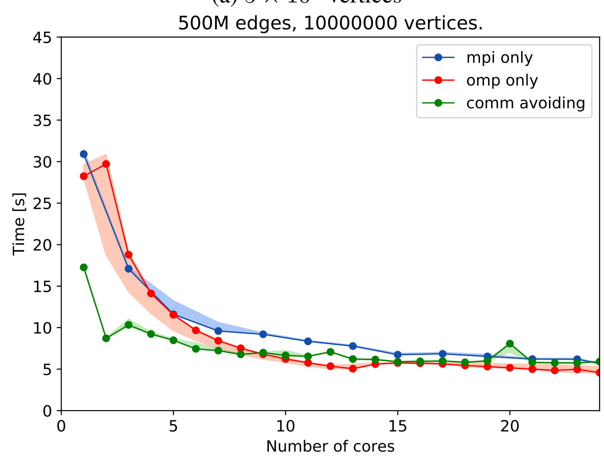
The results in Figure 1 show our algorithm compared with the communication avoiding algorithm [2] on three different graphs with the same number of edges but different densities. Our algorithm was run in MPI and OMP only mode.

Figure 1a shows the MPI version outperforming the OMP version on the densest graph. This can be explained by the combination of two effects. The first is that a dense graph results in more contention between the OMP threads during the edge contractions. The second is that the reduction scales linearly with the number of vertices. Since the number of vertices is comparatively low in a dense graph the reduction is fast. For the OMP version we further observe a significant increase in total compute time from one to two cores and from 12 to 13 cores. The first jump can be explained by the initial overhead of doing the computation in parallel. 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.

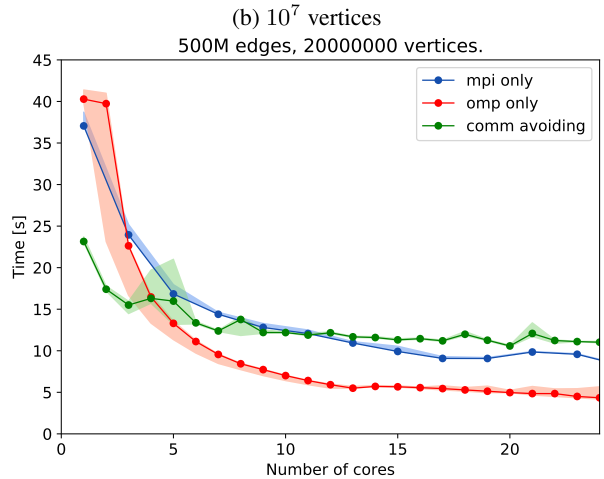
The results in Figure 1b and Figure 1c were obtained using sparser graphs compared to Figure 1a. Here, for a large number of cores, the OMP version is clearly faster than the MPI version. Figure 1 shows a trend of the OMP version speeding up as the graph becomes sparser, while the MPI version slows down. The OMP version's speed up can be



(a) 5×10^5 vertices



(b) 10^7 vertices



(c) 2×10^7 vertices

Fig. 1: Comparison of the total runtime of our algorithm with the communication avoiding algorithm [2] 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.

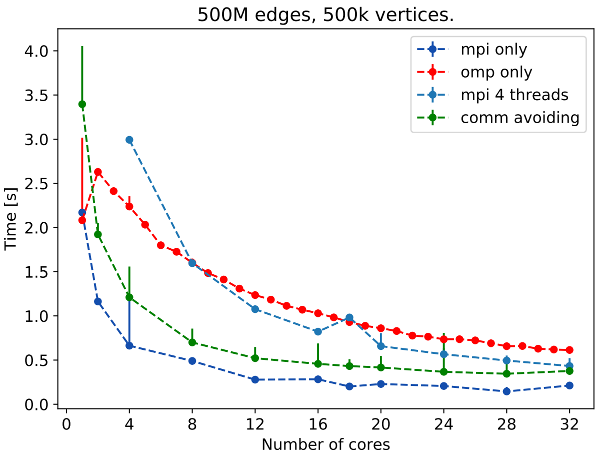


Fig. 2: Piz Daint results on graph with 5×10^8 edges and 5×10^5 vertices.

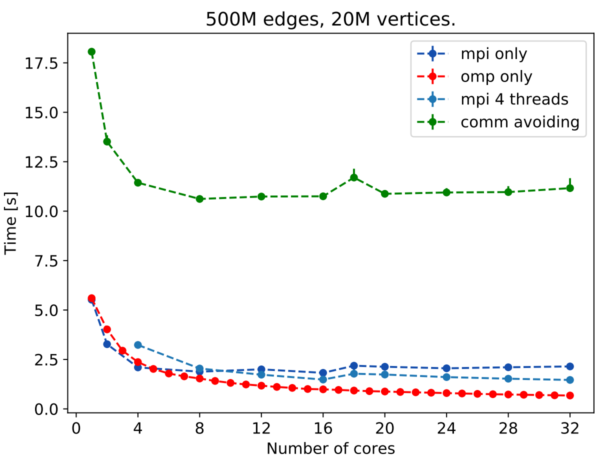
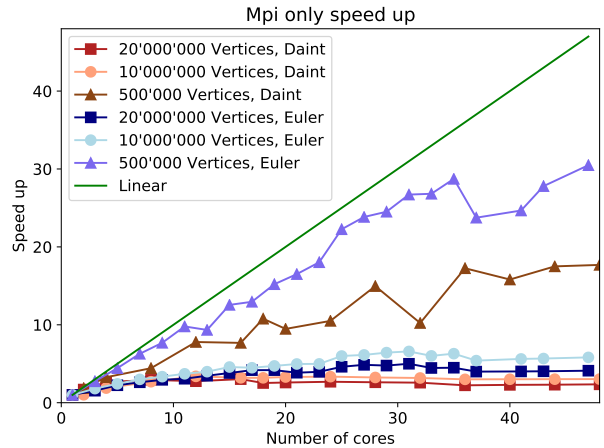


Fig. 3: Piz Daint results on graph with 5×10^8 edges and 2×10^7 vertices.

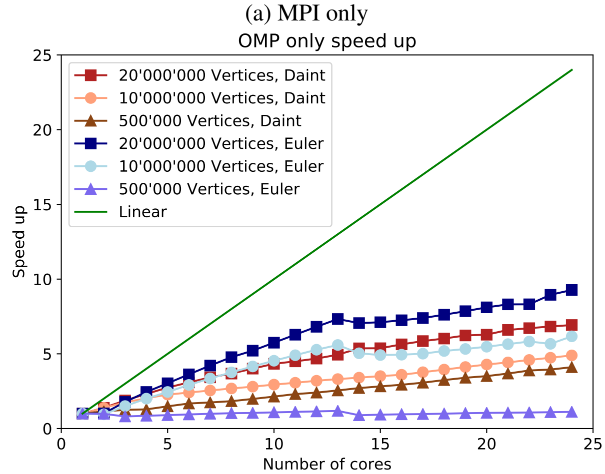
explained by less contention between the OMP threads due to the sparser graph. The MPI version's slow down is the result of the increased reduction time due to the larger number of vertices.
The results in Figure 1 show the communication avoiding algorithm scaling badly with the number of cores. This is expected since the edge contractions are computed on a single node. Since our algorithm does scale with the number of cores up to some point we manage to outperform the communication avoiding algorithm on each graph.

Results on Piz Daint.

Additionally we tested our algorithm 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.
Figure 2 shows the results for the densest graph on Piz Daint. The OMP version behaves differently compared to Euler.



(a) MPI only



(b) OMP only

Fig. 4: Speedup of MPI only and OMP only version on three different graphs each with 5×10^8 edges.

We still observe a decrease in performance when going from one to two cores. The decrease 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. Figure 3 shows the Piz Daint results for different mixtures of MPI and OMP on the sparsest graph. As on Euler, we do see the OMP version performing best for a large number of cores. For all versions, except the OMP version, we can clearly see the effect the number of reduction steps has on the runtime. It is most noticeable as we go from 16 to 18 cores. Here each version has to perform an additional reduction step which results in the increased runtime.

Speedups.

Figure 4 shows the measured speed up of the MPI and OMP version. As one would expect from the results discussed previously the MPI version achieves better scaling

on dense graphs while the OMP version scales better on sparse graphs.
We can also see that our algorithm achieves better scaling on Euler for the MPI version. The reason our algorithm scales better on Euler is that the single core performance is lower on Euler compared to Daint. This means that the edge contractions dominate the runtime for longer before the reduction step starts to be an issue.
For the OMP version the discrepancy in the speed ups can be explained by the differences in the shared memory architecture.

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. The results show our algorithm outperforming the communication-avoiding algorithm [2] in each experiment except for the densest graph on Euler.

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.

Limiting the reduction "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.

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.

~~Finally we expect that large improvements can be made in the distributed vertices algorithm described in section ?? by batching several MPI one sided reads before synchronization. Further data should be gathered to validate the algorithm with different components sizes, while comparing different MPI implementation such as foMPI [?].~~

6. REFERENCES

[1] Uzi Vishkin, "An optimal parallel connectivity algorithm," *Discrete Applied Mathematics*, vol. 9, no. 2, pp. 197 – 207, 1984.

[2] Lukas Gianinazzi, Pavel Kalvoda, Alessandro De Palma, Maciej Besta, and Torsten Hoefler, "Communication-Avoiding Parallel Minimum Cuts and Connected Components," 02 2018, Accepted at The ACM Conference Principles and Practice of Parallel Programming 2018 (PPoPP'18).

[3] Jia-Ping Wang, "Stochastic relaxation on partitions with connected components and its application to image segmentation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, no. 6, pp. 619–636, June 1998.

[4] Andrew D. Wilson, "Robust computer vision-based detection of pinching for one and two-handed gesture input," in *Proceedings of the 19th Annual ACM Symposium on User Interface Software and Technology*, New York, NY, USA, 2006, UIST '06, pp. 255–258, ACM.

[5] Jayaram K Udupa and Venkatramana G Ajjanagadde, "Boundary and object labelling in three-dimensional images," *Computer Vision, Graphics, and Image Processing*, vol. 51, no. 3, pp. 355 – 369, 1990.

[6] Luigi Ambrosio, Vicent Caselles, Simon Masnou, and Jean-Michel Morel, "Connected components of sets of finite perimeter and applications to image processing," *Journal of the European Mathematical Society*, vol. 3, no. 1, pp. 39–92, Feb 2001.

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

[8] M Manohar and H.K Ramapriyan, "Connected component labeling of binary images on a mesh connected massively parallel processor," *Computer Vision, Graphics, and Image Processing*, vol. 45, no. 2, pp. 133 – 149, 1989.

[9] Yujie Han and Robert A. Wagner, "An efficient and fast parallel-connected component algorithm," *J. ACM*, vol. 37, no. 3, pp. 626–642, July 1990.

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

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

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

[13] "Euler cluster," .

[14] 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.