VF2 AND GLASGOW: PARALLEL INDUCED SUBGRAPH ISOMORPHISM SOLVERS

Philipp Lindenberger, Ajay Unagar, Kenza Amara, Chia-I Hu

Design of Parallel and High Performance Computing, Fall 2019
Department of Computer Science - ETH Zürich
Zürich, Switzerland

ABSTRACT

We propose two parallel implementations of popular subgraph isomorphism solvers: VF2 and Glasgow. The recursive DFS in VF2 was parallelized using OpenMP tasks, and we aimed to limit the excessive work done by additional threads. In Glasgow, we parallelized the complete algorithm using OpenMP, and we put special attention to compiler optimizations and OpenMP task amount limitation. Furthermore, a large section of Glasgow was parallelized with MPI one-sided communication. All the implementations are benchmarked on a wide range of graph pairs from literature, and we compare our Glasgow OpenMP implementation with the parallel version proposed by the author of Glasgow algorithm. We discuss when good scaling can be expected, and where improvements are possible.

1. INTRODUCTION

The subgraph isomorphism problem is concerned with finding a pattern graph within a larger target graph. Applications include chem- and bioinformatics, design of electric circuits, computer vision, and pattern recognition. Multiple algorithms exist for subgraph isomorphism, such as nRF+ [1], VF2 [2], and Glasgow [3], the common idea in most of them is similar to the constraint-based search approach by Ullmann [4].

We focus on VF2 and Glasgow, as VF2 is commonly used for benchmarking and Glasgow is currently the fastest algorithm for solving hard instances. Experiments by McCreesh and Prosser [3] also indicate that VF2 is the best choice for short runtimes, whereas Glasgow has better success rates on harder problems.

Subgraph Isomorphism is an NP-hard problem. Even though there are some very good heuristic algorithms, they don't work efficiently for different applications. The aim of this work is to introduce parallelism for these algorithms that scale on a variety of problems.

In this work, we present our strategies to parallelize VF2 and Glasgow algorithm. We discuss how shared memory and distributed memory paradigm works in our context. We will present and discuss our results using speedup plots and runtime analysis of algorithms.

2. ALGORITHMS AND RESEARCH

There are two types of subgraph isomorphism: induced and non-induced. For non-induced subgraph isomorphism, additional edges not contained between the nodes in the pattern graph can exist in the target graph. For induced SI, all edges between our mapped nodes have to be included, additional edges within the target graph are not allowed; this is the definition we went for in this project. Note that to compare with VF2, we adapted the Glasgow algorithm to solve for induced subgraph isomorphism. Additionally, we allow loops (an edge from a node to itself) and restrict ourselves to non-attributed, undirected graphs. Throughout this work, we stop the search if a satisfactory mapping has been found, i.e. we look if pattern and target graph are isomorphic to each other rather than finding all possible mappings.

VF2. VF2 [2] algorithm is a state-space exploration algorithm, which maintains partially mapped nodes between pattern and target graph. We present pseudo code for this in Algorithm 1. It uses the partially mapped *state* to decide which pair of *candidates* (line 3) to explore and introduces *feasibility rules* (line 4) to prune its search tree. These feasibility rules check three necessary constraints for subgraph isomorphism. Then it recursively *searches* (line 7) by adding each feasible pair to the current state variable. We can see that the algorithm dynamically expands its search space and does DFS to find a solution. However, it might explore an exponential number of states in the worst case.

Algorithm 1 VF2

```
1: procedure VF2(pattern, target, state=φ)
2:     if state is full then return State
3:     candidates ← findCandidates(pattern, target, state)
4:     for each candidate c in candidates do
5:         if c satisfy feasibility rules then
6:             NewState ← state.addCandidates(c)
7:             results ← VF2(pattern, target, NewState)
```

Algorithm 2 Glasgow (induced subgraph isomorphism)

1: **procedure** GLASGOW(pattern, target, k, l)
2: **if** *pattern.size* $>$ *target.size* **then return** false
3: Remove isolated vertices in *pattern*
4: $L \leftarrow$ Build Supplemental Graphs (k,l)
5: $D \leftarrow$ Initialize Domains
6: **if** ! Check all different (D) **then return** false
7: *result* $\leftarrow$ search & assign (D, L)

Glasgow. The main idea behind Glasgow (Alg. 2) is to introduce implied constraints, which are generated by including supplemental graphs (line 4). A supplemental graph defined by parameters (k, l) has an edge between two nodes if there are at least k paths of length l between them in the initial graph. We do this for $k = 2, 3$ and $l = 1, 2, 3$. Then, we look for a mapping which is simultaneously a subgraph isomorphism between all those pairs of graphs in L. [3]

The second step is initialising a set of nodes in the target graph where a pattern node can be mapped to. During the initialisation of domains (line 5), we check for degree and neighbourhood consistency between all the supplemental pattern-target pairs.

After checking an all-different constraint (line 6), we start a recursive search (line 7). After some sorting-heuristics, we pick a pattern node p, and loop over the target nodes in its domain D_p. With this assignment in mind, we update the other nodes' domains and remove target nodes from their domain where the mapping is impossible. As long as all nodes can be mapped, we recursively search again. Otherwise, we try the next node in our initial domain D_p.

The paper [3] presented a version of Glasgow algorithm for non-induced subgraph isomorphism. As we aimed for induced SI, we adapted the algorithm where required, while maintaining the general structure. We add the constraint that if v is not connected to w in the pattern graph, then their image (their paired vertex in the mapping) $i(v)$ and $i(w)$ in the target graphs are neither [5].

We also use a different graph structure than the one proposed in the paper [3]. While the authors use bit-set graphs, we use C++ vectors to store our adjacency matrices. Nevertheless, an adaption to bit-set graphs was considered but not realized yet.

3. PARALLEL IMPLEMENTATIONS

VF2 OpenMP. As explained before, VF2 recursively employs a Depth First Search (DFS) strategy on search space to find an optimal solution. We tried to parallelize this recursive search using data parallelism. In this context, all threads can work independently on some parts of the tree and keep expanding the search tree until one finds a solution. At this point, the terminal thread needs to communicate with every other thread to stop processing further and return the result. A similar approach has been tried in the past [6], the only difference being they have explored the whole search space without stopping at the first solution. Ideally, a termination call is the only communication needed between threads. In such a case, one would expect good load balancing between threads and higher speedups. However, we will explain some limitations of this strategy below.

We implemented a parallel recursive search using OpenMP tasks. OpenMP tasks provide a good way to parallelize recursive functions [7]. In this approach, one thread creates a pool of subtasks from a single call of a function. Once all tasks are generated threads can work independently on each task and might generate nested tasks if required. In our case, one thread will create subtasks for some search tree branches at a specific level. However, a drawback of this approach is that tasks are not guaranteed to run in the order they are generated. This has big implications on our algorithm since our strategy is to run DFS. Without a specific order, we might start exploring a different branch of the search tree. Since threads working on each branch will generate more tasks (search space to explore), this might slow down our algorithm. We tried to get around this using OpenMP tasks priority, however, it did not give us any improvement. Another drawback of tasks is that with nested tasking, the task generation process might overcome actual work. To solve this, we used the final-clause to make sure we do not generate more tasks after a thread reaches a certain depth.

Another approach to parallelize recursive algorithms is to use a stack-based representation for each function call. In this approach, rather than recursively calling functions, we put newly generated states in a stack (Last In First Out - LIFO). However, this approach requires communication between threads for each Push and Pull. Also, writing the states to the main stack was a costly operation and it dominates the runtime in the cases where we need to explore a large number of states. Since this overhead was very large for our datasets, we did not pursue this option further and we leave it as an open question for further investigation.

Glasgow MPI. We use MPI to parallelize the function *build supplemental graphs* from the Glasgow algorithm. Each processor works on a subset of vertices according to its rank. It builds the subgraphs for all the supplemental graphs. The processor 0 (master), P_0, constructs the final supplemental graphs by combining the information of every processor (servants), and their subgraphs.

Each servant puts in a pattern and target buffer the edges found in the supplemental graphs, connected to its subset of vertices. The master gathers the buffers. By iterating on the buffers, the master constructs two objects *pattern graphs* and *target graphs* that contain the edges for all pairs of supplemental graphs. For small graphs, the two-sided

communication worked well, but we faced some problems with memory space for bigger graphs. One-sided communication helped. We open a window for processor 0, so that each processor can put its pattern and target buffers into the window. The master iterates on the buffers to add edges into the two global objects, *pattern graphs* and *target graphs*.

With two-sided communication, using MPI_Gather, we faced some difficulties with memory space for big graphs. Instead of sending a buffer containing all the edges for a subset of vertices, we thought of sending each new edge at a time. Using, MPI_Send and MPI_Recv, we did not have any problem with the size of information in the communications between servants and master. However, we quickly had too many communications to handle. Then, we looked at one-sided communication: if each processor put its buffers into a window with MPI_Put, the communication servants-master is easier to manage. Since the result of concurrent MPI_Put is undefined, we used fences to achieve active synchronization on the memory window. In this case, we didn't face any space limit. Alternatively, we have also tried to open a window for each processor, and use MPI_Get for the master to fetch the buffers from the servants. The code to implement this method was slightly more complex than with MPI_Put.

Glasgow OpenMP. Initial runtime tests showed that for graphs with lots of nodes, *building supplemental graphs* dominates runtime. This method has a simple structure: In total 9 for-loops, each nested in an if-clause and all independent. But instead of parallelizing these blocks which are difficult to balance, we parallelize the outermost for-loop within each block, where iterations are also independent. To limit the overhead, we create the threads at the start of the method and just fork them at the beginning of each loop. Since each call from the outermost for-loop is independent, we add the no-wait clause for even better load balancing, as well as dynamic schedulers (varying block sizes did not influence the performance). This method scaled quite well without compiler optimization (i.e. optimization level *-O0*), and while switching on compiler optimization did reduce the sequential runtime, the parallel runtimes failed to decrease any more on strong scaling, see fig. 1.

Investigations on this behaviour are suggesting that the compiler removes the fork-call at the beginning of each section because it does not understand how to optimize the reads from other supplemental graphs which are written in parallel, although we guaranteed mutual exclusion with an if-clause. To solve this, we proposed a simpler method: Instead of reading, we count how often a connection occurs, and we store it in a vector. Finally, we loop (in parallel) over this vector, and build the supplemental graphs according to the edge-count in the vector. Now, compiler optimization is working as expected, and we even managed to reduce the sequential runtime.

Second, we tackled the *initialization of domains*. This

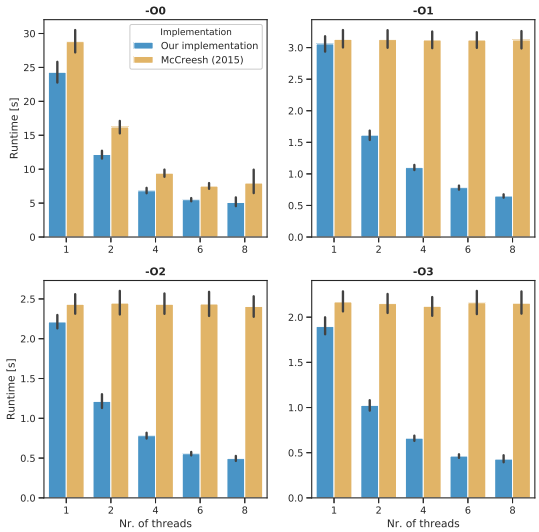

Fig. 1. Runtimes w.r.t. compiler optimization levels (Dataset: scalefree-F, N=20).

method is comprised of a while-loop with multiple nested for-loops. Since the while-loop is sequential and the number of iterations is data-dependent, we forked on the nested for-loops. We faced two challenges there: The write to a union domain needs to be in a critical section, and just wrapping for-loops with *pragma omp parallel for* would create a huge thread-creation overhead, since threads are opened and closed again in every iteration in the while loop. We therefore create the threads outside the method and fork them at the beginning of a parallel section.

To make *search & assign* on top of the previously initialized domains parallel, we use OpenMP tasks. Investigating the topology of the average search tree revealed that while it is wide in the first couple of levels, the assignments in each node prunes the available domains so that the branch is rendered almost sequential, i.e. every node can only be mapped to one target-node. Furthermore, we are most likely to make wrong decisions on the top levels, where the value-ordering heuristics are not so confident. Therefore, we let the first thread run through the first level, creating new tasks if the assignment is successful. To guarantee that at least one thread follows the sequential search order, preventing worse than sequential runtime, an if-clause associated to the task creation evaluates to false (i.e. threads executes the new task immediately) in two cases: When there is no free thread available to instantly take the task, or if we reached a certain, empirically obtained threshold level, where the branch is likely to be almost sequential. If the second case is triggered, this thread actually always goes depth-first from here instead of looping over all possible assignments first. If a thread finds a solution, we atomically write the node correspondences and forbid other threads to overwrite assignments. Finally, we cancel all open tasks and exit the search.

4. EXPERIMENTAL RESULTS

Setup and Dataset.

Our algorithms were implemented in C++, and compiled using gcc version 4.8.2 (GCC). For OpenMP, we used version 4.5 and the following flags: -O3 -g -fopenmp. For MPI, we used the Open MPI 1.6.5 version. We performed our experiments on the Euler clusters, testing up to 16 processors and 16 threads. For evaluation, the "LV", "SI", "scalefree" and the "images" [5] families were used to evaluate both Glasgow and VF2, where their pattern/target pairs are a mix of satisfiable and unsatisfiable queries. A brief summary of the number of instances and the number of nodes within pattern/target graphs are provided in Table 1. Since some instances require much more time to solve, we used timeouts to limit the explorations on these cases.

Table 1. Families of benchmark instances

Dataset	# of Nodes		Nr. pairs
	Target	Pattern	
LV	10-128	10-128	793
SI	200-1296	20-60% of target	1170
Scalefree	200-1000	90% of target	100
Images	201-5631	8-199	222

VF2 OpenMP results. VF2 algorithm is slow on large graphs. To do fast testing and maintain balanced runtimes, we terminated at runtimes which are greater than 1000 seconds in the sequential or single-threaded version. As we discussed, the parallel implementation of VF2 suffers from doing extra work. As we increase the number of threads, we might do more unusable work. As we can see from fig. 2, the algorithm does not scale very well with increasing number of threads. Another thing to notice is that speedup variability increases with the number of threads. This can be explained by speculative redundant work done by parallel threads.

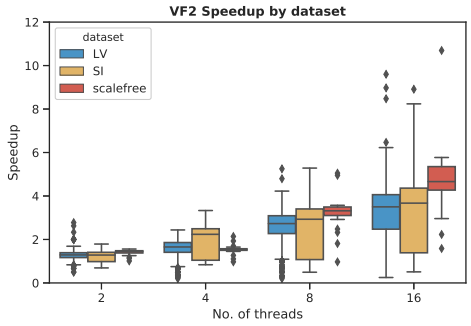

Fig. 2. Performance of VF2 OpenMP implementation (speedup w.r.t. single thread) (Dataset size: LV = 472, SI = 595, Scalefree = 24)

Another interesting thing to look at is whether the algorithm scales with larger graph search space (i.e. larger runtimes). As we can see from fig. 3, speedups increase with runtime for a single thread, because now parallel work done by threads is more likely to be useful.

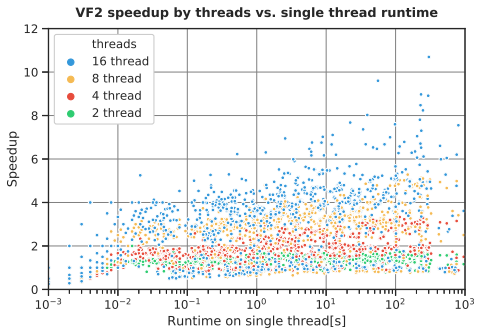

Fig. 3. Performance overview of all successfully conducted experiments (speed up w.r.t. single thread) (N = 1091)

Glasgow MPI results. MPI parallelism was introduced for the *build supplemental graph* method of Glasgow, and we tested it on a subset of pairs from three of our graph datasets (SI, LV and scalefree). From fig. 4, we observe good speedups with efficiencies of above 50% at 16 processors. The missing efficiency can be explained by overhead and especially due to difficult work balancing when splitting the graphs for each processor to work on (every processor gets an equal amount of nodes, but densities may vary).

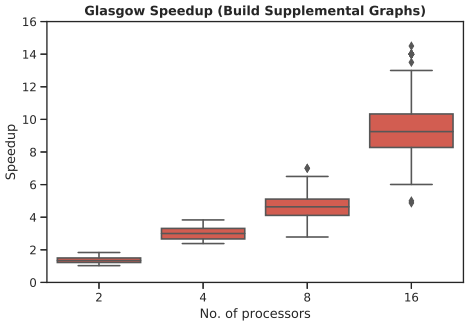

Fig. 4. Performance overview of *build supplemental graphs* on MPI (speed up w.r.t. single thread) (N = 486)

Glasgow OpenMP results. All tests were completed multiple times and on different numbers of threads. To limit the total runtime, a timeout of 600 seconds was set. Aborted experiments have been removed from the results. All experiments in this section were set to cancel if a solution has been found, thus computed runtimes will show high variances (if the solution is found on the left-most branch, our parallel search will not yield any performance improvements).

A total of 1926 instances have been solved successfully, while 359 instances aborted due to the set time limit. In fig. 5, the conducted experiments are visualized. The previously formulated hypothesis, of expecting high variances due to

cancellation when the solution is found, manifests itself. Especially at lower runtimes ($< 1.0s$), we observe hardly any speedup. The heavy pruning of the search space from using implied constraints via supplemental graphs often makes *search & assign* fully sequential for smaller graphs, and since this method is dominant on easy instances, we do not observe great scaling there. For harder graph pairs, we have to search through a bigger chunk of the search space, thus speculative work done is more often effective, and we observe higher efficiencies. Furthermore, on simpler instances with runtimes $< 1.0s$, thread-creation overhead further limits the improvements.

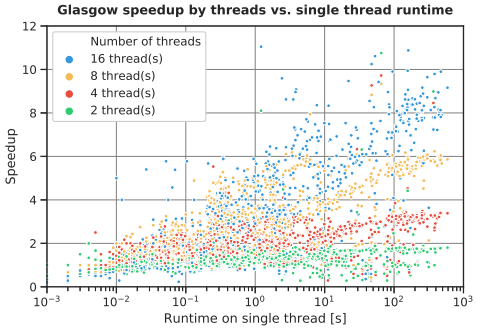

Fig. 5. Performance overview of all succesfully conducted experiments (N=1926, speedup w.r.t. single thread).

Our experiments are comprised of different graph families, thus we were interested in how good our implementation scales on each of them. In fig. 6, we observe the best improvements for the PR-15 and CVIU-11 image datasets. Those graphs generally have lots of nodes but relatively low densities and most of the pairs do not show subgraph isomorphism. For smaller, complex cases, especially the SI dataset, we observe the smallest speedups. Most of these graph pairs are isomorphic to each other, which hints that our scaling indeed depends on the isomorphism result.

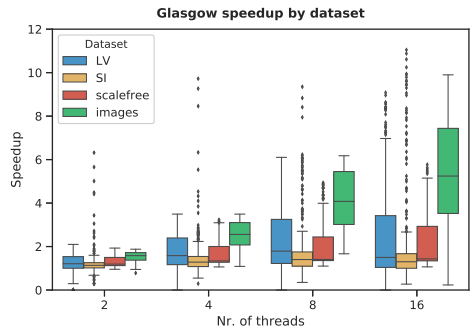

Fig. 6. Performance of Glasgow OpenMP implementation on different datasets w.r.t. single threaded runtime (Dataset size: LV=759, SI=860, scalefree=100, images=207).

Investigating runtime and scaling of our three main methods, see fig. 7, we observe that *search & assign* and *build supplemental graphs* have the largest runtimes on a single

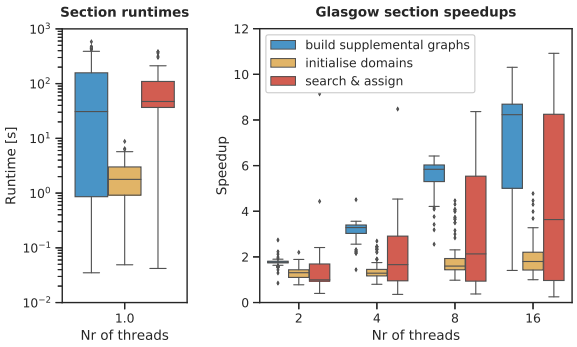

Fig. 7. Section runtimes on 1 thread and speedups by section(w.r.t single thread runtime), only instances with seq. runtimes $> 30.0s$ included (N=407).

thread, and while both have good performance improvements, *build supplemental graphs* scales best. For search, we observe large variances because of the search space being completely sequential on satisfactory instances, while on bad instances we look on a bigger chunk of the search space, thus speculative work done is actually beneficial. Initialize domains has low speedups, but this is most probably due to being dominated by thread-creation overheads, taking the generally low runtime into account.

5. COMPARISONS

Glasgow comparison with literature. The algorithm proposed in [3] was also parallelized using C++ native threads. Therefore, we compare the Glasgow OpenMP implementation against the original parallel method. Though, there are some key differences between the two implementations:

- The original paper worked on bitset-graphs [3], which clearly outperforms our implementation (adjacency matrix with two-dimensional vector). Therefore, the total runtime will certainly be worse in our implementation.
- McCreesh implemented work stealing, which certainly improves load balancing in the search & assign method.
- Our implementation relies on the new proposed optimizations in *build supplemental graphs* presented in section 3.
- The algorithmic setup was non-induced subgraph isomorphism for both implementations, as opposed to the other implementations discussed in this work.

All experiments were conducted on the same experimental setup as outlined in section 4, and with compiler optimization -O3.

Fig. 8 shows the number of instances that can be solved up to a certain runtime on 16 threads, both for our implementation and McCreesh's, respectively. Latter is clearly outperforming us by a factor of 10+. Again, this is most probably because of the different graph implementations.

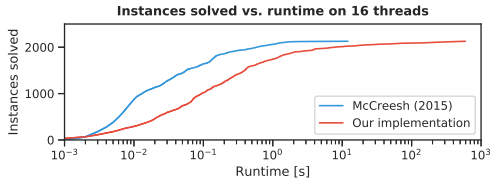

Fig. 8. Comparison of solved instances by runtime between our implementation and literature (N=2129).

Comparing efficiency of both implementations, see fig. 9, we actually observe less differences. The median of the speedup is almost identical for both implementations, but ours has much higher and non-symmetrical variance. Though we have not implemented work stealing, we outperform them on several instances. Reasons for that are probably really good task scheduling by OpenMP and improved total task amount limitation and fast cancellation in our code. Finally, also our adapted build supplemental graph section might have impacted this plot, but we could not verify this yet since the original code does not provide sectional runtimes.

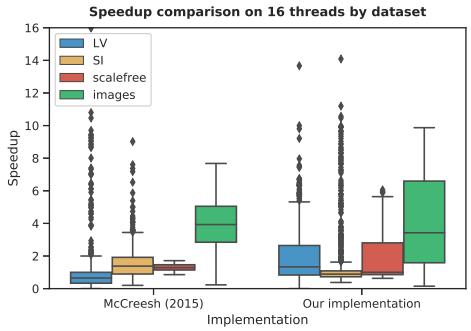

Fig. 9. Comparison of speedup on 16 threads by datasets for all succesful experiments conducted (N=2129).

Glasgow vs. VF2. Comparing the two algorithms we implemented, we observe what literature ([3] [5]) suggests: Glasgow algorithm is faster than VF2, though the differences are not that massive, see fig. 10. On lower instances (runtimes less than 10ms), VF2 performs better than Glasgow, because supplemental graphs just add additional overhead there. The implied constraints do not make up this difference on such small instances. On harder instances, we see that Glasgow performs significantly better because there the implied-constraints heavily prune the domains.

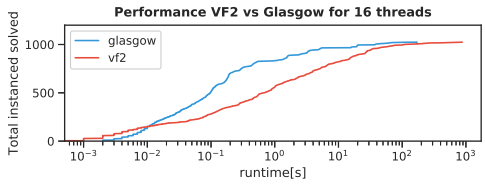

Fig. 10. Comparing performance of VF2 and Glasgow with 16 threads (compiler optimization -O3)

6. CONCLUSIONS

We presented an analytical and experimental evaluation of two algorithms - VF2 and Glasgow- using OpenMP, and MPI. Parallelizing sequential algorithms require attention to how compiler optimization can hinder efficiency. Our implementations take this into consideration and we redo parts of the original code to make it efficient. We also discussed limitations of OpenMP tasks in solving recursive functions where the strong ordering of the threads is of high importance. In MPI-communication with large data transfer, we should consider switching to one-sided communication. We showed how we could leverage this idea for our algorithm.

Future Work. The biggest drawback of our implementation remains the slow graph implementation. Therefore, we suggest using bitset-graphs [3], and combine them with SIMD fine-grain parallelism. Initial tests suggest that SIMD on our graph methods might bring significant performance improvements in the Glasgow algorithm. Work stealing, e.g. with Intel TBB, combined with our task limitation strategies are also potential optimizations.

7. REFERENCES

[1] J. Larrosa and G. Valiente, "Constraint satisfaction algorithms for graph pattern matching," *Mathematical Structures in Computer Science*, vol. 12, no. 4, pp. 403422, 2002.

[2] L. P. Cordella et al., "A (sub)graph isomorphism algorithm for matching large graphs," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 26, no. 10, pp. 1367–1372, Oct 2004.

[3] Ciaran McCreesh and Patrick Prosser, "A parallel, backjumping subgraph isomorphism algorithm using supplemental graphs," in *International conference on principles and practice of constraint programming*. Springer, 2015, pp. 295–312.

[4] J. R. Ullmann, "An algorithm for subgraph isomorphism," *J. ACM*, vol. 23, no. 1, pp. 31–42, Jan. 1976.

[5] Blair Archibald et al., "Sequential and parallel solution-biased search for subgraph algorithms," in *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*. Springer, 2019, pp. 20–38.

[6] Vincenzo Carletti et al., "A parallel algorithm for subgraph isomorphism," in *Graph Based Representation in Pattern Recognition*, IAPR-TC-15 Internation Workshop GbRPR, Ed., 2019, vol. 12, pp. 141–151.

[7] Ruud Van der Pas, "Openmp tasking explained," online: http://openmp.org/wp-content/uploads/sc13.tasking.ruud.pdf.