Karlsruhe Institute of Technology

Parallel FlowCutter Refinement for Hypergraph Partitioning

Master Thesis of

Florian Grötschla

At the Department of Informatics
Institute of Theoretical Informatics

Reviewers:	Torsten Ueckerdt
	Prof. Dr. Peter Sanders
Advisors:	Lars Gottesbüren

Time Period: 1st April 2020 – 5th October 2020

Statement of Authorship

Karlsruhe, 5th October 2020

Abstract

The balanced k-way hypergraph partitioning problem (HGP) has usages in a wide array of applications where performance is critical. Most partitioners use multilevel schemes that coarsen the hypergraph, find an initial partitioning and apply refinement algorithms while iteratively uncoarsening the hypergraph again. FlowCutter is a flow-based algorithm that uses the max-flow-min-cut theorem to find small cuts between blocks of the partition by solving a sequence of flow problems. This thesis presents parallelizations of flow algorithms that can be used in the weighted HyperFlowCutter framework (WHFC), an adaption of FlowCutter, that works for weighted hypergraphs, as well as an algorithm to extract smaller hypergraphs around an existing cut that are used as the input for the flow-based refinement. We also explore two approaches for parallel partitioning algorithms that use the fast sequential partitioner PaToH to find initial partitions and apply the flow-based refinement in parallel. One of them uses recursive bisection to split the hypergraph into increasingly smaller blocks, the second algorithm uses an initial k-way partition and schedules block pairs for parallel refinement steps. We conclude with an analysis of the algorithms in an experimental evaluation of our implementation. We show that our parallelized WHFC implementation provides good speed-ups and the sequential implementation of our Push-Relabel flow algorithm outperforms the existing Dinic implementation. The parallel refinement profits from the utilization of the parallel extraction and flow algorithms, resulting in a partitioner that scales well.

Deutsche Zusammenfassung

Das balancierte k-wege Hypergraph Partitionierungsproblem (HGP) findet Anwendungen in vielen Bereeichen in denen Performanz essentiell ist. Die meisten Partitionierer nutzen Multilevel-Ansätze, die den Hypergraph zuerst vergröbern, eine initiale Partitionierung berechnen und dann den Hypergraph iterativ zum Original zurückbauen während Verfeinerungsalgorithmen nach jedem Schritt angewendet werden. FlowCutter ist ein flussbasierter Verfeinerungsalgorithmus, der das Max-Flow-Min-Cut-Theorem benutzt um durch das Lösen einer Serie von Flussproblemen den Schnitt zwischen Blöcken der Partition zu verbessern. Die Thesis beschäftigt sich mit der Parallellisierung von Flussalgorithmen die im *weighted HyperFlowCutter* framework (WHFC), eine Adaption von FlowCutter für gewichtete Hypergraphen, eingesetzt werden können. Außerdem wird ein Algorithmus vorgestellt mit dessen Hilfe kleinere Hypergraphen um einen existierenden Schnitt extrahiert werden können um diese als Instanz für den Flussalgorithmus zu verwenden. Weiter werden zwei Ansätze für parallele Partitionierungsalgorithmen vorgestellt, die den sequentiellen Partitionierer PaToH für die Berechnung von initialen Partitionen verwenden und danach den flussbasierten Verfeinerungsalgorithmus parallel ausführen. Einer nutzt rekursive Bisektion um den Hypergraph in immer kleiner werdende Blöcke zu zerteilen, der zweite Algorithmus verwendet eine initiale k-wege Partition und plant die Ausführung von Verfeinerungsschritten, welche dann parallel ausgeführt werden können. Abschließend analysieren wir die vorgestellten Algorithmen in einer experimentellen Evaluation unserer Implementierung. Wir zeigen dass unsere parallele WHFC-Implementierung gute Speed-Ups aufweist und dass die sequentielle Implementierung unseres Push-Relabel Flussalgorithmus schneller ist als die existierende mit Dinic. Das parallele Refinement profitiert vom Einsatz der parallelen Extraktion und Flussalgorithmen und führt zu einem Partitionierer der gut skaliert.

Contents

1. Introduction **1**

2. Preliminaries **3**

3. Related work **5**

4. Partitioning approaches **9**
- 4.1. Recursive Bisection . . . 9
- 4.2. Using an initial k-way partitioning . . . 11
  - 4.2.1. Sequential k-way scheduling . . . 11
  - 4.2.2. Parallel k-way scheduling . . . 12

5. Snapshot extraction **17**
- 5.1. Parallel BFS . . . 17
  - 5.1.1. Visiting vertices . . . 18
  - 5.1.2. Scanning cut hyperedges . . . 19
- 5.2. Building the hypergraph . . . 19
  - 5.2.1. Vertex id mapping . . . 20
  - 5.2.2. CSR representation of the hypergraph . . . 21
  - 5.2.3. Adding hyperedges . . . 21

6. Dinic's max-flow algorithm **23**
- 6.1. Building the layered network . . . 23
- 6.2. Finding augmenting paths . . . 25

7. Push-Relabel **27**
- 7.1. Push-Relabel for graphs . . . 27
- 7.2. Sequential Push-Relabel implementation . . . 29
  - 7.2.1. Overview of the max-flow algorithm . . . 29
  - 7.2.2. Max preflow . . . 30
  - 7.2.3. Global Relabeling . . . 32
  - 7.2.4. Flow decomposition . . . 32
  - 7.2.5. Adaptions for the FlowCutter framework . . . 34
  - 7.2.6. Optimizations . . . 34
    - 7.2.6.1. Current arc datastructure . . . 34
    - 7.2.6.2. Current edge for depth first search . . . 35
    - 7.2.6.3. Sorting pin arrays . . . 35
- 7.3. Parallel Push-Relabel . . . 35
  - 7.3.1. Pulse operation . . . 36
  - 7.3.2. Discharge . . . 37

8. Overview of the main datastructures **39**
- 8.1. Lawler-FlowHypergraph-Datastructure . . . 39

8.2. Partition with cut-edge precomputation 40
8.2.1. Obtaining cut hyperedges 41
8.2.2. Changing the block of a vertex 41

9. Experimental evaluation **43**
9.1. Benchmark set 43
9.2. Methology 43
9.3. Parallelization of extraction and flow algorithms 44
9.3.1. Snapshot extraction 44
9.3.2. Parallelization of building the layered network 46
9.3.3. Push-Relabel 46
9.4. Evaluation of partitioners 49
9.4.1. Partitioning quality 50
9.4.2. Running times of partitioners 51
9.4.3. Scalability of the k-way partitioners 53

10. Conclusion **57**

Bibliography **59**

Appendix **65**
A. Set of hypergraphs used for the experiments 65

1. Introduction

Hypergraphs are generalized graphs where a hyperedge consists of a set of vertices and can therefore have an arbitrary number of endpoints. An ε-balanced k-way partitioning of a hypergraph is a partition of its vertex set into disjoint blocks such that no block has a weight more than $1 + \varepsilon$ times the average block weight. The balanced k-way hypergraph partitioning problem (HGP) asks for a solution to this problem that minimizes a certain metric, in our case the *connectivity metric*, reflecting the number of blocks that hyperedges are part of. These partitionings find applications in domains like VLSI design, for example to place components of a circuit [AK95, KAKS99], for the parallelization of algorithms like parallel sparse-matrix vector multiplication [CA99, CA95], to optimize the data placement and task assignment in workflows [ÇKU11] or as a preprocessing step in distributed computing where blocks of the hypergraph are assigned to machines, for example in machine learning [HZY15]. On the other hand, the problem is known to be NP-hard [Len90] and different heuristics lead to varying solution quality. Because instances can become big in these areas, we need efficient and scalable algorithms.

For the multi-objective optimization of runtime and solution quality we focus on algorithmic approaches that target shorter running times and make compromises in quality. To reach this goal, we use the fast multi-level partitioner PaToH [CA99] and apply the flow-based FlowCutter [HS18] algorithm as a refinement step [GHW19] to an initial partitioning. This refinement method uses the max-flow-min-cut theorem and improves a cut by solving a sequence of incremental flow problems. The partitioner PaToH lends itself to this task as it is very fast, but not the best in quality. Here, the flow-based refinement can be applied as a method that still tends to be fast, even on bigger instances, and compensates the disadvantages of PaToH by improving the quality. Because we work with hypergraphs, we do not use the original FlowCutter implementation for graphs, but the adapted version for weighted hypergraphs, the weighted HyperFlowCutter algorithm (WHFC) [GHSW20]. We start with this framework and parallelize the steps that are consuming the most running time. Similar to the algorithm introduced in [GHSW20] and going back to ideas of KaFFPa [SS11], to apply the flow-based refinement on a pair of blocks whose cut should be refined, we extract a smaller hypergraph around the cut using breadth first searches into both blocks and apply the algorithm on this instance.

This has the advantage that the flow-refinement works on a smaller hypergraph and still delivers results that can improve the cut between the blocks. We call this extracted hypergraph *snapshot* and discuss a parallelization of the extraction in Chapter 5. Then, we look into ways to parallelize WHFC itself by using parallel algorithms to compute the max-flow. We start with the Dinic algorithm [Din70] that is already implemented in the sequential code of WHFC and parallelize the computation of the layered network that it uses to find augmenting paths. The parallelization of the procedure that finds augmenting paths proved to be more difficult. We proceed with an algorithm that lends itself more easily to parallelization: The Push-Relabel algorithm by Goldberg and Tarjan [GT88] with optimizations for the parallel case proposed by [BBS15]. To apply our flow-based refinement in a fully fledged hypergraph partitioner, we use (parallel) scheduling schemes that apply the refinement to block pairs of the partition. To test the effectiveness and scalability of our algorithms, we then compare different versions in an experimental evaluation of our implementation on a set of instances stemming from different fields. The parallelization of our code is realized with the TBB library [Phe08].

After going into basic concepts and giving an overview of related work in the field in Chapter 2 and 3, we discuss implementation details and possibilities for parallelization of two approaches that we use to partition a hypergraph and apply the refinement in Chapter 4. The first one uses recursive bisection, the second refines an initial k-way partition using an active block pair scheduling algorithm [SS11]. Then, we parallelize the main ingredients of the flow-based refinement: the snapshot extraction and the flow algorithm. Chapter 5 covers the parallel snapshot extraction. Afterwards, we discuss parallelization possibilities and challenges of the Dinic flow algorithm in Chapter 6. In Chapter 7, we first recapitulate a sequential variant of the Push-Relabel flow algorithm and conclude with a parallel variant. After we explain implementation details of some of the datastructures in Chapter 8, we end the thesis with the experimental evaluation of the algorithms.

2. Preliminaries

Hypergraphs

A *hypergraph* H is defined as a tuple $H = (V, E, \omega, \varphi)$ of a set of *vertices* V, a set of *hyperedges* E, a *weight function* $\omega : E \to \mathbb{N}$ for hyperedges and a weight function $\varphi : V \to \mathbb{N}$ for vertices. A hyperedge $e \in E$ is a subset of V and we call its elements *pins.* We say that a vertex $u \in V$ is *incident* to a hyperdge e if $u \in e$ and call vertices that are incident to the same hyperedge *adjacent.* With $I(u)$ we denote the set of all incident hyperedges of a vertex u. A set $V' \subset V$ defines the *induced* hypergraph $H[V'] := (V', \{e \cap V' \mid e \in E\})$ of H. Functions f are extended to sets X by $f(X) := \sum_{x \in X} f(x)$. We use different notations to distinguish between the elements of hypergraphs and graphs: A graph consists of nodes and edges, whereas a hypergraph consists of vertices and hyperedges.

Lawler network

To represent a hypergraph H as a graph, we use the *Lawler* construction. For a hyperedge $e \in E$, we define e_{in} and e_{out} to be nodes of the directed graph H_L that we call the *Lawler network.* We call nodes e_{in} *edge-in* nodes and e_{out} *edge-out* nodes. The sets of all edge-in and edge-out nodes are represented by $V_{\text{in}} := \{e_{\text{in}} \mid e \in E\}$ and $V_{\text{out}} := \{e_{\text{out}} \mid e \in E\}$ respectively. The Lawler network H_L of a hypergraph H is defined as a directed graph with nodes $V_L := V \cup V_{\text{in}} \cup V_{\text{out}}$, edges $E_L := \bigcup_{e \in E} (\{(u, e_{\text{in}}), (e_{\text{out}}, u) \mid u \in e\} \cup \{(e_{\text{in}}, e_{\text{out}})\})$ and a capacity function $c_L : E_L \to \mathbb{N} \cup \{\infty\}$ with $c_L(e_{\text{in}}, e_{\text{out}}) = \omega(e)\ \forall e \in E$ and $c_L(u, v) = \infty$ for the remaining edges $(u, v) \in E_L$.

Hypergraph Partitioning

A *k-way partition* Π of a hypergraph $H = (V, E)$ is a partition of V into k disjoint and non-empty sets $\Pi = \{V_1, V_2, \ldots V_k\}$. More formally: $\bigcup_{i=1}^{k} V_i = V$, $V_i \neq \emptyset\ \forall i \in [k]$ and $V_i \cap V_j = \emptyset\ \forall i, j \in [k], i \neq j$. We use $[k] := \{i \mid 0 < i \leq k\}$ to denote the set of natural numbers smaller or equal to k. We call the sets V_i *blocks* of the partition and will sometimes refer to i as the *partition id* of a block. To ease the notation, $\Pi(u)$ for a vertex u references its block id. The *cut* $E^{\text{cut}}(p_1, p_2) := \{e \in E \mid |e \cap V_{p_1}| > 0, |e \cap V_{p_2}| > 0\}$ between two blocks with ids p_1 and p_2 is the set of all hyperedges that have a pin both in p_1 and p_2. A k-way partition Π is *ε-balanced* for an $\varepsilon \in [0, 1)$, if each block V_i satisfies the *balance constraint* $\varphi(V_i) \leq (1 + \varepsilon)\frac{\varphi(V)}{k}$. For a hyperedge $e \in E$, the *connectivity set* is denoted as $\Lambda(e) := \{V_i \in \Pi \mid V_i \cap e \neq \emptyset\}$. The *connectivity* of the hyperedge $e \in E$ is defined as $\lambda(e) = |\Lambda(e)|$. The *connectivity metric* of a hypergraph $H = (V, E)$ is $\sum_{e \in E} \omega(e)(\lambda(e) - 1)$.

The *k-way hypergraph partitioning problem* asks for an ε-balanced k-way partition of a hypergraph H that minimizes the connectivity metric.

Network flows

We define a *flow network* $\mathcal{N} = (\mathcal{V}, \mathcal{E}, c, s, t)$ as a symmetric, directed graph with nodes $\mathcal{V}$, edges $\mathcal{E}$, a capacity function $c : E \to \mathbb{N}_0$ and two distinguished nodes $s, t \in \mathcal{V}$, the *source* and the *target* that are sometimes referred to as the *terminals*. A *flow* of $\mathcal{N}$ is a function $f : \mathcal{E} \to \mathbb{Z}$ fulfilling the following constraints:

$$f(e) \leq c(e) \ \forall e \in \mathcal{E} \qquad (\textit{capacity constraint})$$

$$f(u,v) = -f(v,u) \ \forall (u,v) \in \mathcal{E} \qquad (\textit{skew symmetry})$$

$$\sum_{(u,v)\in\mathcal{E}} f(u,v) = 0 \ \forall u \in \mathcal{V} \setminus \{s,t\} \qquad (\textit{flow conservation})$$

The *value* of a flow $|f| := \sum_{u\in\mathcal{V}} f(s,u)$ is the amount of flow that leaves s and thus has to arrive at t due to the constraints. The *residual capacity* $r_f(u,v) := c(u,v) - f(u,v)$ is defined for edges $(u,v) \in \mathcal{E}$. The *residual network* $\mathcal{N}_f = (\mathcal{V}, \mathcal{E}_f, r_f)$ for a flow f of $\mathcal{N}$ is a directed graph with nodes $\mathcal{V}$, edges $\mathcal{E}_f := \{(u,v) \in \mathcal{E} \mid r_f(u,v) > 0\}$ the *residual flow* r_f of f. An *augmenting path* is a path in $\mathcal{N}_f$. A flow f of $\mathcal{N}$ is *maximum* if there exists no flow f' of $\mathcal{N}$ with $|f'| > |f|$. A *cut* of a flow network $\mathcal{N}$ is a set of edges that disconnects s and t. The value of a cut is the sum of all edge-weights in the cut.

FlowCutter solves flow-problems with multiple sources and multiple targets. In this case, we have two terminal sets S and T that are non-empty disjoint subsets of $\mathcal{V}$. The only adaptions we have to make for the definitions is to exclude nodes $S \cup T$ from the flow conservation instead of just s and t. An augmenting path becomes and S-T-path and the value of the flow is the amount leaving S: $|f| = \sum_{s\in S,(s,u)\in\mathcal{E}} f(s,u)$. The set S_r of *source-reachable* nodes contains all nodes u where there exists a path from S to u in $\mathcal{N}_f$. The set T_r of *target-reachable* nodes contains all nodes u where there exists a path from u to T in $\mathcal{N}_f$.

Preflows

We use *preflows* as a weaker form of network flows for an efficient maximum flow algorithm. A preflow f_p is a flow where the flow conservation constraint is replaced by the weaker *nonnegativity constraint*:

$$\sum_{(u,v)\in\mathcal{E}} f_p(u,v) \geq 0 \ \forall u \in \mathcal{V} \setminus \{s\} \qquad (\textit{nonnegativity constraint})$$

We call the difference between ingoing and outgoing flow $e(u) := \sum_{(u,v)\in\mathcal{E}} f_p(u,v)$ the *excess* of the node u. A preflow f_p is maximum if there is no other preflow f'_p with bigger excess of t.

3. Related work

Hypergraph partitioning

Like the partitioning problem for graphs, the hypergraph partitioning problem is NP-complete [BJ92]. Therefore, partitioning algorithms employ heuristics to run in polynomial time. One of the most used is the multilevel heuristic [HL95]. This approach consists of three main phases: coarsening, initial partitioning and uncoarsening. First, we obtain a series of progressively smaller hypergraphs by contracting vertices. The contraction is done in a way that preserves the structure of the input hypergraph. When the hypergraph has become small enough, an initial partitioner is used to find a partitioning for it. In the last phase, the hypergraph is iteratively uncontracted and refinement algorithms are applied after each uncontraction to improve the cut. The contractions can be found using matchings, while some of the heuristics used for the refinement are: Fiduccia-Mattheyses [FM82], Kernighan-Lin [KL70], flow-based refinement (such as KaFFPa [SS11] for graphs), tabu search [Glo86, RPG96], label propagation [RAK07, UB13], the greedy refinement [LK13] and hill scanning of mt-Metis [LK16] or the usage of the probabilistic fanout as a target function employed in the social hash partitioner [KKP$^+$17].

The KaFFPa approach was adapted to hypergraphs in [HSS19] and works by extracting corridors around cuts and solving maximum flow problems using the Lawler network [Law73]. The authors of [SS11] also introduce an active block scheduling scheme to refine a k-way partition. It considers pairs of blocks for the refinement as long as they contribute to improvements of the cut. Another algorithm that uses the idea of the flow-based refinement is FlowCutter [HS18] which is presented in more detail in the next paragraph. For more work on (hyper)graph partitioning we refer the reader to the following survey articles: [BMS$^+$16, Tri06, AK95].

FlowCutter

The FlowCutter algorithm as introduced by Hamann and Strasser [HS18] was originally meant to compute bisections for Customizable Contraction Hierarchies [DSW14], a method to compute shortest paths in road networks, but can also be used to compute small, balanced cuts on general graphs. The algorithm was extended to hypergraphs by [GHW19] and is called HyperFlowCutter (HFC). FlowCutter only workw with unweighted (hyper)graphs, limiting its usage as a refinement algorithm in multilevel partitioning schemes where contracted hypergraphs can have weighted vertices as well as weighted hyperedges even if the input hypergraph is unweighted. This problem was addressed with the weighted

HyperFlowCutter algorithm (WHFC) [GHSW20] that can handle weighted hypergraphs and also introduces a method of computing maximum flows directly on the hypergraph. The algorithms in this thesis are implemented in the context of the WHFC framework.

The main idea of the FlowCutter bipartitioning algorithm is to use the max-flow-min-cut-theorem [FF56] to get a minimum cut by computing a maximum flow between two terminal sets S and T of source and target nodes. Starting with the sets $S = \{s\}$ and $T = \{t\}$ for some nodes s and t, the algorithm solves a sequence of maximum S-T flow problems with increasing cut size. The cut of a S-T flow induces the two cutsides S_r and T_r, which are the sets of nodes that are reachable in the residual network by S and T respectively. If one of the sides fulfills the given balance constraint, then we found a balanced cut and can terminate. In the other case, we extend the terminal set of the smaller cutside, i.e. $S = S_r$ or $T = T_r$. We take the smaller side to grow both sides evenly to guarantee that we will always be able to find a balanced cut. After the extension of one of the terminal sets, the flow is still a maximum S-T flow. To get a new cut in the next iteration, we add an additional node, the piercing node, to one side. With respect to the new sets, we then augment the existing flow to a maximum flow again.

The authors of [GHW19] propose ReBaHFC, a bipartitioning algorithm that uses HFC to refine an initial partition obtained by using the multilevel partitioner PaToH. ReBaHFC does not use the whole hypergraph for the flow network, but extracts a hypergraph by running two weight-constrained breadth first searches from boundary vertices of the cut into both blocks.

Flow algorithms

We use maximum flow algorithms to improve the cut of a bipartition. For a good introduction we refer the reader to [GT14]. Historically, one of the first specialized maximum flow algorithm was invented by Ford and Fulkerson [FF56]. It uses the notion of augmenting paths (i.e. a path with positive capacity in the residual network) that the algorithm computes in the network. For general graphs $G = (V, E)$ the runtime can not be bounded and it might not terminate, for integer capacities, the running time is bounded by $\mathcal{O}(|E| \cdot |f|)$ where f is the maximum flow. The algorithm of Edmonds and Karp [EK72] uses the same approach, but always augments the shortest augmenting path and thereby improves the runtime guarantee to $\mathcal{O}(|V| \cdot |E|^2)$. Dinic's algorithm [Din70] is also based on this idea and improves it by not searching for the shortest augmenting path in each step, but computing layered networks and finding *blocking* flows in them. The layered network consists of the set of shortest paths in the residual network from s to t. Therefore every augmenting path in the layered network is a shortest augmenting path. When there are no augmenting paths left in the layered network, a blocking flow has been achieved and the layered network is recomputed. The length of the shortest paths in the residual network increases now has to be longer than in the previous iteration and we compute the next blocking flow. The algorithm runs in $\mathcal{O}(|V|^2 \cdot |E|)$ time. Even and Itai proposed a version of the algorithm that uses a breadth first search to compute the layered network and searching for augmenting paths using a depth first search [Din06]. The implementation of Cherkassky [Che94a] added further optimizations like the non-explicit representation of the layered network, finding augmenting paths for one phase with a single DFS or skipping edges that were scanned before. The Dinic algorithm used in the WHFC framework is based on this highly optimized version from Cherkassky.

Instead of augmenting flow along paths and maintaining a flow during the execution of the algorithm, Karzanov [Kar74] introduced the concept of the preflow, a relaxation of the flow, that allows for changes of flow on single edges and not only paths. Goldberg and Tarjan use preflows in their Push-Relabel algorithm [GT88] to push flow from one node to another. For these preflows, nodes can have more incoming than outgoing flow; we call the difference between these the excess of a node. Nodes that have positive excess are called active and a discharge operation is used to push flow from active nodes to their neighbors. If an active node can not be discharged any more, it is relabeled. Some heuristics can improve the performance of the Push-Relabel method. For one, the choice of the next active node to discharge can have an impact on the runtime. For example, nodes can be considered in a FIFO-order or the node with highest label is chosen. Other useful heuristics include global relabeling of nodes and gap relabeling. The running time of the general Push-Relabel algorithm is in $\mathcal{O}(|V|^2 \cdot |E|)$, like Dinic's algorithm. The highest label node selection manages to push it down to $\mathcal{O}(|V|^2 \cdot \sqrt{|E|})$ while dynamic trees, also introduced by Goldberg and Tarjan [GT88] even has a theoretical running time of $\mathcal{O}(|V| \cdot |E| \cdot log(\frac{|V|^2}{|E|}))$. However they show in the same paper that dynamic trees are slower in practice.

Another max flow algorithm that is closely related to the one from Goldberg and Tarjan is Hochbaum's pseudoflow algorithm [Hoc08] that introduced the concept of pseudoflows to be used in flow algorithms. In addition to the mentioned flow algorithms, there also exist some algorithms that perform well for several problems in computer vision (for example image segmentation), due to the special structure of the instances. These algorithms include the Boykov-Kolmogorov (BK) algorithm [BK04] which has no strongly polynomial time bound, but is widely used in computer vision. The incremental breadth first search (IBFS) is bounded by $\mathcal{O}(|V|^2 \cdot |E|)$ like the Push-Relabel algorithm while still being competitive with BK [GHK+11] in practice. The more recent Excess IBFS is a generalized IBFS that uses preflows and has the same time bound [GHK+15].

Parallel flow algorithms

One of the advantages of the Push-Relabel algorithm is that it lends itself better to parallelization as flow can be pushed independently from one node to another without the need for a complete augmenting path to be found first. Shiloach and Vishkin proposed a parallel algorithm in 1981 [SV82] that can be seen as a predecessor to Goldberg and Tarjan's algorithm. They use the same approach as Dinic and compute the layered network, but employ preflows and push-operations to find maximum flows in the layered network. Goldberg and Tarjan themselves describe a version of the Push-Relabel algorithm in their 1988 paper that runs in distributed or parallel systems. The algorithm works with pulse operations that process all active nodes at once and mark nodes that became or still are active for the next round. Andersen and Setubal introduced a parallel implementation [AS95] with the distinguishing feature that the global relabeling heuristic is executed concurrently with the main algorithm. The algorithm of Hong [Hon08, HH10] works lock-free by adapting the push and relabel operations and using atomic operations, but is significantly more complicated. The parallel implementation from Baumstark et al. [BBS15] uses the original Push-Relabel approach proposed by Goldberg and changes the discharge operation to allow nodes to be relabeled more than once during one parallel pulse. It also employs a parallel BFS for the global relabeling. Furthermore, there exists a map-reduce based algorithm that uses the Ford-Fulkerson method to compute the maximum flow [HYW11].

Flow decomposition

We use flow decomposition to transform a preflow into a flow in the implementation of our Push-Relabel algorithm. The flow decomposition theorem [AMO93, FJF15] states that for each preoflow (or flow) f_p, we can find a set of flows along simple cycles and flows along simple paths $g_1, \dots, g_k$ whose union is f_p. If f_p is a flow, a decomposition consisting only of flows along s-t-paths can be found. In our use case, we want to keep the s-t paths for the maximum flow and remove all flow cycles or paths leading to nodes with excess that are not t. During the decomposition we will list these elements and act accordingly.

To decompose the s-t-preflow f_p, we can employ a depth first search in the residual network starting in s. At each step, the DFS can encounter an edge that leads to a node u that was previously visited. In this case, the recursion stack of the DFS contains a cycle W with positive flow that starts at the previous occurrence of u. We remove the minimum amount of flow that any edge on W has from the cycle and backtrack to u on the stack. The flow cycle we removed is one cycle of the decomposition. While progressing the search, we can also encounter nodes u with positive excess. We do a similar backtrack as before for the s-u-path on the stack, but remove at most $e(u)$ flow from the path. This path is also part of the decomposition. When reaching t we also encountered an s-t-path that we add. After the DFS finishes and can not reach any nodes from s, there can still remain some disconnected flow cycles in the network. They have to be treated separately and added to the decomposition.

4. Partitioning approaches

The FlowCutter algorithm can be used to improve bisections between two blocks of a partition. We discuss two existing approaches for general k-way partitioning that apply the refinement algorithm to pairs and manage to improve the total partition. They come with different advantages and disadvantages regarding the size of the instances that have to be refined and the amount of parallel executions of refinements that are made possible.

Recursive Bisection computes a series of 2-way partitioning (*bisections*) to split the hypergraph into pieces of increasingly smaller size. Directly after each bisection, we apply the flow-based refinement. Here, not only the refinement steps can be executed in parallel, but also the bisection algorithm. We use the idea and a scaling scheme from [SHH$^+$16].

The second method computes an initial k-way partition and then schedules block pairs for the flow-based refinement. The scheduling ensures that no two threads process the same block to let them work independently and without synchronization. To implement this, we use an active block scheduling similar to the one introduced by [SS11].

4.1. Recursive Bisection

The first algorithm divides the hypergraph into two blocks using an initial partitioner, refines the cut between these blocks and recursively calls the algorithm on the refined blocks. The two blocks contain $\left\lfloor \frac{k}{2} \right\rfloor$ and $\left\lceil \frac{k}{2} \right\rceil$ subpartitions and the recursion stops for $k = 1$. If k is even, it is sufficient to bisect the hypergraph into blocks of approximately equal size (up to some imbalance). If k is odd, the bisection has to be made imbalanced as one of the two blocks holds more of the final blocks. Furthermore, the imbalance parameter for each call of the initial partitioner has to be chosen in a way such that the final imbalance is no bigger than ε. Although this is a well known scheme that has been around for some time, we want to go into the details of our implementation to lay the basis for its experimental evaluation and make the results more comprehensible and reproducible.

The advantage of this approach is the inherent parallelism gained by the divide-and-conquer algorithm. The recursive calls can be parallelized very easily as they are completely independent: Once the two blocks were found by the initial partitioner and the cut has been improved by the refinement algorithm, we already know which of the two blocks will contain which blocks of the final partition. The number of parallel executions that are possible depends on the parameter k. With each level of the recursion, more blocks can be

Algorithm 1 Recursive bisection

```
Input: Hypergraph $H = (V, E)$, number of parts $k$, imbalance $\varepsilon$
Output: $\varepsilon$-balanced $k$-way partition $P$ of $H$
 1: function partitionRecursively($H = (V, E), k, k', \varepsilon, w_{\text{total}}$)
 2:     if $k = 1$ then return partition $P$ with $P(v) = 1\ \forall v \in V$
 3:     $k'_1 \leftarrow \left\lfloor \frac{k'}{2} \right\rfloor, \quad k'_2 \leftarrow \left\lceil \frac{k'}{2} \right\rceil$
 4:     $\varepsilon' \leftarrow \left((1+\varepsilon)\frac{k' \cdot w_{\text{total}}}{k \cdot \varphi(V)}\right)^{\frac{1}{\lceil log_2(k') \rceil}} - 1$
 5:     $w_1^{\max} \leftarrow (1+\varepsilon') \cdot \frac{k'_1}{k'} \cdot \varphi(V), \quad w_2^{\max} \leftarrow (1+\varepsilon') \cdot \frac{k'_2}{k'} \cdot \varphi(V)$
 6:     $P \leftarrow$ bisect($H, w_1^{\max}, w_2^{\max}$)
 7:     refine($P, H, w_1^{\max}, w_2^{\max}$)
 8:     if $k > 2$ then
 9:         $P_1^{sub} \leftarrow$ partitionRecursively($H[V_1], k, k'_1, \varepsilon, w_{\text{total}}$)
10:         $P_2^{sub} \leftarrow$ partitionRecursively($H[V_2], k, k'_2, \varepsilon, w_{\text{total}}$)
11:         for all $v \in V$ do                                  ▷ compute new ids
12:             if $P(v) = 1$ then
13:                 $P(v) = P_1^{sub}(v)$
14:             else
15:                 $P(v) = P_2^{sub}(v) + k'_1$
16:     return $P$
17: end function
```

processed in parallel. In the last level of the recursion, a total of at most $\left\lfloor \frac{k}{2} \right\rfloor$ bisections can be found in parallel, which is also the maximum amount of parallelization that we get.

An overview of the algorithm is given in 1. The number of blocks for the current instance we solve recursively is k' and the first call is using the parameters $H, k, k, \varepsilon, \varphi(V)$. Before we bisect the hypergraph, we establish some parameters. The variables k'_1 and k'_2 represent the number of blocks that V_1 and V_2 contain in the final partition and thus have to be split into. We use an adapted imbalance parameter ε' that restricts the maximum total weights $w_1^{\max}$ and $w_2^{\max}$ of the two blocks that we want to create. They can be deduced using the restrictions that the sizes of blocks for balanced partitionings have to fulfill. We use these values for the bisector and the refinement afterwards. They both need to respect this maximum size of the blocks while trying to improve the connectivity metric of the cut. Note that $w_1^{\max} = w_2^{\max}$ iff $k'_1 = k'_2$ and that we target a balanced bisection in this case. If $k'_1 \neq k'_2$ the bisection is imbalanced.

To guarantee ε-balance of the final k-way partition, we need to use an adapted imbalance parameter ε' for the bipartitioning. This is because using the bisector multiple times in a row increases the imbalance and simply using ε for each call can result in partitions with imbalance greater than ε. Instead, we use

$$\varepsilon' = \left((1+\varepsilon)\frac{k' \cdot w_{\text{total}}}{k \cdot \varphi(V)}\right)^{\frac{1}{\lceil log_2(k') \rceil}} - 1$$

as proposed by Ahkremtsev et al. [SHH+16]. The value of ε' is computed for each recursive call and chosen as the biggest value such that all succeeding bisections of the recursive calls could all use ε' and the final partition is still guaranteed to have an imbalance of at most ε. This works even if all following bisections output blocks of the maximally allowed weight. The idea is that we want to execute all bisections with roughly the same ε' and tailor it to be suitable even in the case that we always reach the maximum imbalance. However, the partitioner does not always output a maximally imbalanced bisection and as a result we can relax ε' in the following calls, meaning that we do not use the same one as before, but set it a bit higher such that all following recursive calls from this point can use this value. Doing so makes it possible that depending on the imbalance of bisections that the partitioner outputs, ε' can get bigger in the deeper recursive calls. A less strict ε' has the advantage that the partitioner and the refinement have more freedom in choosing which vertices to assign to which block.

Before we can proceed with call the algorithm recursively, we need to compute the induced subgraphs $H[V_1]$ and $H[V_2]$ that the recursive calls work on. While the construction is not made explicit in the pseudocode, it is straightforward: $H[V_i]$ contains the vertices V_i and for each hyperedge $e \in E$ a new hyperedge e' is added consisting only of the pins of e that are in V_i: $e' := e \cap V_i$. Hyperedges e' with $|e'| \leq 1$ are discarded. Therefore, an iteration over all hyperedges E is sufficient to construct $H[V_1]$ and $H[V_2]$.

Afterwards, we use the subpartitions P_1^{sub} and P_2^{sub} to update the partition ids of the final partition P. At this point, P maps vertices to ids 1 and 2, representing the two blocks V_1 and V_2. P_1^{sub} maps to ids $[1, \left\lfloor \frac{k}{2} \right\rfloor]$ and P_2^{sub} to $[1, \left\lceil \frac{k}{2} \right\rceil]$. For vertices $v \in V_1$ we take the partition id of P_1^{sub}, for $v \in V_2$ we add $\left\lfloor \frac{k}{2} \right\rfloor$ to the partition id of P_2^{sub} to avoid an overlap with the ids belonging to P_1^{sub}. To do so, we process each vertex of H and update the partition accordingly.

4.2. Using an initial k-way partitioning

The second approach to apply the refinement is to use an existing partitioner to find an initial k-way partition and schedule block pairs for refinement afterwards. We first show a version that only schedules one refinement step at the same time and is very close to the idea from [SS11] and then proceed with a scheduling scheme that allows us to do multiple refinements in parallel.

4.2.1. Sequential k-way scheduling

Algorithm 2 k-way refinement using active block scheduling

```
 1: function KWayRefinement($H, k, \varepsilon$)
 2:     $active(p) \leftarrow$ true $\forall p \in [k]$
 3:     $Q \leftarrow \emptyset$
 4:     while there exists at least one active block do
 5:         $Q \leftarrow \{(p_1, p_2) \in k \times k : (active(p_1)$ or $active(p_2))$ and $|E^{\text{cut}}(p_1, p_2)| > 0\}$
 6:         $active(p) \leftarrow$ false $\forall p \in [k]$
 7:         for $(p_1, p_2) \in Q$ do
 8:             refine($P, H, p_1, p_2$)
 9:             if $E^{\text{cut}}(p_1, p_2)$ improved then
10:                 $active(p_1) \leftarrow$ true
11:                 $active(p_2) \leftarrow$ true
12: end function
```

Algorithm 2 shows the active block scheduling algorithm that refines the partition P [SS11]. At its core, the algorithm keeps track of blocks that contributed to a refinement that could improve the cut in the last round and sets them as active. In the beginning, all blocks are active. In each iteration, all block-pairs with at least one active block and at least one hyperedge in the cut are added to Q. These pairs are then refined and blocks that contributed to an improvement are activated again. This goes on until no blocks are active any more.

The advantage of this refinement approach over the recursive bisection is the parallelization potential that it offers for the execution of refinements, which can all be run in parallel from the very beginning. Block-pairs can be refined independently from each other without additional adaptions to the flow-based refinement we use if no two threads work on the same block at the same time. In the next Section, we propose a parallel implementation of this scheduling scheme that aims to maximize the number of parallel refinements that can be executed.

4.2.2. Parallel k-way scheduling

To be keep the individual refinement steps self-contained and without the need to be aware of other refinements, we want each block to take part in at most one refinement at the same time. Otherwise, the problem arises that two threads working on the same block can both reassign vertices of it to other blocks, thus changing the input instance of the other thread. We avoid this by scheduling the assignment of block pairs to threads such that at most one thread is working on a given block. Consequently, the scheduling algorithm can not just assign any block pair $\{p_1, p_2\}$ to a thread that is waiting for work as p_1 or p_2 could currently undergo a refinement procedure. For each iteration of the algorithm, we use an active block scheduling similar to the sequential scheduling. At the beginning of each iteration we know which pairs should be considered for a refinement step. We call pairs that were already processed in the iteration *finished* and pairs that still have to be refined *open pairs*. It is the job of the scheduling algorithm to assign one of the open pairs to a thread that ran out of work. To reach a maximum level of parallelism, we want to keep all threads busy all the time. What gets in the way of this are pairs that depend on the same block. Even if there are open pairs left we might not be able to schedule them as they all have to refine blocks that are currently being processed. At the beginning of an iteration, when there are still a lot of open pairs left, this might not be an issue because we have a lot of pairs to choose from. But especially at the end of an iteration, when only few open pairs are left, the parallelism can be hindered strongly because all open pairs depend on a small set of blocks. In this case, we can only schedule a small number of pairs at the same time.

To make this more clear, consider the following example: We start with all blocks being active, so the set of open pairs consists of all possible block pairs. Assume that when scheduling new pairs, there was one block that never gets scheduled. Towards the end of the iteration, we will have a lot of open pairs that contain this block. Then, we always have to wait for the block to be processed until we can schedule a new pair that will very likely work on this block too. This leads to the refinement steps running sequentially towards the end.

The idea that we want to use for the following scheduling algorithm is that this situation could have been prevented by letting the block we avoided take part in some refinements at the beginning. We will try to process blocks evenly, meaning that we actively try to avoid a small number of blocks to be part of a lot of open pairs. We do this by favoring blocks that still have to undergo more refinement steps to ones that are part of less open pairs when scheduling a new pair.

The implementation of this approach is shown in Algorithm 3. To assess the number of open pairs that a block is part of, we count them as the number of *participations* of a block. Their values are updated during the execution to reflect the correct value. The main datastructure Q always contains a set of conflict free (i.e. no pairs work on the same block) open pairs that can be processed in parallel. After a thread took a pair of Q and processed it, we search for new pairs that can be added. Before we can process any pairs, Q needs to be initialized with a set of block pairs. The function INITIALIZEBLOCKPAIRS

Algorithm 3 Parallel k-way refinement

```
 1: function KWayRefinement($H, k, \varepsilon$)
 2:     $active(p_i) \leftarrow$ true $\forall p_i \in [k]$
 3:     $active'(p_i) \leftarrow$ false $\forall p_i \in [k]$

 4:     $scheduled(p_i) \leftarrow$ false $\forall p_i \in [k]$
 5:     while there exists at least one active block do
 6:         initializeBlockPairs()                                  ▷ fills $Q$ with pairs
 7:         parallel for $\{p_1, p_2\} \in Q$ do
 8:             $refiner_{\text{local}}$.refine($\{p_1, p_2\}$)
 9:             if $E^{\text{cut}}(p_1, p_2)$ improved then
10:                 $active'(p_1) \leftarrow$ true
11:                 $active'(p_2) \leftarrow$ true
12:                 reportImprovement($\{p_1, p_2\}$)
13:             $scheduled(p_1) \leftarrow$ false
14:             $scheduled(p_2) \leftarrow$ false
15:             findBlockPair($p_1$)        ▷ assume $participations(p_1) > participations(p_2)$
16:             findBlockPair($p_2$)
17:         $active(p_i) \leftarrow active'(p_i)$ $\forall p_i \in [k]$
18:         $active'(p_i) \leftarrow$ false $\forall p_i \in [k]$
19: end function
```

initializes some datastructures and will be explained in the next Section. For now, assume that it fills Q with conflict free pairs that we can start working on. The parallel processing itself works by consuming elements from Q in parallel and adding new elements to Q at the same time. Our C++ implementation uses TBB's `tbb::parallel_do_feeder` which enables these operations.

Once a thread gets an element $\{p_1, p_2\}$ from Q, it starts with the refinement of the pair. As the refiner uses local datastructures, we cannot use the same refiner for multiple pairs in parallel. Instead, each thread uses its own refiner $refiner_{\texttt{local}}$ that is exclusive to it. The implementation uses the `tbb::enumerable_thread_specific` class to achieve this. If the refinement improved the cut, we activate the blocks for the next round and report an improvement. The information that a certain block pair improved a cut will later be used to only schedule block pairs that were already refined in a previous iteration and improved the cut. To find new pairs to schedule, we also need to know whether a block is currently part of any pair in Q. The flag *scheduled* provides this information and is **true** if the block is currently taking part in an refinement or waiting for it and is contained in Q. After we finished the refinement, we can set it to **false**. Now that p_1 and p_2 are not scheduled any more, we can search for new block pairs that contain one or the other. We first try to find pairs that include the block with the higher number of participations, assume that this is p_1. Then we try to find pairs that include p_2.

A round finishes if there are no more elements left in Q. The *active*-flags for the next iteration *active'* are taken and *active'* is reset. Next, we will discuss how the functions responsible for the scheduling are implemented.

Scheduling

As explained earlier, the goal of the scheduling is to try to add blocks as part of a pair that have more participations in open pairs left. What we hope to achieve by adding blocks with the most participations is that at the end of an iteration when there are few open pairs left, we can still process them in parallel as they do not depend on the same blocks. However, this remains a heuristic and there is no guarantee that the chosen order is optimal.

Algorithm 4 Initial block pairs

```
1:  function initializeBlockPairs
2:      $participations(p_i) \leftarrow 0 \ \forall p_i \in [k]$
3:      $blockPairScheduled(\{p_1, p_2\}) \leftarrow$ false $\ \forall \{p_1, p_2\} \in [k] \times [k]$
4:      $blocks \leftarrow \emptyset$
5:      for $\{p_1, p_2\} \in [k] \times [k]$ do
6:          if isEligible($\{p_1, p_2\}$) then
7:              $participations(p_1) \leftarrow participations(p_1) + 1$
8:              $participations(p_2) \leftarrow participations(p_2) + 1$
9:      for $p_i \in [k]$ with $participations(p_i) > 0$ do
10:         $blocks$.add($p_i$)
11:     sort $blocks$ descending by number of participations
12:     for $p_i \in blocks$ do
13:         findBlockPair($p_i$)
14: end function

15: function isEligible($\{p_1, p_2\}$)
16:     return $|E^{\mathrm{cut}}(p_1, p_2)| > 0$ and ($active(p_1)$ or $active(p_2)$)
17:         and $\{p_1, p_2\}$ contributed to improvement or we are in the first iteration
18: end function
```

We use the following datastructures for the scheduling: $participations(p)$ of a a block p is the approximate number of refinements that p still has to undergo. When we compute the initial block pairs, we will calculate this value. $blockPairScheduled(\{p_1, p_2\})$ of a pair $\{p_1, p_2\}$ indicates whether the pair was already refined or is scheduled for refinement in this round. These pairs don't have to be considered for the scheduling of new pairs. The array *blocks* contains a sorted array of block ids. They are sorted in decreasing order of participations and the list only contains ids that have a positive number of participations left. The array will help us to scan for new blocks to schedule and has to be kept sorted when participations decrease.

As Algorithm 4 illustrates, to find the set of initial pairs, we first reset the datastructures and then look at every possible block pair. We define a pair to be eligible if the conditions described starting from line 15 hold. The size of the cut between the blocks has to be positive, at least one of the blocks has to be active (these two are the same conditions as for the sequential version) and the pair contributed to an improvement before or we are in the first round. For each eligible block pair, we increase the respective number of participations by one. Then, we add all ids with a positive number of participations to the array *blocks* and sort it descending by the number of participations. Next, we iterate through the ids, from the one with the most participations to the one with the lowest, and try to add block pairs containing the id.

Pseudocode for the search of new pairs is shown in Algorithm 5. Our goal is to find the id p_2 such that p_2 is the one with the highest number of open participations that is eligible and was not yet scheduled. The *blocks* array contains the ids sorted from highest participations to lowest, so it suffices to scan the list and check for admissibility. Once we found a pair, we schedule it by both setting the ids as scheduled and the block pair itself. When decreasing the number of open participations, we possibly break the sorting of *blocks*. To fix it, we swap the element that we changed with an element further back in the array. If participations decrease to 0, we remove the element from the array altogether.

Algorithm 5 Find block pair

```
1: function findBlockPair($p_1$)                                ▷ synchronized function
2:     for $p_2 \in blocks$ : not $scheduled(p_2)$ do
3:         if isEligible($\{p_1, p_2\}$) and not $blockPairScheduled(\{p_1, p_2\})$ then
4:             $scheduled(p_1) \leftarrow$ true , $scheduled(p_2) \leftarrow$ true
5:             $blockPairScheduled(\{p_1, p_2\}) \leftarrow$ true
6:             $participations(p_1) \leftarrow participations(p_1) - 1$
7:             move $p_1$ back in $blocks$ such that $blocks$ is sorted
8:             $participations(p_2) \leftarrow participations(p_2) - 1$
9:             move $p_2$ back in $blocks$ such that $blocks$ is sorted
10:            $Q$.add($\{p_1, p_2\}$)
11:            break
12: end function
```

Then we can add the pair to Q and stop the scanning process as we scheduled p_1 and there can be now other eligible pair containing it left. It is also possible that we find no admissible pair. In this case, the function returns without adding an element.

The function is not thread safe as it changes the array that it is iterating over. Furthermore, we would have to check and set the values for scheduled blocks and pairs atomically at the same time if other threads would execute the same function. The operation is rather fast as we only iterate over an array that has size at most k. In our experience it showed that in comparison to the more costly refinement operations, this is negligible. Therefore, the access is synchronized and only one thread at a time can execute it to schedule new pairs.

5. Snapshot extraction

For a block pair that undergoes the refinement procedure we do not use the complete subgraph induced by the vertices of the two participating blocks. Instead, we use an excerpt, the snapshot $\mathcal{H}$, that contains vertices close to the already existing cut. The aim is to reduce the runtime of the algorithm by making the flow instance smaller while trying to compute a cut that is as close as possible to the best cut between the whole blocks. We do so by starting a weight constrained breadth first search into each block, starting with vertices that are incident to cut-hyperedges. Each breadth first search visits vertices in its block that are later added to the hypergraph $\mathcal{H}$, remaining vertices in the block are contracted to s for one side and to t for the other. Vertices are therefore visited in order of increasing hop-distance to the nodes incident to the cut. When the BFS visits a new vertex, we only add it to $\mathcal{H}$ if it does not exceed the maximum weight of the respective side.

The algorithm can be divided into two phases. First, we identify the vertices that should be added to $\mathcal{H}$ using weight constrained breadth first searches (BFS) into both blocks. The two searches are independent from one another as they visit vertices in different blocks and are therefore executed in parallel. The algorithm for one BFS is explained in Section 5.1. To initialize the BFS in this phase, we scan the cut hyperedges and add vertices to the initial queues of both searches. After phase 1, we know which vertices have to be added to $\mathcal{H}$ and we build the hypergraph in phase 2. How this is done and some explanations for the datastructures we use are given in Section 5.2.

5.1. Parallel BFS

A classical and conceptually simple way to parallelize a BFS [QD84, RC78, BM11] for graphs is to add all nodes that have been visited to a queue Q and process them in parallel by searching for neighbors that were not yet visited. The newly found nodes are added to a second queue Q' using atomic operations. Once all nodes in Q were processed by iterating over their incidences in parallel, the threads are synchronized, Q and Q' are swapped so that Q now contains the nodes that have to be processed next and Q' is cleared. To start the BFS, we initialize Q either with a single node or a set of nodes that we want to search from. Each iteration consisting of processing all nodes in Q represents a layer of the BFS, meaning the nodes have the same hop-distance to the initial nodes in Q.

For our implementation we use the same general approach and extend it to hypergraphs. In a graph, we can scan an edge at most twice during one iteration iff both endpoints are part of the same layer and no nodes have to be added to the queue as both endpoints are already in the queue. In a hypergraph this changes and multiple vertices can be incident to the same hyperedge with pins that still have to be added to the queue in the same iteration. It is not necessary to process this hyperedge multiple times as all threads will try to add the same pins. This holds for all iterations, once we scanned a hyperedge, we do not have to do it again. Therefore we add a datastructure that tells us if a hyperedge has already been processed and use atomic TESTANDSET operations so that only one thread scans it. Other changes are attributed to the fact that we execute a weight constrained BFS and have to stop visiting new vertices once a certain maximum weight is exceeded. To be more clear, in our notation, we say that we *visit* a vertex when it is discovered by some thread and can be added to Q' without violating the weight constraint. At the end of the BFS, all nodes that were visited are part of $\mathcal{H}$. The function TRYVISITVERTEX uses synchronization primitives and checks if it can visit the vertex and does so while respecting the maximum weight restriction. We will go into more detail on this function later.

Algorithm 6 Weight-constrained BFS

```
1: function WEIGHTCONSTRAINEDBFS
2:     while $Q \neq \emptyset$ do
3:         parallel for $u \in Q$ do                          ▷ process layer in parallel
4:             for $e \in I(u)$ do
5:                 if $visitedHyperedges(e)$.TESTANDSET() then
6:                     for $v \in e$ with $P(v) = p$ do
7:                         TRYVISITVERTEX($u, p$)                ▷ adds vertices to $Q'$
8:         $Q \leftarrow Q', Q' \leftarrow \emptyset$
9: end function
```

Pseudocode 6 depicts the parallel BFS algorithm. We process every layer Q in parallel and add vertices we visited to the next layer Q' that will be processed in the next iteration. To process a vertex, we scan all incident hyperedges e and try to set *visitedHyperedges* of e atomically. If we succeed, we iterate over all pins and for those that are in the block that we process we try to visit them by calling TRYVISITVERTEX. What this function will do is to try to visit the vertex and add it to Q' if it is the first one to do so. Instead of adding the vertices to one single queue from multiple threads using atomics, we use thread-local queues that are implemented using TBB's `enumerable_thread_specific` object. Every thread has its own queue that it adds vertices to, thus no synchronization or atomic operations are necessary. When iterating over Q in parallel, we iterate over all thread-local queues and the vertices therein in parallel.

5.1.1. Visiting vertices

The TRYVISITVERTEX function encapsulates all synchronization primitives and accesses to the $visitedVertex$ datastructure, which is used to keep track of nodes that were visited, and the current weight w_{p_i} of vertices that were visited in block p_i. No other function changes these values. When visiting a vertex u, we have to make sure that we change w_{p_i} and $visitedVertex(u)$ consistently, meaning that we only add the weight of the node to w_{p_i} if we are able to set $visitedVertex(u)$.

First, we check if we actually have to try to visit a vertex u or if it was already visited or the weight limit of the block is already exceeded. Otherwise, we try to visit the vertex by setting $visitedVertex(u)$. We lock u on a per vertex basis for the rest of this operation to avoid conflicts. With the previous checks in place, we avoid to acquire the lock in some cases where it is not necessary. It is possible that another thread visited u before we were

Algorithm 7 To to visit a vertex

```
1:  function tryVisitVertex($u$, $p_i$)
2:      if $w_{p_i} + \varphi(u) > w^{\max}$ then
3:          return
4:      if $visitedVertex(u)$ then
5:          return
6:      lock($u$)
7:      if $w_{p_i} + \varphi(u) \leq w^{max}$ and not $visitedVertex(u)$ then
8:          $w_{p_i} \leftarrow w_{p_i} +_a \varphi(u)$                ▷ atomic add
9:          $visitedVertex(u) \leftarrow$ true
10:         $Q'$.push($u$)
11:     unlock($u$)
12: end function
```

able to acquire the lock, so we have to check after locking u if we are still the first thread that can then visit u. The problem with setting w_{p_i} is that even when we add to the value atomically, we don't know the value that we add to. We can get the value afterwards using an atomic fetch and add, but only after we already added to it and possibly exceeded the maximum weight.

To make the synchronization easier, we allow a small error for the total block weight here: We use an atomic add in line 8 that possibly exceeds the maximum weight but we still set u to visited. By doing so, each thread can visit at most one vertex that is exceeding the limit. If we are the thread that visits u, we not only increase w_{p_i} atomically, but also push u to our local queue so it can be processed by the BFS in the next iteration. We don't have to use atomic operations to set $visitedNode(u)$ because we already hold a lock for u.

5.1.2. Scanning cut hyperedges

To initialize the two BFS queues with vertices that are incident to cut hyperedges, we scan all cut hyperedges and check which pins are part of one of the blocks. We try to visit them and add them to the first layer of the queue.

Algorithm 8 Scan cut hyperedges

```
1: function scanCutHyperedges($E^{\text{cut}}, p_1, p_2$)
2:     parallel for $e \in E^{\text{cut}}$ do
3:         for $u \in e$ do
4:             if $P(u) = p_1$ then                ▷ process pin of block $p_1$
5:                 visitVertex($u, p_1$)
6:             else if $P(u) = p_2$ then           ▷ process pin of block $p_2$
7:                 tryVisitVertex($u, p_2$)
8:         $visitedHyperedges(e) \leftarrow$ true
9: end function
```

Algorithm 8 shows pseudocode for this operation. We process each cut hyperedge e in parallel and try to visit pins u of e using the same tryVisitVertex function we introduced for the BFS. After we scanned all cut hyperedges, the queues for the two breadth first searches are initialized with nodes on the respective side that are incident to the cut.

5.2. Building the hypergraph

After we have identified the vertices that should be included in the snapshot $\mathcal{H}$, we proceed with building the extracted hypergraph. The visited vertices were kept in thread-local vectors during phase 1 that are distributed among threads. After the flow computation on $\mathcal{H}$ is finished, we want to move vertices to other blocks in H to improve the cut. Therefore, we compute a mapping from node-ids in $\mathcal{H}$ to ids in H. After the mapping is established, we use a list of all hyperedges that we visited during the BFS to add them to $\mathcal{H}$. To parallelize this operation, we calculate a prefix-sum over all hyperedge sizes in $\mathcal{H}$ to precompute indices where we have to write pin-data to. We go into more detail on this after we explained the mapping and the Compressed Sparse Row (CSR) representation we use for $\mathcal{H}$.

5.2.1. Vertex id mapping

As we explained earlier, the BFS adds one BFS-layer with vertices of the same distance to the initial vertex set per iteration to the queue. This means that in every thread-local vector, the vertices are ordered layer by layer, assuming that we don't delete vertices of a previous layer and only append new vertices to the vector. Two of these thread-local vectors, named queues here, are depicted in Figure 5.2.1. Q_1 and Q_2 are thread-local queues and contain the vertex-ids (in H) of vertices that they visited during the BFS. In this example, there are four layers for each queue, indexed by L_1^i for Q_1 and L_2^i for Q_2. The first layers are the ones that were added when scanning the cut hyperedges, the following layers were added during three iterations of the BFS. We did not mention that we need these layers in the previous iterations, but the only additional information that we need are the bounds in the local vectors of each thread. These bounds can be stored at the end of each iteration of the BFS. For the mapping from ids in $\mathcal{H}$ to ids in H we create the vector *map* that is depicted on the right. The vector is indexed by the new vertex ids in $\mathcal{H}$ and contains the id in H. We know copy vertex ids from the queues to *map*.

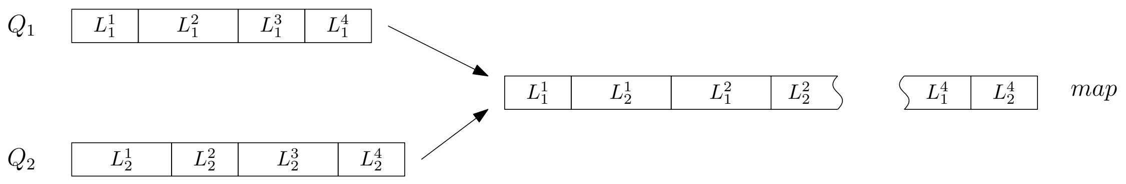

Figure 5.1.: On the left: two thread local queues with four layers each, on the right: the array *map* used for the id mapping

We could do this in any order, but we add the ids to *map* in a way that ids of vertices that were visited in the same layer are close to each other in *map*. The idea is that when hyperedges are visited, we try to visit pins and add them to the same layer. If the layers stay close to each other in *map*, then the pin ids of all hyperedges should be close together too. This can improve cache efficiency for the following flow computation that scans pins of a hyperedge and accesses datastructures that are indexed by their vertex ids. This is why we chose to take the first layers of all queues, then all the second layers and so on to fill *map*. This can be done for any number of queues and not only two like in the example.

To establish the mapping, we first compute the starting and end indices that we want to write a layer L_i^j to. This is done sequentially by iterating over the layer bounds of the queues and calculating their positions in *map*. Then, we process every layer of a queue L_i^j in parallel and copy its ids to the precomputed range. While we are writing vertex ids id_{global} of H to indices id_{local} that are the ids of the snapshot, we also write the snapshot ids into a second mapping vector *globalToLocal* to store the snapshot ids. It then holds that $globalToLocal(id_{\text{global}}) = id_{\text{local}}$. Both mappings are needed, one in the next step to obtain ids of the snapshot, the other later on when we finished the flow computation and write the new partition ids to the original graph H. For the special vertices s and t that are not part of H, we reserve special ids.

The whole operation is done for both sets of queues of the two breath first searches separately. After we established the mapping *map* for the first BFS, we add the second set of vertex ids by writing them behind the first ones in the *map*-vector.

5.2.2. CSR representation of the hypergraph

Before we can continue to actually build the extracted hypergraph, we need to know how it is represented. We use a compressed sparse row representation (CSR) of hyperedges and their incident pins. This representation consists of two vectors: `hyperedges` and `pins`. Both are depicted in Figure 5.2.2.

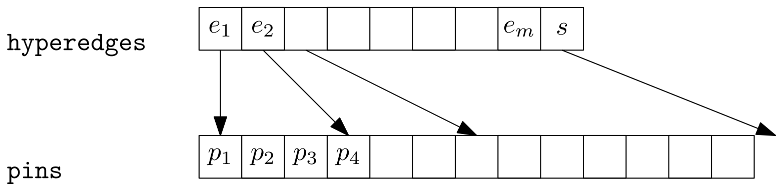

Figure 5.2.: The hyperedge array stores a pointer to the first pin of a hyperedge. s is a sentinel that points behind the last pin. In this example, e_1 has pins p_1, p_2 and p_3

Each hyperedge e_i stores an index of the first pin belonging to e_i in the `pins`-vector, which contains data for pins p_i (for example the vertex id). The last hyperedge that we store is a sentinel that points behind the last pin. Pins of hyperedge e_i are therefore stored at positions `hyperedges`$[i]$ to `hyperedges`$[i+1]-1$ in the `pins`-vector. Going forward, we first construct the `hyperedges`-vector using a prefix sum and then write the pin-data to `pins`. The datastructure is slightly simplified but contains everything we need to represent the hypergraph. Elements that are missing are for example datastructures that store the flow values. A more complete view of the hypergraph is given in Chapter 8. No meaningful changes of the algorithm have to be made to adapt it to the more complete datastructure.

5.2.3. Adding hyperedges

From the id mapping, we already got the vertex ids of $\mathcal{H}$ that we want to use. To build the CSR-hypergraph we now process hyperedges of H that contain pins in one of the blocks and add counterparts in $\mathcal{H}$. A counterpart of a hyperedge e in H has to use the ids of $\mathcal{H}$. Pins of e that were not visited are contracted to either s or t, depending on the block that the pin is part of in H. We implement this with a sorted vector of hyperedges and use a parallel prefix-sum to compute the `hyperedges`-vector. Then, we iterate over the hyperedges and add the pins to the `pins`-vector, using the mapping we presented before.

During the BFS and the scan of cut hyperedges, we can already identify all hyperedges that have to be added to $\mathcal{H}$ as we scan all incidences of pins we visit. We keep a vector E^+ of tuples, each representing a hyperedge. Such a tuple $(e_{\text{id}}, n_{\text{pins}})$ stores the hyperedge-id e_{id} and the number of pins n_{pins} that the hyperedge will have in $\mathcal{H}$. For every scan of a hyperedge that we execute during the search, we consider adding a tuple to E^+. To compute n_{pins} while iterating over the pins of a hyperedge, we need to check for each pin if it is part of $\mathcal{H}$. This property is fixed after we try to visit a pin: If we failed visiting a pin, then no other thread can succeed after us. We count the pins that were visited and check for the other if they have to be contracted to s or t. Hyperedges of the cut can contain different pins that have to be contracted to s and others that are contracted to t. The hyperedge in $\mathcal{H}$ is therefore connected to s and t and will always be part of the cut. We skip these hyperedges and don't add them to E^+. The weight of dropped hyperedges is added up and we can add it to the value of the cut in $\mathcal{H}$ to get the cut-value in H. This way, we can drop hyperedges that are not necessary for the flow problem. Altogether, we can calculate n_{pins} with few additional operations. To avoid synchronization or atomic operations that would occur when all threads add to the same vector, we use thread-local vectors of E^+ so that every thread can add to its own versions. After the BFS is finished, they are concatenated into one vector E^+.

We then sort E^+ by the hyperedge-ids. This makes the order of hyperedges in $\mathcal{H}$ more consistent between multiple parallel runs. If we would always visit the same vertices during the BFS, this would lead to a completely deterministic hypergraph $\mathcal{H}$. This is not the case as the order in which threads try to visit vertices is not necessarily the same and depends on the order vertices in the queue are processed. At least, this leads to results of $\mathcal{H}$ that are a bit more similar. An advantage that comes with the sorting is that following scans of the vector that access the hyperedge by its id may become more cache efficient because the hyperedge ids of successive elements in E^+ are closer together. The sorting is executed in parallel using the build-in TBB-function.

The reason why we store the number of pins for each hyperedge is that we can now compute the `hyperedges`-vector by calculation the exclusive prefix sum over the n_{pins} values in E^+. Parallel prefix sum is a well studied problem [ST06] and there exists a parallel implementation in TBB that we can make use of. Now that `hyperedges` is filled, we only have to fill `pins`. Because we already know the range where pins have to be copied to for every e_i, we iterate in parallel over all hyperedges with id e_i and translate the id of pins that were visited from those in H to the ones in $\mathcal{H}$ using the mapping we established before. No additional synchronization or atomic operations are needed for this step as for the processing of every hyperedge, we work on an exclusive part of the `pins`-vector.

6. Dinic's max-flow algorithm

The algorithm that we discuss here is based on the sequential implementation of the Dinic algorithm in WHFC [GHSW20]. The two most time consuming steps of the algorithm are the computation of the layered network and the augmentation of the flow in the network afterwards. To build the layered network a breadth-first-search starting from a set of vertices V_s will be done, while the search for augmenting paths is done by a depth-first-search. Our goal was to parallelize these steps for better running times. We will start with the parallelization of the routine that builds the layered network. When we tried to parallelize the finding of augmenting paths, several challenges emerged and we did not achieve to find an implementation that can improve upon the sequential performance. We will discuss these issues and proceed with the parallelization of another flow algorithm in the next Chapter.

6.1. Building the layered network

The layered network is build using a BFS and we use a similar approach as in Section 5.1. We do not store the layered explicitely as a hypergraph, but use three datastructures: $d_{\text{vertex}}, d_{\text{in}}$ and d_{out}. They are enough to be able to decide whether a vertex can be reached from another in the layered network when we traverse the hypergraph to find augmenting paths. The main difference to the BFS mentioned before is the fact that we search in the residual network and when visiting a vertex, we can not always reach all pins of a hyperedge but possibly only pins sending flow into the hyperedge. To solve this problem, we use two datastructures d_{in} and d_{out} for hyperedges that are set to the current hop-distance of the search when we can reach all flow sending pins or all pins respectively. For vertices, one hop-distance d_{vertex} to reach the pin suffices. Atomic operation are used to set the hop-distances and the first thread that does so scans either incidences of the vertex if it set d_{vertex} or pins of a hyperedge if one of the other datastructures was set.

Let us go into some detail on when we have to set the hyperedge hop-distances now. Once we reach an edge from a pin u, it depends on u whether we are also able to reach pins that are not sending flow. If u is receiving flow, then this flow can be pushed back to any other pin of e, making them all reachable in the residual network, even if the residual capacity of e is zero. If u is not receiving flow and there is no residual capacity left for e, then we can only push flow to pins that are sending flow into e, as this flow can be pushed back to them. When we are searching for augmenting paths in the second stage of the Dinic-algorithm, we will need the hop-distances of hyperedges to decide which neighboring vertices can be reached by exactly one hop and are thus part of the next layer. So, to keep track of these values, $d_{\text{in}}(e)$ of a hyperedge e is set to the distance from s that is needed to reach pins that send flow into e and $d_{\text{out}}(e)$ is the distance with which we can reach all pins of e. It always holds that $d_{\text{in}}(e) \leq d_{\text{out}}(e)$.

Algorithm 9 Build layered network

```
1:  function buildLayeredNetwork($\mathcal{H}$)
2:      $L \leftarrow \{s\},\ L' \leftarrow \emptyset$
3:      $d_{\text{in}}(e) \leftarrow 0, d_{\text{out}}(e) \leftarrow 0\ \forall e \in E$
4:      $d_{\text{vertex}}(u) \leftarrow 0\ \forall u \in V$
5:      $d \leftarrow 1$
6:      while $L \neq \emptyset$ do
7:          parallel for $u \in L, e \in I(u)$ do
8:              if $d_{\text{out}}(e) = 0$ then
9:                  $scanAll \leftarrow r_f(e) \neq 0$ or $f(e_{\text{out}}, u) > 0$
10:                 if not $scanAll$ and $d_{\text{in}}(e) > 0$ then
11:                     continue
12:                 $scanAll \leftarrow scanAll$ and $d_{\text{out}}(e)$.COMPARE-AND-SWAP$(d) = 0$
13:                 $scanSending \leftarrow d_{\text{in}}(e) = 0$ and $d_{\text{in}}(e)$.COMPARE-AND-SWAP$(d) = 0$

14:                 if $scanSending$ then
15:                     for $v \in e$ with $f(v, e_{\text{in}}) > 0$ and $v \neq t, d_{\text{vertex}}(v) = 0$ do
16:                         if $d_{\text{vertex}}(v)$.COMPARE-AND-SWAP$(d) = 0$ then
17:                             $L'_{\text{local}}$.ADD$(v)$
18:                 if $scanAll$ then
19:                     for $v \in e$ with $f(v, e_{\text{in}}) = 0$ and $v \neq t, d_{\text{vertex}}(v) = 0$ do
20:                         if $d_{\text{vertex}}(v)$.COMPARE-AND-SWAP$(d) = 0$ then
21:                             $L'_{\text{local}}$.ADD$(v)$
22:         $d \leftarrow d + 1$
23:         $L \leftarrow L',\ L' \leftarrow \emptyset$
24: end function
```

Algorithm 9 shows the pseudocode for BFS. For the notation of the algorithm we use the flow values in the corresponding Lawler network, meaning that $f(u, e_{\text{in}})$ is the flow a vertex u sends into a hyperedge e and $f(e_{\text{out}}, u)$ is the amount of flow that u receives. If $f(u, e_{\text{in}}) = 0$ then u can still possibly receive flow from e. However, the algorithm is executed directly on the hypergraph and this is purely for notation. The synchronization of the parallel BFS is mainly done with atomic operations [QD84, RC78]. Additionally, for the current and next layer L and L' of the search, we use thread-local vectors that are implemented using TBB's `tbb::enumerable_thread_specific`. To iterate over the whole set of L consisting of all elements contained in local vectors, we iterate in parallel both over the vectors and their respective elements.

The algorithm uses the current layer L and the next layer L' in an alternating manner. We iterate over the vertices in L in parallel and add new vertices that we encounter to L'. After one layer of the BFS is processed, we swap L and L', making the next layer the current one and we reset L'. For every vertex $u \in L$, we iterate over all incident hyperedges e in parallel. We have to check whether we can already reach all pins by comparing $d_{\text{out}}(e)$ to 0. If not, we want to reach e. We can reach all pins if e is not saturated or u receives flow from e.

At this point we know that not all pins of e were reachable in the previous iteration but there could still be other threads that want to reach e coming from another pin. We resolve this issue by letting the thread scan the flow sending pins if it was able to atomically compare-and-swap the current running distance d with $d_{\text{in}}(e)$ and gets 0. The thread that swaps d with $d_{\text{out}}(e)$ and gets 0 is allowed to scan all non flow sending pins. Only one thread can succeed at this operation and proceeds. The variables *scanAll* and *scanSending* are set to **true** if the thread can scan non flow sending or scan sending pins respectively. When we do this atomic compare-and-swap for *scanSending*, we should only try to write d into the function if was 0 before. Otherwise we could overwrite the previous lower value stored in $d_{\text{in}}(e)$. Pins v are processed by checking if the pin could already be reached before and atomically exchanging the current running distance d with $d_{\text{vertex}}(v)$. Then, v is added to the thread-local vector L'_{local} and will be processed in the next iteration of the BFS.

6.2. Finding augmenting paths

It turned out to be rather difficult to parallelize the depth first search that the Dinic algorithm employs to find augmenting paths. We found no practicable way to do this and want to give some reasons and issues that we ran into.

Before we start, let us recapitulate how the sequential algorithm of this part works in the existing implementation of WHFC [GHSW20]. The algorithm begins with a depth first search at s and considers the residual network of the hypergraph when searching for neighboring vertices that are reachable. Instead of using the Lawler expansion, the algorithm works directly on the hypergraph representation. The DFS is implemented using an explicit stack that contains the vertices that are on our current path and the ids of hyperedges that are used to go from one vertex to the next on the stack. When we reach t, the path that the stack represents is an augmenting path from s to t. At this point we only know that it is possible to route some flow from s to t, but not how much exactly. To get this information, we scan the stack once and compute the minimal residual capacity $c_{\min}$ of any hyperedge on it. Let the hyperedge with minimum capacity be e (if there are multiple ones with the same minimum capacity then take the one closest to s on the path) and the vertex that can route flow into the hyperedge u_1 and the vertex that receives flow u_2. We know that $c_{\min}$ is the maximum amount of flow that we can route through the augmenting path and that after the augmentation by $c_{\min}$, there is still residual capacity left on the path from s to u_1. This is why after augmenting the path by $c_{\min}$, we can pop vertices from the top of the stack until u_1 is on top and resume the DFS from here. The procedure is repeated until we cannot find any augmenting path any more and pop s from the stack. An optimization that proved to be crucial for the implementation of WHFC is keeping pointers for the next hyperedge that we have to check when visiting a vertex. Because we search for multiple augmenting paths and change the residual network after we found one, we can also visit a vertex more than once. To let an example guide us, consider the case when we found an augmenting path and popped the stack down to u_1. We have to resume the search from here and possibly scanned some hyperedges that lead to no augmenting path until we found one that did. As we only reduced residual capacities when augmenting the path, we know that these hyperedges that lead to no augmenting path cannot lead to one now, even if the network changed. So there is no point in scanning them again and searching for pins that we can route flow to. Instead, we keep a pointer to the last hyperedge we scanned and advance it when we scan the next one. This requires that the incident hyperedges of a vertex are always accessed in the same order. The same optimization can be done for the pins of each hyperedge.

With this in mind, let us discuss the problems we faced. In theory, we can can parallelize the DFS by splitting the search when we visit a vertex, for example at s. For all vertices that we can reach from s, a new thread can be started that continues the search for this node. As soon as one thread finds a path to t, we run into the first problems because we need to change the flow and the residual network that other threads are currently working on. This means that we cannot simply resume all parallel searches that we started. The thread that found the path can backtrack to the lowest bottleneck vertex on its stack and continue from there, but other threads might use hyperedges on their current path that do not have capacity left any more. It would be viable that every processor gets notified in the event of an augmentation and checks if any hyperedges on its path became exhausted, but this results in more synchronization being necessary and additional work. Another approach to address this problem is to lock hyperedges for threads and continue with the scan of another hyperedge if we encounter a vertex or hyperedge that is already locked by another thread. It remains the question when we scan the previously locked hyperedge after the other thread released it. If we wait on a locked hyperedge, we don't have to manage this at a later point but hinder the parallelization by waiting for resources.

In addition to these problems, we also have to handle the datastructures that point to the next hyperedge or pin that we should check. If multiple threads use the same vertex, they are working on the same datastructure, leading to race conditions or locking being necessary. A simple solution is to replicate the datastructures, but that increases the need for memory and the amount of total work increases as threads can not make use of the information that another thread could not find an augmenting path when following a certain hyperedge.

These are only some ideas, why the DFS proved to be hard to handle for parallel computation. In a way, parallelizing the DFS to find only one augmenting path is possible, but finding a series of augmenting paths in a changing network is more complicated. Each augmenting path that we find is dependent on the one we found before, because the network changes according to the paths. One reason the sequential implementation is fast lies in the datastructures that enable us to skip the scan of hyperedges and pins that cannot lead to an augmenting path. These are hard to manage in the parallel case and not using them would mean sacrificing some of the single thread performance that the parallel algorithm has to compensate for. To have a parallelized flow algorithm for the WHFC framework, we tried a well known flow algorithm that is conceptually easier to parallelize. Next, we will present the Push-Relabel algorithm together with a parallel implementation.

7. Push-Relabel

The reason why we ventured into the parallel implementation of another flow algorithm are the problems we faced when trying to parallelize Dinic and its routine to find augmenting paths. We needed a max-flow algorithm that lends itself to parallelization more easily. One such algorithm is Goldberg and Tarjans Push-Relabel algorithm [GT88, BBS15] that uses preflows and local push operations to propagate flow in the network. The main procedure used by the algorithm is the discharge-operation for nodes that pushes excess flow to neighboring nodes. This local operation can be executed on several nodes in parallel without too much synchronization. The following Sections will first explain the basics of the Push-Relabel algorithm for graphs. Then, the sequential version for hypergraphs will be discussed, followed by the changes made to run it in parallel.

7.1. Push-Relabel for graphs

In this Section, we will explain the existing algorithms and optimizations of the Push-Relabel algorithm for graphs. The basic notions the algorithm [GT88] uses are those of excess and labels. In contrast to other maximum flow algorithms like Dinic, every node can have an excess of flow, which is surplus flow that goes into the node, but does not leave it. The excess function e can be formalized as

$$e(u) := \sum_{(v,u)\in E} f(v,u) - \sum_{(u,v)\in E} f(u,v).$$

Because nodes can have excess flow during the execution of the algorithm, the flow function f does not always represent a flow, but a preflow. The label function d maps nodes to a label. This label is used to determine if we allow flow to be sent from one node to a neighbor. In the context of this algorithm the operation of sending flow from one node to another is also called pushing flow. To be able to push flow from u to a neighbor v, it has to hold that $d(u) = d(v) + 1$. The algorithm relabels nodes by increasing the label if excess flow can not be pushed to any neighbor. To be called *valid*, the label function has to fulfill the following invariants: $d(s) = n$, $d(t) = 0$ and $d(u) \leq d(v) + 1$ for every edge (u, v) with $r_f(u, v) > 0$. For the Push-Relabel algorithm, the invariant holds that the label function is valid at any time during the execution.

Pseudocode for the algorithm is given in Algorithm 10. We start by initializing the (valid) label function $d(u) = 0\ \forall u \in V \setminus \{s\}$ and $d(s) = n$. Flow on all edges is set to 0 and we start with the first series of push-operations. Flow is pushed to all neighbors of s such

Algorithm 10 FIFO-Push-Relabel

Input: Graph G, source s, sink t, flow function f

Output: f that is a maximum flow from s to t

```
 1: function maxFlowPushRelabel($G = (V, E)$, $s$, $t$, $f$)
 2:     for all $u \in V \setminus \{s\}$ do                ▷ initialize datastructures
 3:         $d(u) \leftarrow 0$
 4:     $d(s) \leftarrow n$

 5:     for all $(u, v) \in E$ do
 6:         $f(u, v) \leftarrow 0,\ f(v, u) \leftarrow 0$

 7:     for $(s, u) \in E$ do                              ▷ push flow to adjacent nodes
 8:         $f(s, u) \leftarrow c(s, u),\ f(u, s) \leftarrow -c(s, u)$
 9:         $Q$.push($u$)

10:     while $Q \neq \emptyset$ do
11:         $u \leftarrow Q$.pop()
12:         $d_{\min} \leftarrow n$
13:         for $(u, v) \in E$ with $r_f(u, v) > 0$ do
14:             if $d(u) = d(v) + 1$ then
15:                 $r \leftarrow min\{e(u), r_f(u, v)\}$
16:                 $f(u, v) \leftarrow f(u, v) + r,\ f(v, u) \leftarrow f(v, u) - r$
17:                 if $Q$ does not contain $v$ then $Q$.push($v$) end if
18:             else
19:                 $d_{\min} \leftarrow min\{d_{\min}, d(v)\}$
20:         if $e(u) > 0$ then                             ▷ relabel
21:             $d(u) \leftarrow d_{\min} + 1$
22:             $Q$.push($u$)
23: end function
```

that the incident edges are completely exhausted. The main loop repeatedly processes (*discharges*) nodes with positive excess. The nodes with flow excess are called *active* nodes and have to be kept track of. The general Push-Relabel algorithm does not specify an order of the discharge-operations. Indeed, there are multiple versions of the algorithm using different approaches to find the next active node that is discharged. The order of these operations can have an impact on the performance of the algorithm. We will focus on an implementation using a FIFO-queue Q to manage active nodes as this version can be emulated in parallel by processing all nodes of the queue at once and add new nodes for the next parallel step. The neighbours of s are the first nodes that become active and they are therefore added to Q.

The main loop of the algorithm then pops nodes u from Q and discharges them, thereby adding nodes to Q that become active, or add u again if $e(u) > 0$ after the discharge. The discharge operation looks at all incident residual edges (u, v) and tries to push excess flow to v. As mentioned before, a push is only possible if $d(u) = d(v) + 1$. After we pushed flow to v, we check if v becomes active and add it to the end of the queue if this is the case. During the discharge, we keep the lowest label $d_{\min}$ of a neighbor connected by a residual edge that we were not allowed to push to. If u still has excess flow at the end of the discharge, we use $d_{\min}$ to relabel u to $d_{\min} + 1$. The relabeling keeps the label function valid, an important invariant of the algorithm.

Instead of only relabeling single nodes, a usual heuristic is the insertion of a global relabeling step after a certain number of discharges [CG95]. The global relabeling function relabels all nodes at once by setting the labels to the shortest distance (in hops) to t. This relabeling is meant to guide the push operations more towards the sink, where we want the excess flow to go to. The gap relabeling heuristic [Che94b, DM89] is another optimization that uses the fact that if there is a value g (the gap) such that there are no nodes with label g, then all nodes u with $g < d(u) < n$ can not reach t and can be relabeled to n. Especially the global relabeling heuristic has been shown to improve the runtime of the algorithm considerably [CG95, HSS19].

Another improvement can be achieved by splitting the algorithm into two phases as proposed by [CG95]. In phase 1 we apply the same discharge operations on nodes in the queue, but add a stopping criterion that will let the algorithm finish phase 1 before the complete max-flow was computed. Instead, phase 1 will output a maximum preflow. In phase 2, we then use a standard flow decomposition [GTT89] to transform this preflow into the actual maximum flow. In the experience of the authors, the second phase takes considerably less time than the first.

7.2. Sequential Push-Relabel implementation

In our case, we want to apply the Push-Relabel algorithm to a hypergraph. To do so, we implicitly work on the Lawler expansion of the hypergraph and introduce additional labels and excess values for hyperedges. For now, we assume that we have labels and excess values for all nodes of the Lawler graph and thereby emulate the Lawler network without explicitly constructing it. This has the advantage that we can continue using datastructures that are close to the ones of the WHFC implementation which makes it easier to fit into the framework. The main disadvantage is that we have to handle every type of node differently in our implementation depending if it is an edge-in, edge-out or normal node of the Lawler network. Furthermore, the question is justified whether an explicit representation of the Lawler network may be more efficient. We argue that the performance penalty of our implementation is not too high because accesses of incidences are still linear in the number of incident vertices and we just translate them to relations in the hypergraph. To get the adjacent nodes of an edge-in node for example, we iterate over the pins of the corresponding hyperedge and add the edge-out node.

7.2.1. Overview of the max-flow algorithm

For the implementation of the sequential algorithm, we split the algorithm into two phases: the first computes a maximum preflow and the second one uses the principles of the flow decomposition to transform the preflow into a maximum flow. We use this approach as it was shown to be faster than computing the maximum flow directly (even in the parallel case) [HSS19]. In addition, we employ the global relabeling technique.

Algorithm 11 shows an overview of the sequential Push-Relabel max-flow implementation. The main datastructure we use is the FIFO-queue storing Lawler nodes that have to be discharged. The implementation uses an extended range of node-ids that make it possible to easily deduce what kind of Lawler node it is and in the case of an edge-in or edge-out node, what hyperedge corresponds to it. We use the following mapping for the ids of the hypergraph $H = (V, E)$:

- For regular nodes: the same id as the one used in the hypergraph
- For edge-in nodes e_{in}: $|V|+$ edge-id of e
- For edge-out nodes: e_{out}: $|V| + |E|+$ edge-id of e

Algorithm 11 Push-Relabel

```
Input: Hypergraph $H = (V, E)$, source $s$, sink $t$, flow function $f$
Output: $f$ that is a maximum flow from $s$ to $t$
 1: function maxFlowPushRelabel($H = (V, E)$, $s$, $t$, $f$)
 2:     for all $u \in V$ do                                ▷ reset datastructures
 3:         $e(u) = 0$

 4:     for $e \in I(s)$ do                          ▷ push flow to in- and out-nodes
 5:         for $v \in \{e_{\text{in}}, e_{\text{out}}\}$ do
 6:             if $r_f(u, v) > 0$ then
 7:                 Push flow $r_f(u, v)$ from $u$ to $v$

 8:     globalRelabel($H$, $s$, $t$, $f$)                       ▷ section 7.2.3
 9:     maxPreflow($H$, $s$, $t$, $f$)                          ▷ section 7.2.2
10:     flowDecomposition($H$, $s$, $t$, $f$)                   ▷ section 7.2.4
11: end function
```

This leads to the three intervals $(0, |V|]$ for regular nodes, $(|V|, |V| + |E|]$ for edge-in nodes and $(|V| + |E|, |V| + 2 \cdot |E|]$ for edge-out nodes. It is therefore not only easy to determine the type of a Lawler node given the id, but also computing the accompanying edge id for edge-in and edge-out nodes or vice versa. All these operations can be implemented using basic arithmetic calculations.

At the start of the algorithm, node excesses are reset. Then, as usual for Push-Relabel flow computations, flow from the source is pushed to all adjacent nodes. Because s is a regular node of the hypergraph and is therefore represented by a regular node in the Lawler network, we only push flow to edge-in and edge-out nodes of the Lawler network. Pushing to an edge-out node e_{out} is possible if there is already flow running from e_{out} to s. In our case this flow from e_{out} to s can exist because we can start the computation on a hypergraph with preexisting flow. This can happen as we have to solve a sequence of flow problems for the FlowCutter framework. We summarize the adaptions that are necessary for the integration in FlowCutter later, this is one of the them.

The labels are initialized by running the global relabeling function. This will also fill the queue with nodes that are discovered by the BFS and have positive excess. Then, we compute the max-preflow by discharging active nodes and interleaving global relabeling steps. After there are no more active nodes left, the flow decomposition transforms the max-preflow into a max-flow.

7.2.2. Max preflow

Because we are using the optimization that splits the algorithm into two phases, we only consider nodes u with $e(u) > 0$ and $d(u) < n$ here. The reason behind this is that flow that is pushed from a node with label higher or equal to n cannot reach t. This is because flow is always pushed down exactly one label and we would need to push flow along a path from u to t in the graph that has length n as $d(t) = 0$. This only works for $u = s$ but we never discharge s in the main loop.

In between discharge operations, we insert the global relabeling steps. The frequency is chosen according to a heuristic used in the implementation of Cherkassky and Goldberg [CG95]. It is based on the constant, predefined values α, β and the global update frequency $freq_{\text{u}}$. Furthermore, the variable *workSinceLastGlobalRelabel* tracks work that was done since the last global relabeling step. Each time a node is relabeled, the number

Algorithm 12 Max preflow

```
Input: Hypergraph $H = (V, E)$, source $s$, sink $t$, flow function $f$
Output: Extend $f$ to a maximum preflow
 1: function MAX PREFLOW($H = (V, E)$, $s$, $t$, $f$)
 2:     while $Q \neq \emptyset$ do
 3:         $u \leftarrow Q$.POP()
 4:         DISCHARGE($u$)
 5:         if $e(u) > 0$ and $d(u) < n$ then
 6:             $Q$.PUSH($u$)
 7:         if $workSinceLastGlobalRelabel \cdot freq_u > \alpha \cdot n + m$ then
 8:             GLOBALRELABEL($H, s, t, f$)
 9:             $workSinceLastGlobalRelabel \leftarrow 0$
10: end function
```

of scanned edges plus β is added to it. This causes the value to rise if nodes are relabeled, while letting it stay the same if discharge-operations can discharge a node completely. If *workSinceLastGlobalRelabel* exceeds $\alpha \cdot n + m$, where n and m are the number of nodes and edges in the Lawler network, we execute the global relabeling routine and reset *workSinceLastGlobalRelabel*.

Algorithm 13 Discharge

```
 1: function DISCHARGE($u$)
 2:     $d_{\min} \leftarrow n$
 3:     if $u$ is regular node then
 4:         for $e \in I(u)$ do
 5:             if $r_f(u, e_{\text{in}}) > 0$ then                ▷ Scan Lawler edge $(u, e_{\text{in}})$
 6:                 if $d(u) = d(e_{\text{in}}) + 1$ then
 7:                     Push flow $min\{r_f(u, e_{\text{out}}), e(u)\}$ from $u$ to $e_{\text{in}}$
 8:                 else
 9:                     $d_{\min} \leftarrow min\{d_{\min}, d(e_{\text{in}})\}$
10:             Scan Lawler edge $(u, e_{\text{out}})$ in the same way
11:     else if $u = e_{\text{in}}, e \in E$ then
12:         Scan Lawler edge $(e_{\text{in}}, e_{\text{out}})$
13:         for $v \in e$ do
14:             Scan Lawler edge $(e_{\text{in}}, v)$
15:     else                                                   ▷ $u$ is edge-out node $e_{\text{out}}$
16:         for $v \in e$ do
17:             Scan Lawler edge $(e_{\text{out}}, v)$
18:         Scan Lawler edge $(e_{\text{out}}, e_{\text{in}})$
19:     if $e(u) > 0$ then                                     ▷ Relabel
20:         $d(u) \leftarrow d_{\min} + 1$
21:         $workSinceLastGlobalRelabel \leftarrow \beta + |I(u)|$
22: end function
```

We will now go into detail on the discharge operation as the core component of phase 1. Its task is to scan all incident edges of a node in the Lawler network, push excess to adjacent nodes and relabel the node if there remains excess that can not be pushed away. Pseudocode for the discharge is depicted in Algorithm 13. Because we work on the hypergraph datastructure and emulate the Lawler network, the discharge operations for regular nodes, edge-in nodes and edge-out nodes look slightly different as their incidences represent different kinds of nodes. For regular nodes, we have to scan all incident edges e and try to push flow to e_{in} and e_{out}. For an edge-in node e_{in}, all pins of the edge scanned in addition to e_{out}. Again, the order of push operations is chosen based on the intuition that in most cases, flow is pushed from e_{in} to e_{out} as pushing from e_{in} to a pin means that already existing flow is pushed back. An edge-out node e_{out} can push flow to pins of e or to e_{in}.

A more precise description of an edge-scan is given for the case that we try to push flow from u to e_{in}. It does not differ substantially from the standard implementation for Push-Relabel algorithms. First, we check if the residual flow is greater than 0. Then, we check for admissibility of a Lawler edge in the sense that the labels differ by exactly one and the label of u is bigger. If this is the case, flow can be pushed from u to e_{in}. This push operation changes the datastructures that represent the flow value of edges and sets the excess values of the nodes by decreasing the excess of u and increasing the one of e_{in}. For a possible relabel of u and the end of the discharge operation, we have to keep track of minimum label of an adjacent Lawler node with positive residual capacity. This is done in the same way as in the standard algorithm.

7.2.3. Global Relabeling

The global relabeling optimization is essential for the performance of the algorithm as it can reduce the total number of discharges that are necessary to compute the maximum preflow. The labels generated by the global relabeling correspond to distances (in terms of hops) to the target t. This means that directly after the relabeling excess flow from a node u can be pushed over the shortest path to t without relabeling any nodes. Still, it is not always optimal to push flow from u over the shortest path and the global relabeling remains a heuristic. In the case that not all flow can be pushed on the shortest path we still need to be able to relabel nodes normally. The frequency of the global relabeling operations is crucial for the performance of the algorithm.

The implementation of the global relabeling function is a simple reverse BFS from t in the residual network using its own separate search queue. For new nodes that are discovered, we check whether they have positive excess and a label smaller than n. If so, we add them to the main Push-Relabel queue Q which we reset at the beginning of the relabeling. Nodes that were not visited cannot reach t and their label is set to n.

7.2.4. Flow decomposition

In the last step of the Push-Relabel algorithm, the max preflow has to be converted into a maximum flow. A standard flow decomposition is suited for this task. A proof of its correctness can be found here: [GTT89]. Usually, a flow decomposition splits a preflow into paths and cycles routing some flow f. Out of this decomposition, we only want to keep the paths that route flow from s to t. Other paths will also start in s, but route flow to a node with excess. We discard these paths as well as the cycles that were created by the Push-Relabel phase before. To this point, we did not have to pay much attention to the fact that we are not working on graph without any preexisting flow, but on one flow problem that is part of a sequence with already existing underlying flow from previous iterations. The details of how flow is stored during the execution of the whole Push-Relabel algorithm are given later. Here, we will focus on the operating principle of the flow decomposition.

The decomposition begins by computing the capacities $c_r := r_{f_{pr}} - r_f$ of the residual network, where $r_{f_{pr}}$ is the value of the preflow computed by the push and relabel operations and r_f is the previous max flow. Negative values represent residual flow in the opposite direction. The computation is done by iterating over all hyperedges and their in- and

Algorithm 14 Flow decomposition

```
Input: Maximum s-t preflow
Output: Maximum s-t flow
 1: function FLOWDECOMPOSITION($H = (V, E)$, $s$, $t$, $f$)
 2:     compute residual capacities $c_r$
 3:     $stack \leftarrow$ empty node stack
 4:     $stack$.PUSH($s$)
 5:     while $stack \neq \emptyset$ do
 6:         $u \leftarrow stack$.TOP()
 7:         $v \leftarrow invalid$
 8:         for $(u, w) \in E_L(H)$ do
 9:             if $c_r(u, w) > 0$ then
10:                 $v \leftarrow w$
11:                 break
12:         if $v$ is not $invalid$ then
13:             $stack$.PUSH($v$)
14:             $P \leftarrow s - v$ path on $stack$
15:             $c_{min} \leftarrow min\{c_r(v_1, v_2) \mid (v_1, v_2) \in P\}$
16:             if $v = t$ then
17:                 push back flow $c_{min}$ along $P$ and add flow to $f$
18:                 pop $stack$ down to lowest bottleneck $v_b$
19:             else if $e(v) > 0$ then
20:                 push back flow $min\{c_{min}, e(v)\}$ along $P$
21:                 pop $stack$ down to lowest bottleneck $v_b$
22:             else if $v$ is on the stack then
23:                 remove cycle by pushing back flow
24:         else
25:             $stack$.POP()
26:     write back flow
27: end function
```

out-pins. Then we start a depth-first search from s using an explicit node stack. The idea is very similar to the one used for the sequential implementation of the part of the Dinic algorithm that finds augmenting paths in Section 6.2. Each time we encounter a node v in the DFS we push it to the stack. From bottom to top, the stack then contains a series of nodes from s to v that represents a path P. After we encountered v, we have to check whether we should progress the DFS or if we can push back flow. There are multiple cases that can arise:

If $v = t$, then we reached the target. Consequently, P is an s-t-path with positive flow value on its edges. This is flow that we want to keep for the final max-flow. The maximum flow by that we can augment the path is $c_{min} = min\{c_r(v_1, v_2) \mid (v_1, v_2) \in P\}$. This value can be found by iterating once over the stack. Additionally, we need to know the lowest bottleneck node v_b on the stack. It is the lowest node on the stack such that the residual capacity to the next node towards t is c_{min}. After we augmented c_{min} on P, v_b is the last node that we can reach from s along P. The augmentation operation itself consists of augmenting the flow in c_r, thereby reducing the residual capacities and we add the flow to the final max-flow f. Then, we pop all nodes from the stack down to v_b.

If $e(v) > 0$, we reached a node with excess that has to be pushed back. We do exactly the same as in the case before, except that we do not add the augmented flow to f, but just reduce the residual capacities and pop the stack down to the bottleneck.

If v was already pushed to the stack before, then we encountered a cycle C between the two occurrences of v on the stack. We search for the minimum residual flow on C and augment the flow on C by this amount. Afterwards, we pop the stack down to the lowest bottleneck node on C. We do not add this flow to the final flow f.

In the last case, we proceed with the DFS by pushing v to the stack.

7.2.5. Adaptions for the FlowCutter framework

Here, we want to briefly discuss the changes to the standard Push-Relabel algorithm that were necessary to run the algorithm in the WHFC framework. The first one was already mentioned before and concerns the underlying flow that the hypergraph may have when we start executing the max-flow algorithm. This can make a difference at the beginning of the algorithm when we push flow away from s as it is already possible to push flow back. FlowCutter solves a sequence of flow problems where the source and target are not just single nodes as we defined them, but sets of terminals . We do not have to pay too much attention to this fact as FlowCutter computes piercing nodes that are the only nodes that can break the maximality of the flow, making them the only ones that have to be considered in the max-flow algorithm.

7.2.6. Optimizations

To further improve the algorithm, certain optimizations were tested. The following Sections explain how they work and what advantages and disadvantages arise from them.

7.2.6.1. Current arc datastructure

One common optimization to speed up the computation of the maximum preflow is the current arc datastructure. Its goal is to prevent scans of edges that were already done before and where no flow can be pushed. The idea is based on the fact that when an incident Lawler edge (u, v) was scanned during the discharge operation of a Lawler node u and we could not push flow to v, then we know that as long as u is not relabeled, we will not be able to push flow to v. This holds because if we scan (u, v) and we cannot push flow, then there are two possible reasons for that: either the residual capacity $r_f(u, v)$ is zero or it is not zero but $d(u) \neq d(v) + 1$. In the first case, the residual capacity on the edge has to become positive which can only happen if flow is pushed from v to u. This implies that $d(v) > d(u)$ and as long as u is not relabeled the edge will not meet the label constraint to let us push flow from u to v. In the second case where the residual capacity is positive, we know that $d(v) > d(u)$ too, because f is a valid label function. Therefore we will also not be able to push from u to v without relabeling u before.

This observation can now be used to avoid unnecessary Lawler edge scans. For a node u, we keep an index to the last incident edge that we scanned. After we finished discharging a node and we did not have to relabel it, we set the index to the last edge we scanned. The next time we discharge this node, we will start scanning the incidences at this position and skip all previous edges. This only works as long as we don't relabel the node. If we relabel the node, we still have to scan all previous edges to find the incident node with the lowest label. We cannot just store the lowest label that we encountered since we last relabeled the node because in between discharge operations of the same node, the labels of neighbors can change. The global relabel function resets all indices to incident edges.

For regular Lawler nodes, we keep an index to incident hyperedges. For Lawler nodes that represent edge-in or edge-out nodes, we keep an index to the pins of the hyperedge. This approach only works as long as the order of incident hyperedges or pins stays the same. We will present another optimization that reorders incident pins, so these two optimizations can not be used at the same time.

7.2.6.2. Current edge for depth first search

The depth first search of the decomposition algorithm can consider a node u of the Lawler network more than once, for example when u is the lowest bottleneck of an augmentation step. Instead of scanning all incident edges every time we consider a node, we can keep a pointer to the last incident edge we scanned and advance it with the scan. When we return to v, we start scanning from this incidence. We can do this because we know that the residual capacity can only get smaller on edges and edges that we already scanned do not have residual flow any more, otherwise we wouldn't have advanced to the next edge. This is especially useful because a lot of the edges that we scan will not have any residual capacity and skipping them when continuing the scan of a node saves a good amount of operations.

To implement this for the Lawler network, we need three additional arrays: One array storing the next hyperedge to scan for regular nodes, and two arrays pointing to the next pins of a hyperedge for edge-in and edge-out nodes.

7.2.6.3. Sorting pin arrays

The goal of this optimization is to make it possible to only iterate over Lawler incidences where flow is either sent or no flow is sent. A similar approach lead to considerably better performance for the depth first search that finds augmenting paths in the Dinic algorithm. This is the reason why we wanted to try it in this context. The idea is that we can use it to speed up the discharge operation of edge-in nodes after we exceeded the capacity of a hyperedge and want to push flow back to pins. For these nodes, we have to iterate over all pins that are sending flow into a hyperedge as only these can be pushed back. The pin-array is partitioned into sending and non sending pins such that the sending pins are stored up front and the non sending pins behind them. An index to the first non sending pin of an edge-in node enables us to only iterate over the sending or non sending pins exclusively. However, the edge-in nodes are the only nodes that benefit from sorting the pin-arrays as edge-out nodes can push to all pins. Nevertheless, we have to keep each pin interval of a hyperedge sorted by pins that send and do not send flow and we need to store a pointer to the first pin that does not send flow. The ordering has to be kept intact, meaning that every push operation that changes the flow of a pin and changes its status from sending to non sending (or the other way around) has to update the pin array to reflect this change. In the end, the optimization did not yield any performance gains, so we dropped it.

7.3. Parallel Push-Relabel

To parallelize the Push-Relabel algorithm, we want to process the discharge operations in parallel. Only phase 1 will be adapted, while the flow decomposition in phase 2 stays the same. The reason for this is that phase 2 takes less time [BBS15, CG95] and the DFS used for the decomposition is harder to parallelize. Phase 1 is replaced by the continuous application of a pulse operation that goes back to an idea of Goldberg and Tarjan [GT88].

The pulse consists of 3 stages, each processing nodes in parallel. Stage 1 discharges all active nodes in parallel, but writes new labels to a a copy d' of d. Changes in excess that reduce the value are applied directly, a positive change in excess is written to e_{change}. Nodes that become active are pushed to thread-local vectors so we can access them in the next pulse. In stage 2, we iterate over all nodes that were active at the beginning of the pulse again, write d' to d and update the excess values. Stage 3 then updates the excess of nodes that were not necessarily active in the beginning, but became active after flow was pushed to them.

7.3.1. Pulse operation

During one pulse, all nodes that are active in the beginning of it are discharged in parallel. To keep the set of active nodes, we use two datastructures Q and Q' that are used in a ping-pong like manner. While Q contains the current set of active nodes, we add the nodes that were activated for the next pulse iteration to Q'. At the end of the pulse, Q and Q' are swapped and Q' is cleared. The main challenge for these datastructures is the fact that as we process nodes from Q in parallel, we add nodes to Q' in parallel from different threads. This would usually cause some synchronization overhead if Q was a vector or queue. To reduce the need for costly synchronization, we chose an implementation using thread-local vectors. Q and Q' are made up of a set up vectors, one for each thread. To iterate over Q in parallel, we iterate over the set of vectors and their respective nodes in parallel. This is realized by using TBB's `tbb::enumerable_thread_specific` class that provides management of thread local objects. The same class is used to store $work_{local}$, a thread-local variable that holds the work that the thread has done during the last pulse. Because we need to know if a node is already part of any of the thread-local vectors before we can add it to the local one, we use one vector of atomics *inQueue* that is **true** if the node is in Q and **false** otherwise.

Algorithm 15 Pulses

```
 1: function MAX PREFLOW PARALLEL($H = (V, E)$, $s$, $t$, $f$)
 2:     while $Q \neq \emptyset$ do
 3:         RESET(inQueue)
 4:         if $workSinceLastGlobalRelabel \cdot freq_{\mathrm{u}} > \alpha \cdot n + m$ then
 5:             GLOBALUPDATE($H, s, t, f$)
 6:             $workSinceLastGlobalRelabel \leftarrow 0$

 7:         parallel for $u \in Q$ do                          ▷ stage 1: discharge
 8:             SAFEDISCHARGE($u$)

 9:         parallel for $u \in Q$ do                          ▷ stage 2: update $Q$
10:             $d(u) \leftarrow d'(u)$
11:             $e(u) \leftarrow e(u) + e_{\mathrm{change}}(u)$
12:             $e_{\mathrm{change}}(u) \leftarrow 0$

13:         parallel for $u \in Q'$ do                         ▷ stage 3: update $Q'$
14:             $e(u) \leftarrow e(u) + e_{\mathrm{change}(u)}$
15:             $e_{\mathrm{change}}(u) \leftarrow 0$

16:         $Q \leftarrow Q', Q' \leftarrow \emptyset$
17:         $workSinceLastGlobalRelabel \leftarrow \sum_{work_{local}} work_{local}$
18: end function
```

The pseudocode 15 shows the three different stages of the algorithm, each iterating over Q in parallel. The first one discharges all nodes in parallel. When discharging a node u, we possibly change $d(u)$ and the excess of u itself or the excess of neighbors of u. Additionaly, flow on incident edges can be changed. If two adjacent nodes are active, this can lead to conflicts on these datastructures. To resolve them, we use a second label function d' that stores new labels for the next iteration so that we do not need to change d while discharge operations take place. For the excess, we allow processed nodes to change their own value, but excess that is pushed to other nodes is added to e_{change}. The access to the network flow function is controlled by logically assigning ownership of Lawler edges to nodes. We go into detail on the new discharge operation in the next Section.

After we finished the first stage and all discharges are done, we copy the new labels d' to d and add the positive excess change to their nodes. This only has to be done for the nodes that were active in the first stage, as only their label had been able to change. However, the excess was also added to neighboring nodes that were not previously active, but got activated by pushing excess to them. That's why in stage 3, the excess is added to the active nodes of the next iteration. Lastly, we need to collect the thread local work values and add them up.

The global relabeling step is applied with the same rules as in the sequential version and is parallelized too. It is a reverse breadth first search starting in t and the same approaches to parallelizing a BFS we described before can be utilized.

7.3.2. Discharge

We will now explain the thread-safe version of the discharge-operation for nodes. Pseudocode 16 shows it for nodes u of the Lawler graph. Keep in mind that we use an implicit representation of the Lawler graph. Therefore, it makes a difference whether we are discharging a regular node or an edge-in/edge-out node in the real implementation. For edge-in and edge-out nodes we iterate over incident pins of the respective hyperedge. For regular nodes we iterate over incident hyperedges and their edge-in and edge-out nodes of the Lawler network.

The main datastructures to avoid data conflicts were introduced before. Only working with the old labels and not adding excess to other nodes when pushing to them means that every discharge works as if it was executed on the graph at the beginning of the pulse. Furthermore, there are no conflicts regarding the access of the flow function of edges as only the node with higher label is allowed to push to the lower label one. With these small modifications we already have a working parallel discharge function. This is also the way that Goldberg and Tarjan presented it in 1988 [GT88].

To take it a bit further, we use an optimization provided in [BBS15]. It addresses the problem that we can only relabel a node once during each pulse which means we can not always push all flow to its neighbors. It takes multiple discharge iterations and eventually more pulses to push away all excess of a node. Another reason is the fact that the workload per node can be relatively small and it is favorable for the performance of the algorithm to not parallelize on a level too fine-grained.

The optimization introduces a winning criterion that assigns edges to nodes that are then allowed to push flow on the edge. It guarantees that for two adjacent nodes, only one node wins the connecting edge. Line 10 shows the criterion for the edge (u, v). Keep in mind that we use d and d' and only the thread that is working on a node u uses d' as its current label while other nodes only consider d which will always retain the value it had at the start of stage 1. The criterion consists of three cases: If $d(u) = d(v) + 1$ then we look at an admissible edge that we would be able to push through even without the optimization. If $d(u) < d(v) - 1$, we know that v cannot push to u, even if another thread relabels v as labels can only get higher. The last condition is a tie-breaker for the case that $d(u) = d(v)$ and we let the node with the lower id win the edge. Because these conditions only take into consideration the value of $d(u)$ and not the label $d'(u)$ we are currently considering, it is possible that after we relabeled the node, we encounter an edge that is admissible (in the sense that $d'(u) = d(v) + 1$) but we do not win the edge and thus cannot push. We can still push flow through other incidences, but we cannot relabel u any further in this iteration.

Algorithm 16 SafeDischarge

```
1: function PARALLEL DISCHARGE($u$)
2:     $d'(u) \leftarrow d(u)$
3:     while $e(u) > 0$ do
4:         $skipped \leftarrow false$
5:         $d_{\min} \leftarrow n$
6:         for $(u, v) \in E(H_L)$ do
7:             if $e(u) = 0$ then break end if
8:             $admissible \leftarrow d'(u) = d(v) + 1$
9:             if $e(v) > 0$ then
10:                $win \leftarrow d(u) = d(v) + 1$ or $d(u) < d(v) - 1$ or $(d(u) = d(v)$ and $u < v)$

11:                if $admissible$ and not $win$ then
12:                    $skipped \leftarrow true$
13:                    continue
14:            $residual \leftarrow min\{e(u), r_f(u, v)\}$
15:            if $admissible$ and $residual > 0$ then
16:                $f(u, v) \leftarrow f(u, v) + residual$
17:                $e(u) \leftarrow e(u) - residual$
18:                $e_{\text{change}}(v) \leftarrow e_{\text{change}}(v) +_a residual$
19:                if $inQueue$.TESTANDSET($v$) then $Q_{\text{local}}$.ADD($v$) end if
20:            if $r_f(u, v) > 0$ and $d(v) \geq d'(u)$ then
21:                $d_{\min} \leftarrow min\{d_{\min}, d(v)\}$
22:        if $e(u) = 0$ or $skipped$ then
23:            break
24:        $d'(u) \leftarrow d_{\min} + 1$                                    ▷ relabel
25:        $work_{local} \leftarrow work_{local} + \beta + |\{(u, v) \in E(H_L)\}|$
26:        if $d'(u) = n$ then
27:            break
28:    if $e(u) > 0$ and $d'(u) < n$ and $inQueue$.TESTANDSET($u$) then
29:        $Q_{\text{local}}$.ADD($u$)
30: end function
```

The reason is the following: Assume we have nodes u and v with $r_f(u, v) > 0$ and $d(u) = d(v) - 1$. If we relabel u without exhausting the residual edge and the label of v stays the same, then we are left with the residual edge (u, v) that is not admissible and $d(u) > d(v) - 1$. This breaks the validity requirements of d and thus a crucial invariant of the Push-Relabel algorithm. As a consequence, when trying to discharge u again in the next iteration, we may try to relabel u to $d(v) + 1$ because v is the neighbor with the lowest label and positive residual capacity on the connecting edge. u is then relabeled to a value that is not bigger than $d(u)$ and labels do not necessarily grow strictly monotonically any more. This is why when this case arises, *skipped* is set to *true* and we don't relabel.

The other changes to the sequential version were already discussed. Excess that is pushed to other nodes is written to separate value $e_{\text{change}}(u)$. This operation has to be atomic as multiple nodes can push to u at the same time. Other places where atomic operations are necessary are line 19 and 28 to atomically test and set the value of the *inQueue*-array to be able to add a node to the thread-local vector.

8. Overview of the main datastructures

In this chapter we want to give a small overview of the main datastructures that are performance critical for our application. We want to concentrate on the hypergraph datastructure that is used for representation of the snapshots and therefore has to store flow values and the partition implementation that is optimized to give faster access to cut hyperedges between two blocks. The running time greatly depends on the time it takes to make accesses to these datastructures, for example to get the incidences of a vertex in the hypergraph. This is why we did not want to leave this contemplation out of the thesis.

8.1. Lawler-FlowHypergraph-Datastructure

In this Section we briefly want to present the datastructure used for the hypergraph that stores flow and has to enable fast access on incidences of hyperedge-nodes and incidences of Lawler-nodes. It is a more complete description of the simplified version that we introduced in Section 5.2.2 and is therefore based on the compressed sparse row representation. Figure 8.1 depicts the main arrays that represent the hypergraph, including the member variables for each type. These arrays are:

`nodes`: Indexed by the hypergraph node id. It stores a node's weight and the first hyperedge incidence of the node. A sentinel is inserted at the end of the array. This makes it possible to efficiently scan incident hyperedges.

`hyperedges`: Stores the capacity and flow going through a hyperedge. `first_out` is an index for the first in- and out-pin of a hyperedge. This array also contains a sentinel at the end.

`pins_in` and `pins_out`: `pins_in` stores the flow going into a hyperedge and `pins_out` the flow from the hyperedge to the pin. `he_inc_iter` gives access to the hyperedge incidence data.

`incident_hyperedges`: The hyperedge incidence data stores the edge-id `e` which is also used as an index into the `hyperedges` array. `flow` represents flow on the incidence. `pin_iter_in` and `pin_iter_out` point to the correct in- and out-pins for the hyperedge incidence.

The reason that there are two pin-arrays instead of just one (and consequently two indices `pin_iter_in` and `pin_iter_out`) is the optimization presented in Section 7.2.6.3. However, it turned out that it did not yield the speedups we hoped for. To have faster access to

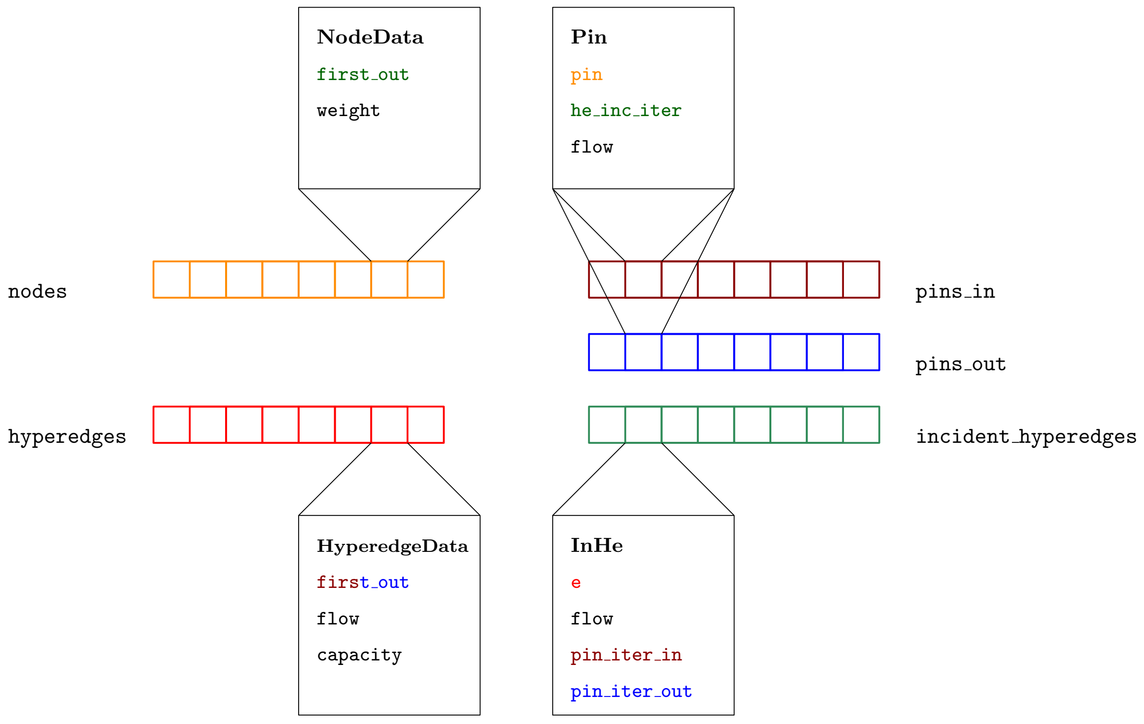

Figure 8.1.: Overview of the datastructure to describe the hypergraph. Colors represent the array that a value is an index of. `first_out` of HyperedgeData indexes both `pins_in` and `pins_out`

pins that send or receive flow, we wanted to be able to partition the pins into those that send, receive or don't have any flow. Because during the execution of the Push-Relabel algorithm pins can send and receive flow at the same time, we need two arrays. Although we move pins that belong to the same hyperedge around, the first pin of a hyperedge will always remain at the same index, which is why HyperedgeData only needs one value for the first pin that refers to two arrays.

Flow can be stored in three different places: the `HyperedgeData`, the `Pin` and the `InHe`. The flow that runs through a hyperedge is always stored in `HyperedgeData`. For the flow of pins we use two different representations. The flow stored in `InHe` is positive for flow that the pin sends into a hyperedge and negative for flow that is received. The flow values in the in- and out-`Pins` store separate values and are used by the Push-Relabel algorithm to be able to let pins send and receive flow at the same time. These values are always greater or equal to zero. When transforming the maximum preflow into a maximum flow in the Push-Relabel algorithm, we make sure that the flow values are consistent. The maximum preflow phase only updates the values of the `Pin` and `HyperedgeData` structure. The decomposition synchronizes them between the `Pin` and `InHe` structures in the sense that they represent the same values at the end of the decomposition.

Besides the hypergraph datastructure itself, the Push-Relabel algorithm also needs to store labels and nodes excesses. This is done with two arrays that are indexed by the Lawler node id. The advantage of using arrays here is that they are easier to reset as a whole, which is necessary during the execution of the algorithm.

8.2. Partition with cut-edge precomputation

Alongside the hypergraph, the datastructure representing the partition is used for a lot of operations. A partition can be implemented in an easy manner by using one vector that stores the partition id of each vertex. However, looking at our use case, there are some operations that are frequently used and become costly on this implementation. The extraction of the flow hypergraph needs to iterate over all cut hyperedges between two blocks. With the simple implementation, we have to iterate over all hyperedges of the hypergraph and their pins to determine if it is part of the cut. The implementation that will be presented next aims to speed up the following computations: Get the number of pins in one block of a hyperedge, get the total weight of a block and get the cut hyperedges between two blocks. The precomputation of these values especially makes sense for the k-way refinement where we keep using the same partition with the same number of blocks. For the recursive bisection, we compute new partitions at each step and only apply one FlowCutter iteration to them. Here, we cannot make much use of the precomputation as the values only have to be used once.

For the parallel k-way refinement, another challenge arises. There are multiple threads working on different blocks at the same time. We have to be careful to rule out race conditions that might occur otherwise.

Mirroring the previous computations that should be sped up, we introduce the following additional datastructures:

$pinsInPart(p_i, e)$:	The number of pins that $e \in E$ has in $p_i \in [k]$
$weight(p_i)$:	The total weight of block $p_i \in [k]$
$cutEdges(p_1, p_2)$:	Candidates for cut-edges between p_1 and p_2

The first two are implemented using vectors and their value can be returned on request. We will go into detail how the last operation works. Another thing that remains to be shown is how we handle the movement of a vertex to another block. This happens after the refinement when we move vertices to other blocks to improve the cut. The precomputed values have to be adapted to reflect these changes. Lastly, we need an initialization procedure that precomputes the values.

8.2.1. Obtaining cut hyperedges

For each block pair $\{p_1, p_2\}$ we keep two vectors with hyperedges. These hyperedges are cut-hyperedge candidates and may not be real cut-hyperedges, but we guarantee that the union of both vectors is a superset of the real cut-hyperedge set. Furthermore, the vectors may contain duplicate elements. Why we need two vectors and why there may be hyperedges that are not part of the cut will be clarified when we explain how we change the block of a vertex.

We differentiate between the two vectors of a block pair by using $cutEdges(p_1, p_2)$ and $cutEdges(p_2, p_1)$ to address them. Upon request, we iterate over both vectors to remove hyperedges that are not part of the cut and eliminate duplicate hyperedges. Then, we can return the concatenation (as a range) of both vectors.

8.2.2. Changing the block of a vertex

Because it is to be expected that vertices will be moved between blocks several times, we want this operation to work without additional synchronization between threads. To realize this, we make the assumption that the calling thread has just done a refinement between the old block p_{old} and the new block p_{new} of u. Now it moves u from one block to the other to improve the cut. Our parallel k-way scheduling guarantees that no two threads work on the same block for a refinement, meaning that no other thread can change the block of a vertex to p_{new} as it would have to refine this block to do so. We will use this fact to synchronize accesses to certain vectors.

Algorithm 17 Change block

```
 1: function CHANGEBLOCK($u, p_{\text{new}}$)
 2:     if $P(u) \neq p_{\text{new}}$ then
 3:         $p_{\text{old}} \leftarrow P(u)$
 4:         $P(u) \leftarrow p_{\text{new}}$
 5:         for $e \in I(u)$ do
 6:             if $pinsInPart(p_{\text{new}}, e) = 0$ then
 7:                 for $p' \in [k] \setminus \{p_{\text{new}}\}$ with $pinsInPart(p', e) > 0$ do
 8:                     $cutEdges(p_{\text{new}}, p')$.ADD($e$)
 9:             $pinsInPart(p_{\text{old}}, e) \leftarrow pinsInPart(p_{\text{old}}, e) - 1$
10:             $pinsInPart(p_{\text{new}}, e) \leftarrow pinsInPart(p_{\text{new}}, e) - 1$
11:         $weight(p_{\text{old}}) \leftarrow weight(p_{\text{old}}) - \varphi(u)$
12:         $weight(p_{\text{new}}) \leftarrow weight(p_{\text{new}}) + \varphi(u)$
13: end function
```

Algorithm 17 iterates over all incident hyperedges e of u to check whether e previously had no pin in p_{new}, making u the first one. This is implemented by reading *pinsInPart* which provides constant access to the precomputed value. It still contains the old values at this point and is updated later. If we find a hyperedge e that did not have a pin in p_{new} before, we have to add e to all cuts between p_{new} and other blocks p' of e. It is possible that e was already added to the cuts before, which is why we have to eliminate duplicates later.

For each cut, we chose to use two vectors because two threads can try to add a new hyperedge to the cut at the same time. When adding hyperedges to the cut p_{new}, p' then we know that p_{new} is a block that thread t_1 just refined and is now moving vertices to. The block p' can be any other block. If another thread t_2 adds to the same cut, then this means that t_2 was working on the block p' (the variable of t_1) and is moving vertices to it. Therefore, the variables p' and p_{new} have to be switched for the two threads. That's why we can use the order of ids p_1 and p_2 to differentiate the two vectors of a cut $cutEdges(p_1, p_2)$ and $cutEdges(p_2, p_1)$. This way we will always add to a vector that we have exclusive access to.

One remaining problem is that to remove e from cuts that is not part of any more, we would possibly have to access these other vectors and need to iterate over all elements to see if it was already added. Instead, we filter the cut-edges when they are requested as shown before.

To initialize the partition, we iterate over all hyperedges e and their respective pins u, set *pinsInPart* accordingly and add e to all cuts that it is part of. Afterwards, we iterate over all vertices and increase the weight of their block by their vertex weight.

9. Experimental evaluation

For the experimental evaluation of the algorithms, we created a C++17 implementation that extends the WHFC project[1] and is available as open source software[2]. The parallelization was done using TBB[3]. The code was compiled with `g++9.2` and the flags `-O3 -mtune=native -march=native`. We ran the tests on compute nodes of the bwUniCluster 2.0 with two Intel Xeon Gold 6230 processors clocked at 2.1GHz and 96GB RAM.

We tested the different parallelizations for the extraction and the computation of the layered network for Dinic, as well as the sequential and parallel implementation of the Push-Relabel algorithm. In addition, the partitioners employing k-way refining with sequential and parallel scheduling were evaluated. We wanted to compare the sequential and parallel versions of the Recursive Bisection algorithm too, but ran into the problem that the interface for PaToH does not allow for parallel executions as some global allocations are made. Therefore, we only tested the Recursive Bisection with sequential execution of multiple partitionings and refinement steps.

9.1. Benchmark set

Our benchmark set is based on the one composed by [GHSS20] and was derived from several sources that span three application domains. Out of the 94 hypergraphs, there are 42 instances from the SuiteSparse Matrix Collection [DH11], 42 instances from the 2014 SAT Competition [BDHJ14] and all ten instances from the DAC 2012 Routability-Driven Placement Contest [VAS+12]. From the 94 hypergraphs that we tried to partition with PaToH, we ran into problems for 26 of them. PaToH failed with either an error message or a segfault on the cluster machines for them. After finishing the experiments we noticed that this was due to setting the `MemMul_CellNet` of PaToH too low which caused it to allocate not enough memory. The 68 hypergraphs that were tested are listed in Appendix A.

9.2. Methology

We use a value of $\varepsilon = 0.03$ for all our tests. All runs were made using five different seeds. To test the performance of the extraction and flow algorithms, we execute one

[1] `https://github.com/larsgottesbueren/WHFC`
[2] `https://github.com/floriangroetschla/WHFC`
[3] `https://www.threadingbuildingblocks.org`

iteration of WHFC on a bipartition of the benchmark set computed by PatoH. For the tests of the k-way refinement we do not restrict the number of WHFC executions and use $k \in \{2, 4, 8, 16, 32, 64\}$ for the instances. For algorithms executed in parallel, we always use 1,2,4,8 or 16 threads.

To aggregate running times and solution quality (the connectivity metric), we use the arithmetic mean over the seeds as a preprocessing step before continuing with the evaluation.

To compare the partitioning quality of different algorithms, we use performance profiles [DM02]. We define $\mathcal{A}$ to be the set of algorithms that we want to compare and $\mathcal{I}$ the set of instances, in our case a combination of input graph, ε, k and number of threads. Further, for $a \in \mathcal{A}$ and $i \in \mathcal{I}$, the solution quality $q_{a,i}$ is the arithmetic mean connectivity value (aggregated over the seeds) that algorithm a computed on instance i. The *performance ratio* is defined as

$$r_{a,i} := \frac{q_{a,i}}{min\{q_{a',i} \mid a' \in \mathcal{A}\}}$$

and represents the ratio of the solution quality for algorithm a on instance i compared to the best solution any algorithm found for i. The performance profile ρ_a of an algorithm $a \in \mathcal{A}$ is

$$\rho_a(\tau) := \frac{|\{i \in \mathcal{I} \mid r_{a,i} \leq \tau\}|}{|I|},$$

the ratio of instances for which a is within a factor τ of the best solution. The fractions of instances that timed out for all seeds are reported as the steps for the special symbol ✻, while instances that violate the balance constraint are depicted as steps for ✘.

For the comparison of running times, we use a combined scatter and box-and-whiskers plot. The scatter plot depicts the arithmetic mean time per instance as points, while the box shows the quartiles of the aggregated running times.

To analyze the scalability of an algorithm, we use speedup plots similar to the ones employed in [GHSS20]. The plot depicts points for the speedups of an instance and the cumulative harmonic mean speedup over all instances with a single-threaded running time $\geq x$ seconds as a line.

9.3. Parallelization of extraction and flow algorithms

To compare the performance of the extraction and WHFC with different implementations of extraction and flow algorithms independently from any scheduling scheme, we executed one iteration of WHFC on a 2-way partition (obtained with PaToH and preset D) of the hypergraphs in the benchmark set. We denote the different implementations as

`{seq, par}Extraction-{seq, par}{Dinic, PR},`

indicating which part was executed sequentially or in parallel and what flow algorithm is used. The combinations we tested were: `seqExtraction-seqDinic`, `seqExtraction-parDinic`, `parExtraction-seqDinic`, `seqExtraction-seqPR` and `seqExtraction-parPR`. The selection was chosen such that the different extraction and flow algorithms can be tested independently and sequential and parallel computations can be compared to each other.

9.3.1. Snapshot extraction

Figure 9.1 shows the running times of the different phases of the extraction when executed with one thread. The adding of hyperedges which is implemented as iterating over the hyperedges in parallel and the BFS take the most time. Both operations read larger

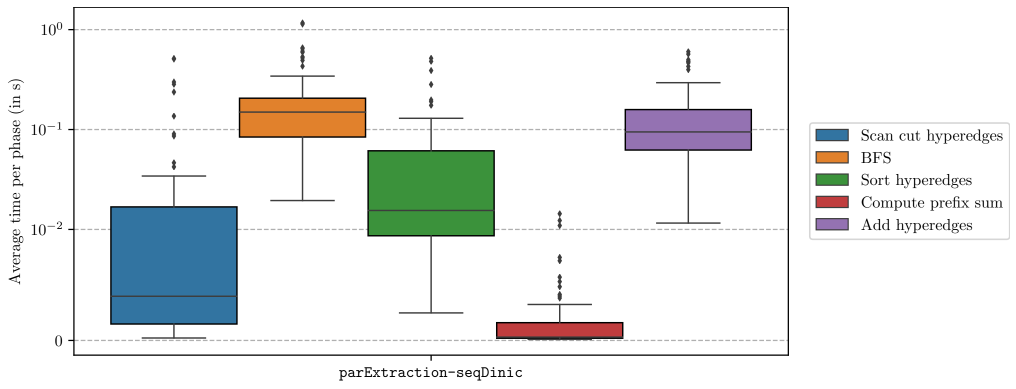

Figure 9.1.: Average running times of the extraction phases running with one thread

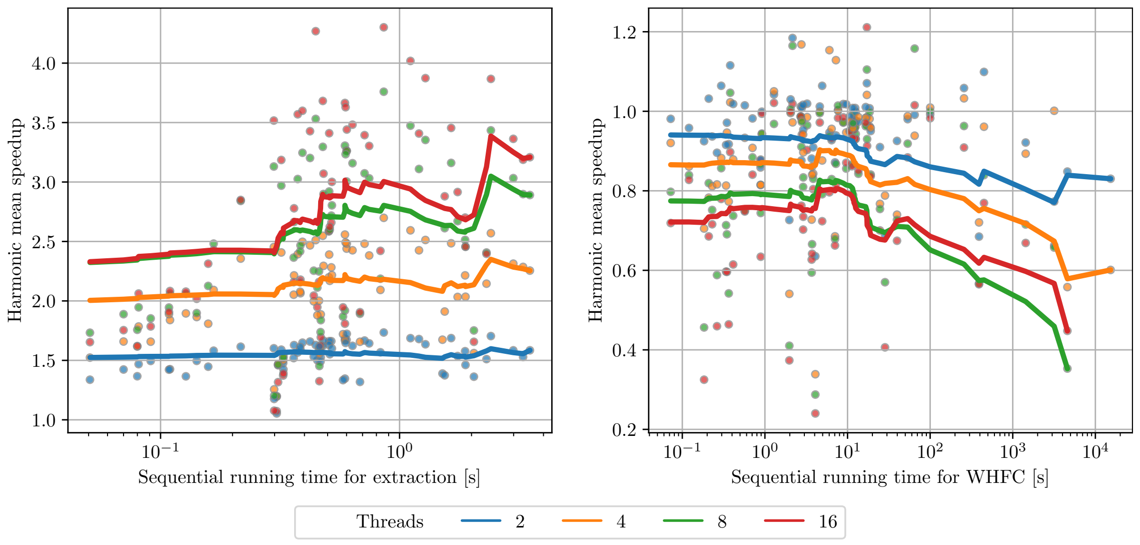

Figure 9.2.: Scalability plots for `parExtraction-seqDinic`

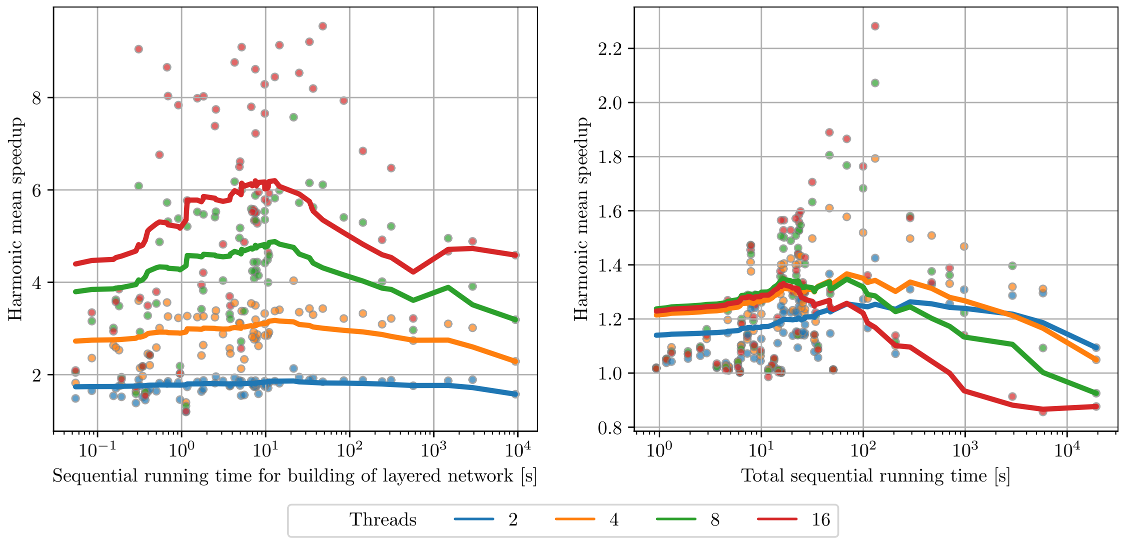

Figure 9.3.: Scalability plots for `seqExtraction-parDinic`

portions of memory with random accesses, suggesting that these memory operations are the bottleneck for the algorithm.

To analyze the scaling behavior, we consider Figure 9.2 with the speedup plots for the extraction and WHFC of `parExtraction-seqDinic`. While the speedups of the extraction are promising, we can see that the WHFC-run suffers greatly from different vertex/hyperedge-orders that the Extraction generates. This has a bad impact of the total running time of the algorithm, as the median running time for WHFC lies at 6.03 seconds for runs with one thread while the Extraction only takes 0.48 seconds.

9.3.2. Parallelization of building the layered network

The building of the layered network for the Dinic algorithm shows good speedups in Figure 9.3 which evaluates `seqExtraction-parDinic`. On the left, the speedups for building the layered network are depicted and for two threads, the algorithm has a consistently good speedup close to the theoretical optimum of 2 on all instances. However, in addition to building the layered network, the Dinic algorithm searches for augmenting paths sequentially, which is one reason why the speedups for the complete WHFC running times are smaller. In addition to this effect, the WHFC-speedup suffers for instances that take more than 100 seconds of sequential running time. Especially for very big instances, this effect becomes more evident. We can postulate that the way the layered network is built has an effect on the running time of the rest of the algorithm.

9.3.3. Push-Relabel

From preliminary tests on smaller instances, it became apparent that the value of the global update frequency $freq_{\mathrm{u}}$ for the Push-Relabel algorithm can have a big impact on the running time. The value determines how often the global update steps are executed and we saw that bigger values than those used by [BBS15] tend to work better for our use case. To assure that this is still the case for the bigger instances, we tested the values 1 and 5 for $freq_{\mathrm{u}}$ with our sequential and parallel implementation. The frequency $freq_{\mathrm{u}} = 5$ lead to consistently better running times, which is why all following executions are executed with this value. We chose $\alpha = 6$ and $\beta = 12$ for all runs, similar to [BBS15] and the implementation from Goldberg and Cherkassky.

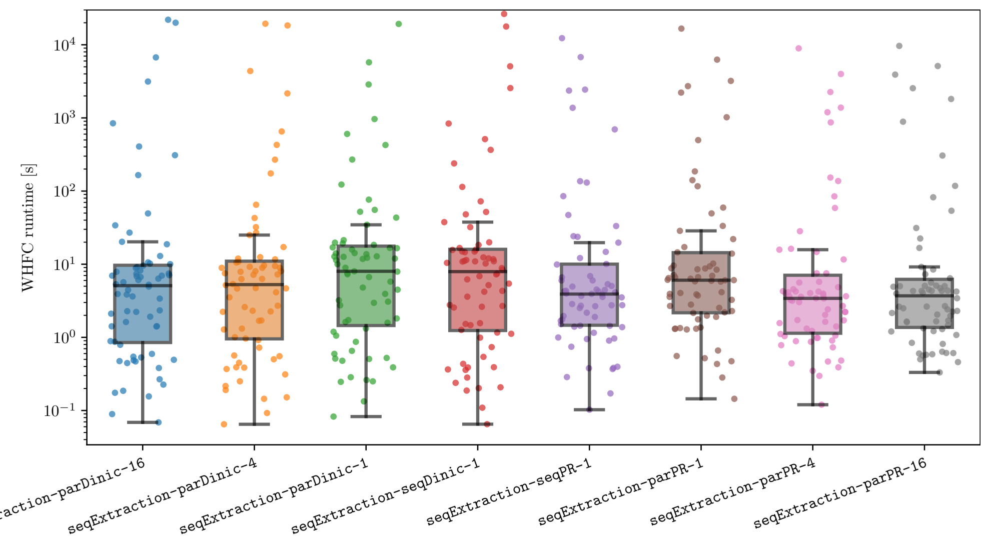

Figure 9.4.: WHFC runtimes for some algorithms. The suffix behind the name indicates the thread count.

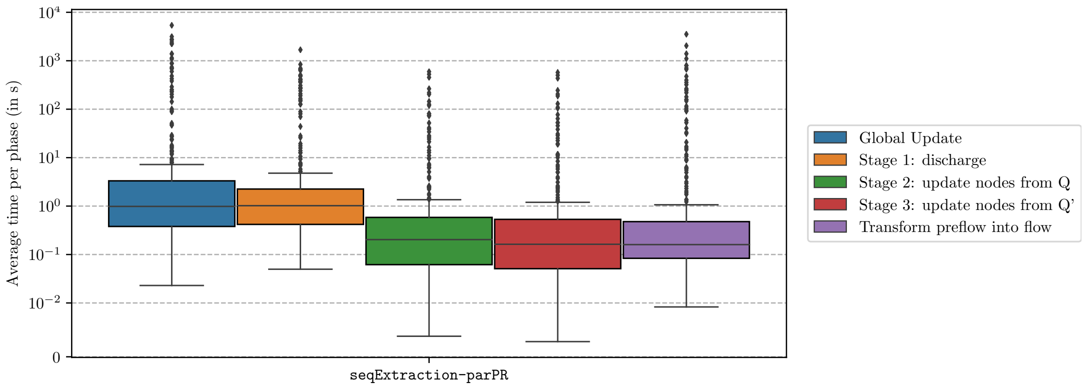

Figure 9.5.: Running times of the different phases executed with 1,2,4,8 or 16 threads

We compare the WHFC running times of the sequential and parallel Push-Relabel implementations to the running times using Dinic with and without the parallelization for the layered network. Figure 9.4 shows the running times of sequential and parallel flow algorithms with different numbers of threads in a box and scatter plot. The suffix behind the name of the algorithm indicates the thread count used for the execution. What we can see is that from all algorithms that were executed with one thread, the running times are relatively comparable, except the sequential implementation of the Push-Relabel algorithm that sticks out with a slight advantage in running time. One caveat that has to be considered is the granularity of timing measurements that we did for the different algorithms. The Dinic versions only measured the time it takes to build the layered network and the total running time of the flow algorithm. The sequential Push-Relabel algorithm only measured the time it took for the phase that computes the maximum preflow and the phase that transforms the preflow into a maximum flow by using the decomposition principles. The parallel Push-Relabel algorithm also measured the running times of the three different parallel stages (the stages are: discharging all nodes, updating excess and label of nodes that were discharged in the iteration and updating the excess of nodes that were activated) during the maximum preflow computation. Removing these more fine-grained measurements from the parallel version could improve its running times.

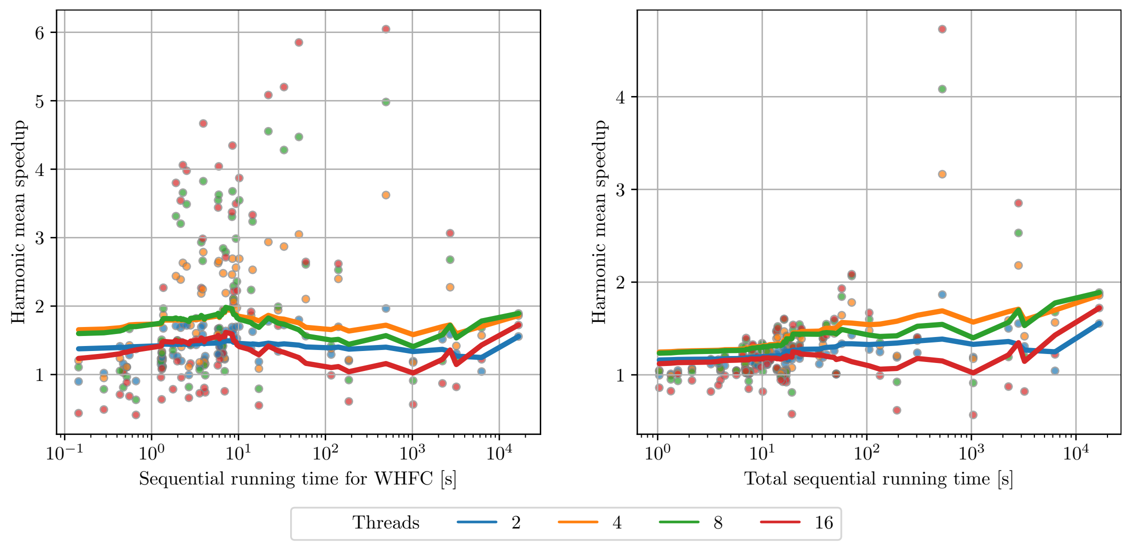

Figure 9.6.: Scalability plots for `seqExtraction-parPR`

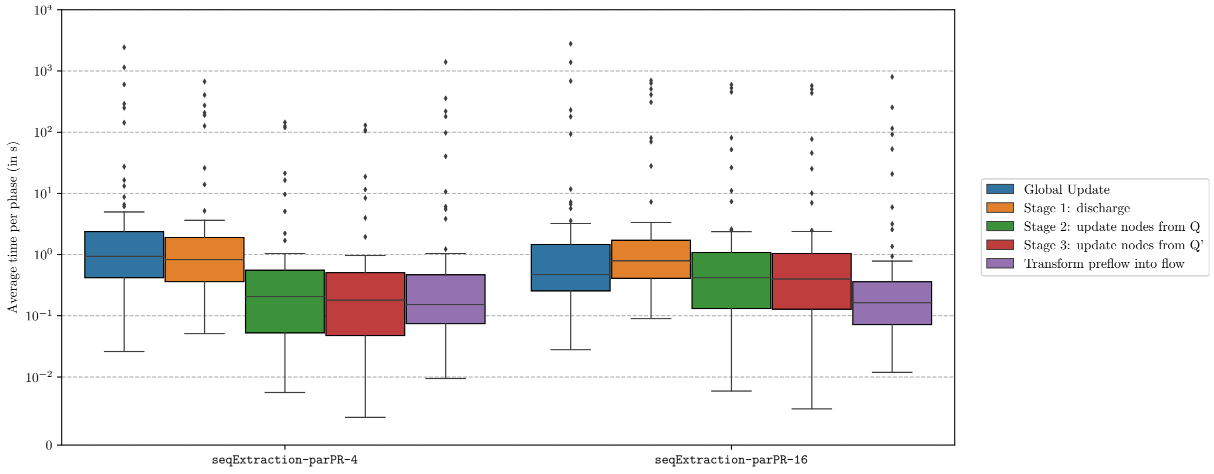

Figure 9.7.: Comparison of the phase times between 4 and 16 threads

A more detailed breakdown into the different parts of the parallel Push-Relabel algorithm is depicted in Figure 9.5. The running time for the Global Update includes the initial relabeling before any discharges are executed. The three stages represent the ones that are described in Algorithm 15. The algorithm spends most of the time with the global update of the labels and the parallel discharges in stage 1. Phase 2 is the only part of the algorithm that is not parallelized, but also takes considerably less time.

In terms of scalability, the performance gets better for more threads at first, but less so the more threads we use. Consider Figure 9.6 for a clearer representation in a scalability plot that shows the speedups of the `seqExtraction-parPR` algorithm for the WHFC and total running time. For 16 threads, the algorithm performs worse than for less threads. The best trade-off seems to be reached for 4 threads, where the algorithm provides the best speedups.

This is an unexpected result and to find the reason we consult a comparison of running times of different parts of the algorithm for 4 and 16 threads in Figure 9.7. Interestingly, all phases get faster or stay the same, except for stage 2 and 3. These stages update excess and label values and iterate over the thread-local vectors to do so, but don't execute any

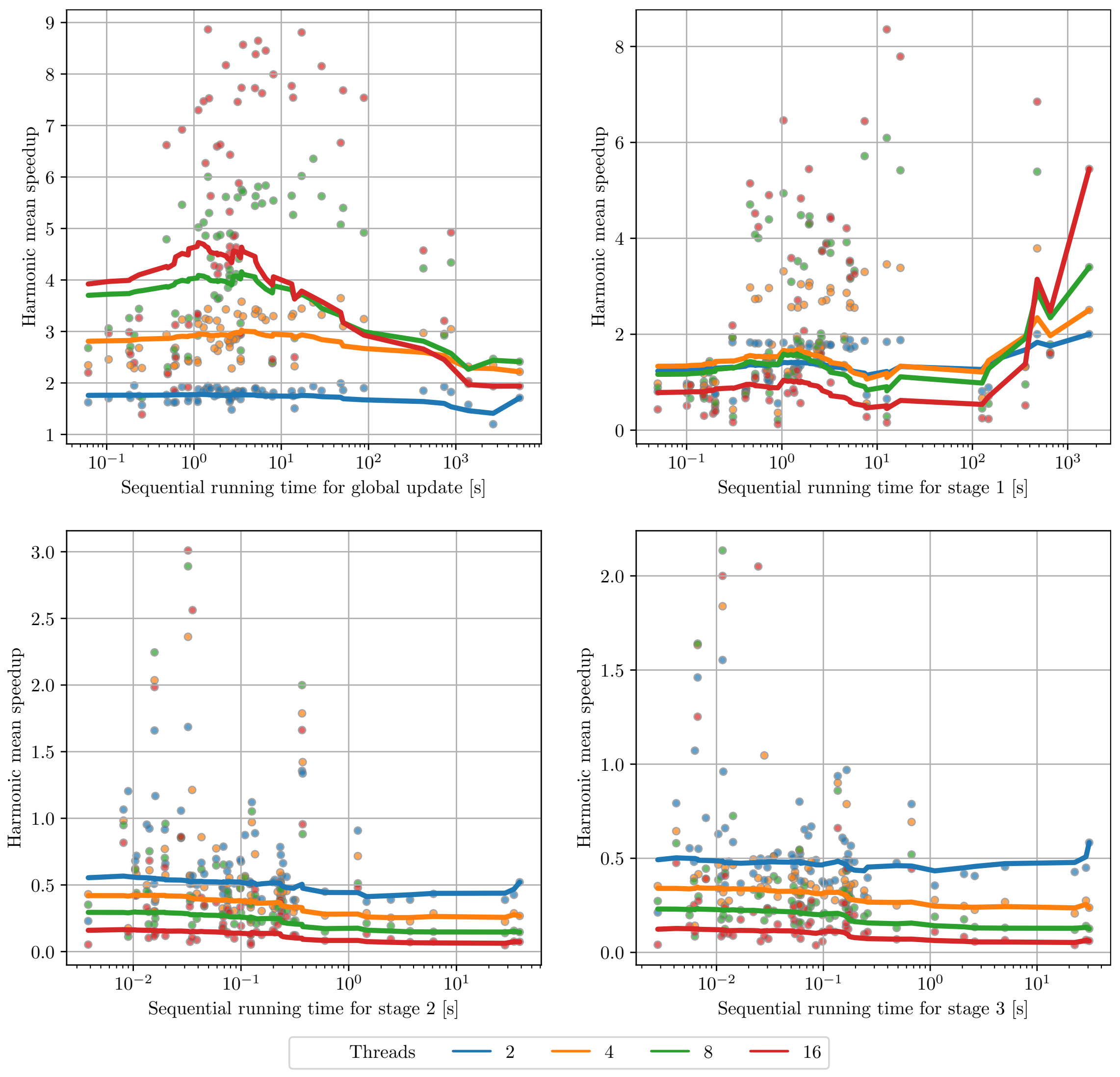

Figure 9.8.: Scalability plots for parallel parts of `seqExtraction-parPR`

other costly operations (except accessing the label and excess vectors). The more threads there are, the more thread-local vectors exist that we have to iterate over. It seems that it is this specific implementation technique with multiple vectors that shows disadvantages here. Because in stage 1 we also iterate over the vectors in parallel, but also do some additional memory accesses on the hypergraph, we can expect this stage to suffer from this behavior too.

Indeed, if we look at the speedup plots for the parallel phases in Figure 9.8, they seem to reinforce this hypothesis. To solve this problem, one could investigate better queue implementations.

9.4. Evaluation of partitioners

In Chapter 4 we introduced partitioning approaches that can use the WHFC refinement to improve bisections. We compare the derived partitioners, namely the Recursive Bisection, the k-way refinement with sequential scheduling and the k-way refinement with parallel scheduling. As an initial partitioner, we always use PaToH with the D preset. Our setup consists of the following partitioning configurations:

- `rb-seqExtraction-seqDinic`

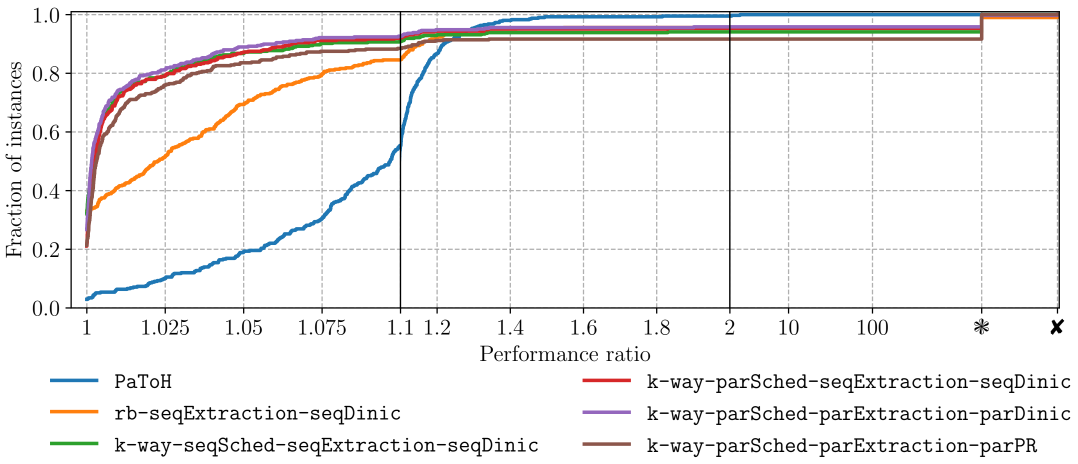

Figure 9.9.: Performance profiles comparing all partitioners

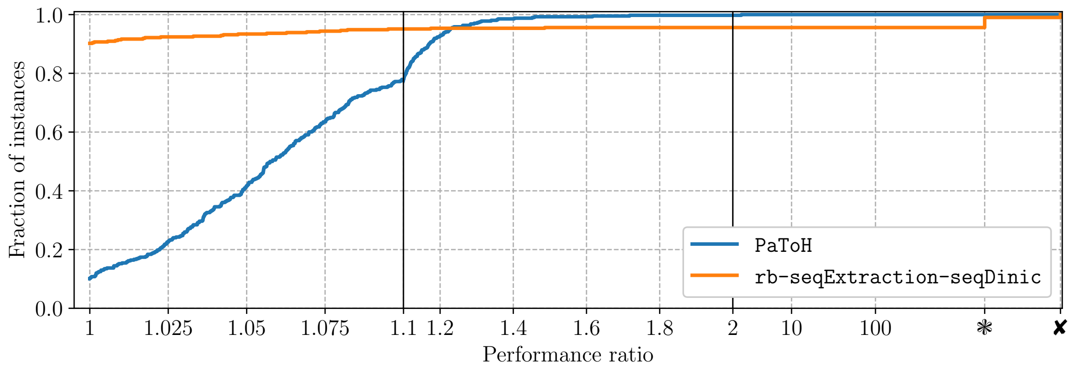

Figure 9.10.: Performance profiles comparing `PaToH` and sequential recursive bisection

- `k-way-seqSched-seqExtraction-seqDinic`
- `k-way-parSched-seqExtraction-seqDinic`
- `k-way-parSched-parExtraction-parDinic`
- `k-way-parSched-parExtraction-parPR`

`rb-seqExtraction-seqDinic` uses Algorithm 1 for the recursive bisection and sequential refinement with Dinic. Everything is executed sequentially. Although we described how the initial partitioner can be run in parallel after splitting the hypergraph, the interface for PaToH did not let us run multiple partitionings in parallel as they use the same predefined global datastructure. `k-way-seqSched-seqExtraction-seqDinic` uses the same sequential refinements with Dinic and schedules them sequentially as in Algorithm 2. The three `k-way-parSched-*` algorithms use the parallel scheduling from Algorithm 3 and employ different algorithms for the refinement.

We compare the quality of the algorithms in comparison to PaToH and evaluate absolute running times as well as speedups for the algorithms in the next sections.

9.4.1. Partitioning quality

First we compare the quality of the tested algorithms (and PaToH) using performance profiles. The profiles use the connectivity function as the quality metric and are depicted in Figure 9.9. Note that not all instances could be solved by every algorithm as some instances timed out. While it is clear that `PaToH` offers the worst quality because all other

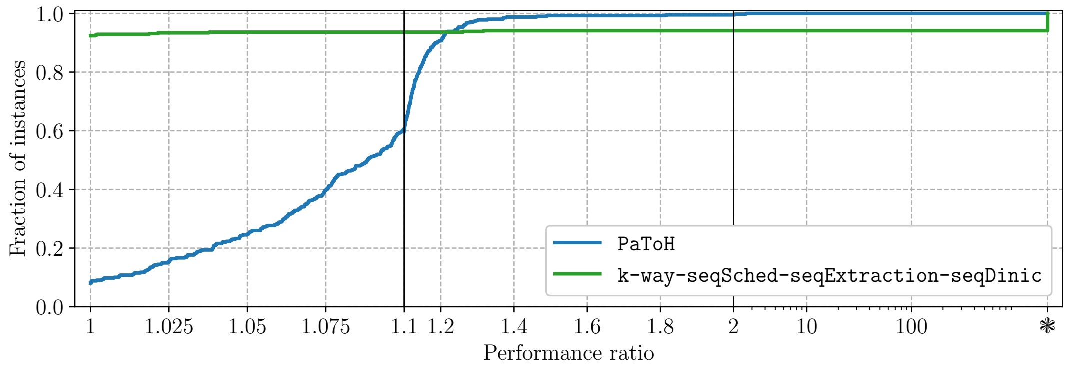

Figure 9.11.: Performance profiles comparing `PaToH` and sequential k-way refinement

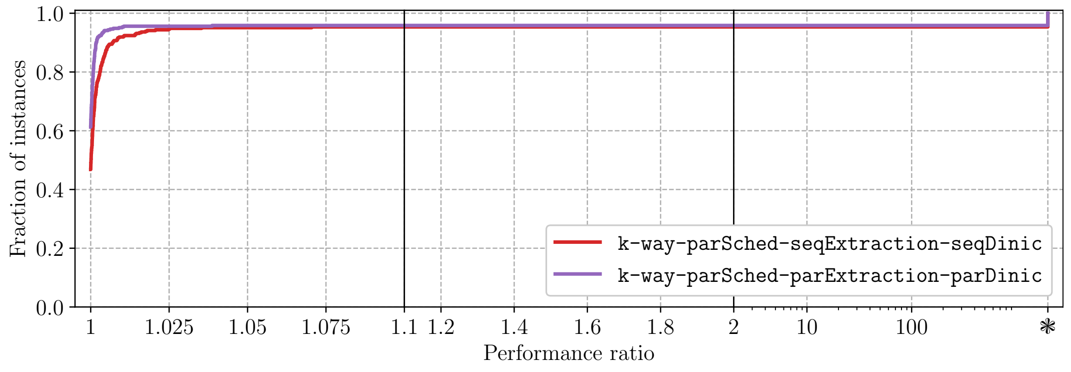

Figure 9.12.: Performance profiles comparing sequential and parallel extraction

algorithms improve the results obtained by it with refinement strategies, the comparison between Recursive Bisection and k-way refinement is more interesting. The algorithm that is used for the flow algorithm has no relevant impact on the quality of the partition, but the k-way refinement consistently dominates the recursive bisection by a fair amount. This could be due to the fact that after the recursive bisection splits the hypergraph and refines it, vertices are not moved between these two parts any more and the blocks they can be moved to is therefore restricted. In the k-way refinement, vertices can be moved freely between blocks.

Figure 9.10 shows a more detailed comparison between PaToH and recursive bisection, while Figure 9.11 shows a comparison between PaToH and the sequential k-way refinement. PaToH can only dominate the algorithms for instances that timed out. Apart from this, the quality is much better for the algorithms that use refinements.

We already showed that using sequential or parallel extraction can have an effect on the running time of the succeeding flow algorithm. However, we could not observe a big difference between the quality of the algorithms. Figure 9.12 shows the performance profiles for sequential and parallel extraction. The parallel version has only a very slight edge over the sequential implementation.

9.4.2. Running times of partitioners

Figure 9.13 shows the total running times of the partitioners executed with one thread, Figure 9.14 shows the same partitioners executed with four threads. When run sequentially, the running time of the k-way refinement is slightly worse than the recursive bisection, but four threads suffice to make the k-way refinement using Dinic faster than the Recursive Bisection. The overhead of the parallel scheduling in comparison to the sequential scheduling

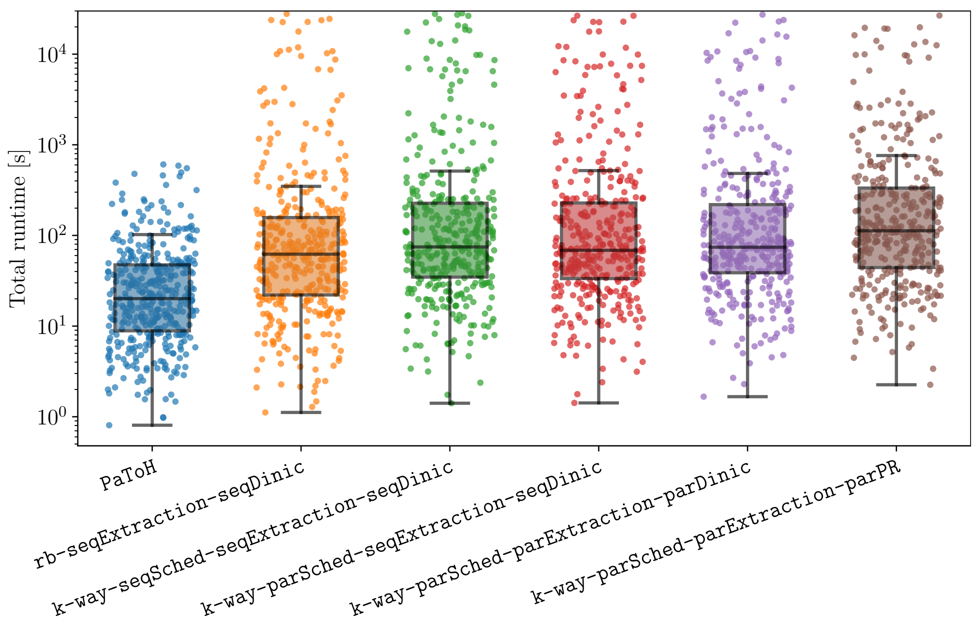

Figure 9.13.: Running times for partitioners executed with one thread

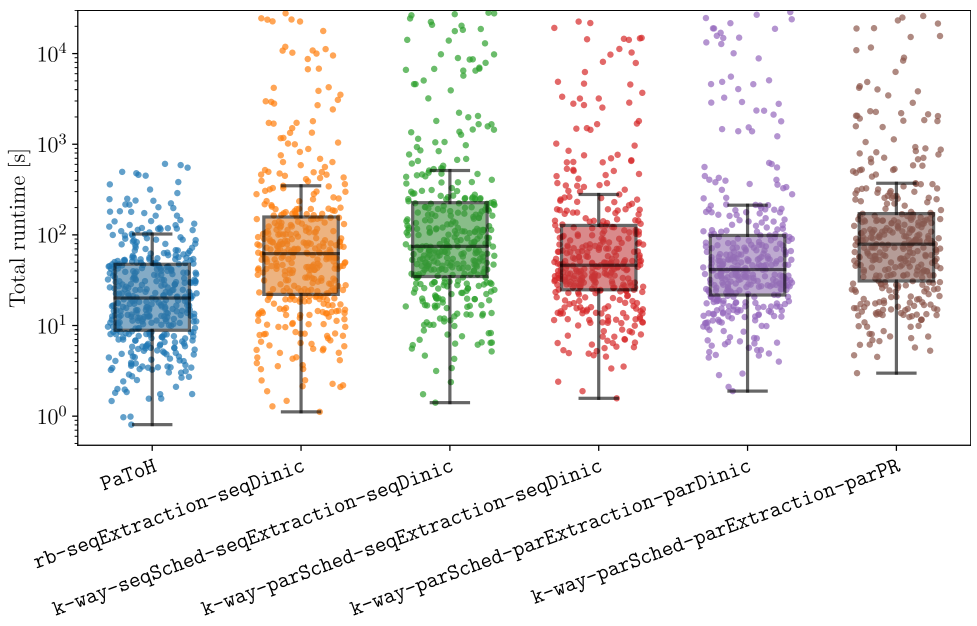

Figure 9.14.: Running times for partitioners executed with four threads

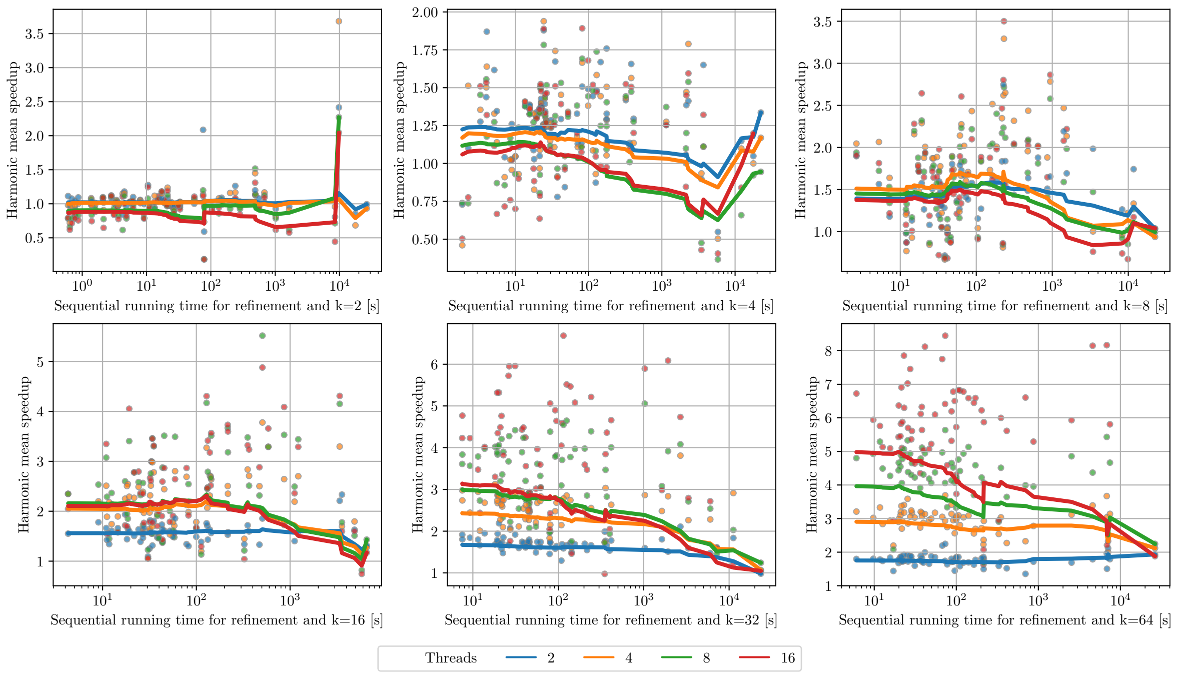

Figure 9.15.: Scalability plot for `k-way-parSched-seqExtraction-seqDinic`

in the k-way refinement is marginal and the parallel search and scheduling of new block pairs only accounts for a negligible fraction of the total running time. The performance of the k-way refinement using the Push-Relabel algorithm stands out and is worse than that of the Dinic variants. We will give some reasons in the next sections.

9.4.3. Scalability of the k-way partitioners

The scalability plots for the running times of the refinement (=total running time without the initial partitioning with PaToH) with parallel scheduling split for the different values of k are shown in Figure 9.15 for sequential extraction and sequential Dinic, Figure 9.16 shows the plot for parallel extraction and parallel Dinic and Figure 9.17 for parallel extraction and parallel Push-Relabel.

We start with the evaluation for the sequential extraction and sequential Dinic. For $k = 2$, no refinement steps can be executed in parallel, which is why the speedup stays around 1. For $k = 4$, at most 2 refinements can be executed in parallel and we can observe a small speedup for smaller instances. However, some bigger instances show worse performance. This is also true for the other values of k: Speedups for bigger instances get worse for sequential running times greater than around 100 seconds. Up to this running time, results look promising and get better with increasing k as more refinements can be executed in parallel.

The problem that bigger instances cause problems in terms of speedup is smaller for the parallel implementations of extraction and Dinic. While the general behavior looks similar, the speedups are consistently better. This can be attributed to the greater amount of parallelization and the fact that even towards the end of the partitioning, when there are not enough block pairs left to assign one to every thread, the threads can still contribute by running single refinements in parallel.

For the parallel Push-Relabel implementation, the results differ. Speedups are not as good as for the parallel Dinic implementation, but the Push-Relabel algorithm seems to handle

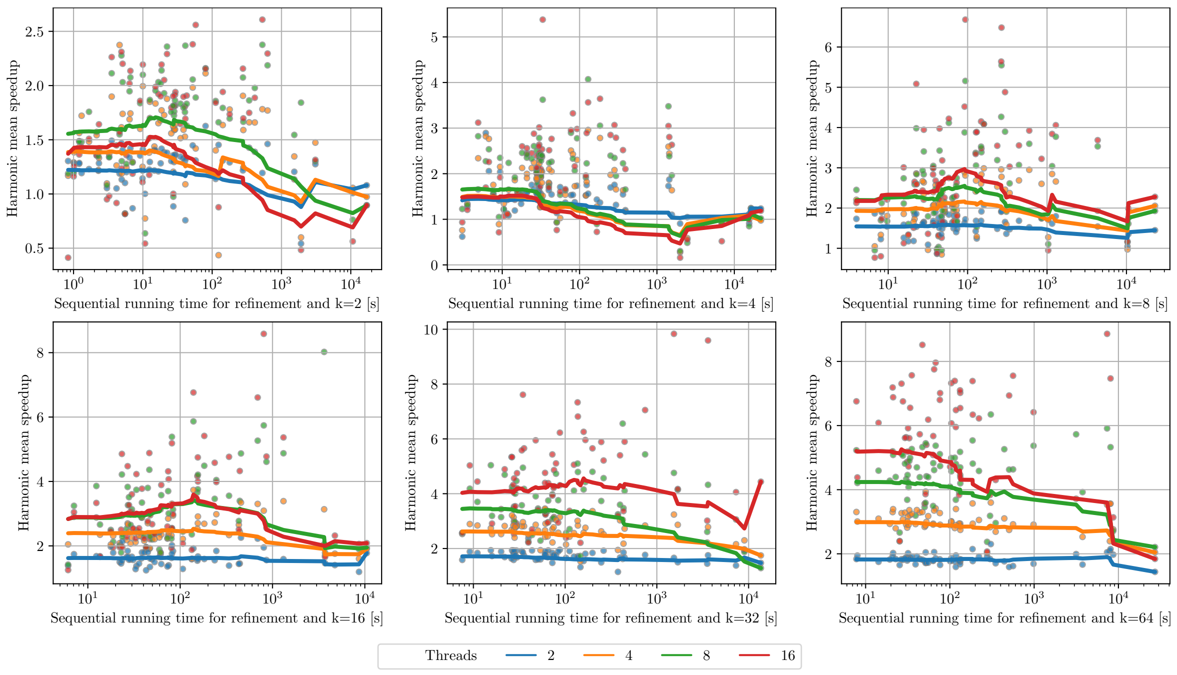

Figure 9.16.: Scalability plot for `k-way-parSched-parExtraction-parDinic`

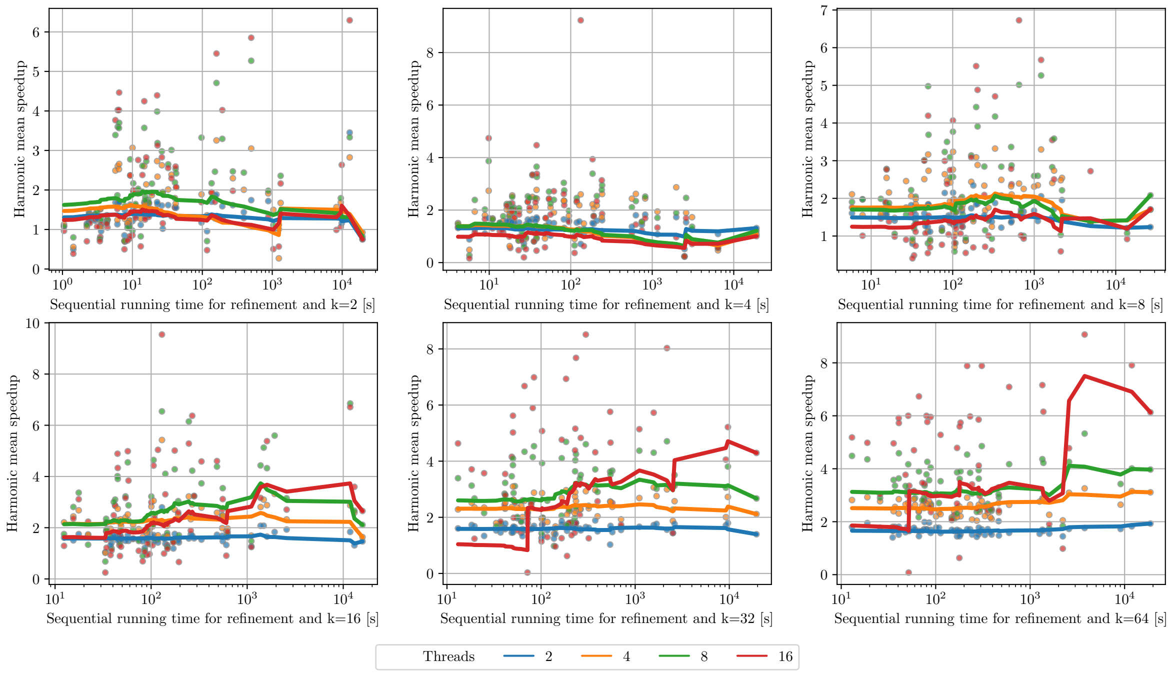

Figure 9.17.: Scalability plot for `k-way-parSched-parExtraction-parPR`

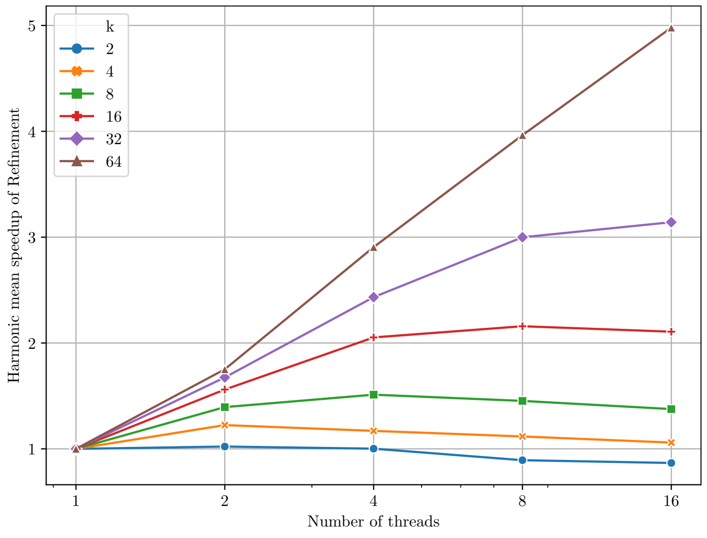

Figure 9.18.: Scalability summary for `k-way-parSched-seqExtraction-seqDinic`

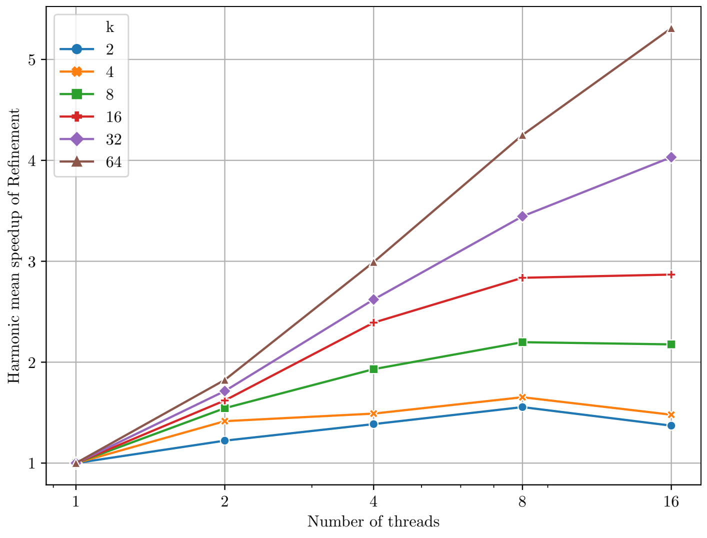

Figure 9.19.: Scalability summary for `k-way-parSched-parExtraction-parDinic`

bigger instances better. This makes it interesting to consider the algorithm for these graphs, although the running time with Push-Relabel is still slower than the others as discussed before.

To have a better overview of how the speedups behave for different values of k and number of threads, we consider Figure 9.18 for sequential extraction and Dinic. The plot depicts the harmonic mean speedup of the refinement for values of k. The plotted values correspond directly to the leftmost harmonic mean values of the subplots in Figure 9.15. We can see clearly that greater values of k lead to better speedups as more refinements can be executed in parallel. The same kind of plot for the parallel scheduling with parallel extraction and Dinic is shown in Figure 9.19. As we have already been able to tell from the scalability plots before, the speedup values are better while the general behavior stays the same. For the parallel Push-Relabel implementation in Figure 9.20, the plot shows very clearly that the algorithm suffers from a greater thread count. We already saw this effect in Section 9.3.3 when evaluating the algorithm on its own and came to the conclusion that the way we use thread-local vectors causes problems for a bigger thread count. It can be expected that solving this issue will also improve the speedups that we observe here.

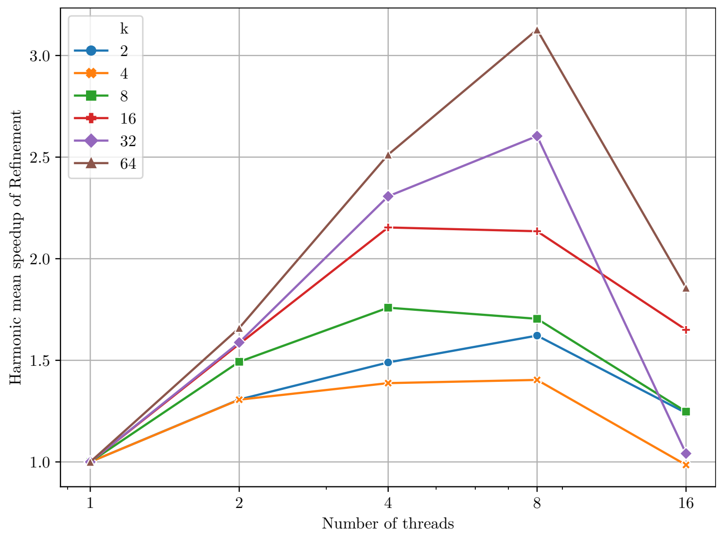

Figure 9.20.: Scalability summary for `k-way-parSched-parExtraction-parPR`

10. Conclusion

The experimental evaluation of the implemented algorithms showed that the parallelization of extraction and flow algorithms are viable approaches to speed up the refinement procedure of WHFC. While the extraction only makes up for a small amount of the running time, we gained bigger speedups by parallelizing the building of the layered network of the Dinic algorithm. However, we observed that changing the extraction and layered network procedures can have subtle effects on other parts of the algorithm. We didn't succeed at completely parallelizing the Dinic algorithm because the search for augmented paths didn't offer much possibilities to us. Instead, we implemented a sequential and parallel version of a Push-Relabel algorithm. The sequential Push-Relabel algorithm is competitive with the sequential Dinic implementation and outperforms it on some instances, even having a better mean running time on the instances we tested. The parallel Push-Relabel algorithm also stays competitive but does not scale as well as the parallel Dinic. We determined that the reason to be the usage of thread-local vectors in our implementation which is a good point to tackle for better performance and may offer promising running times.

Testing the parallelizations of WHFC in the context of complete (parallel) partitioners showed that even if refinements are scheduled in parallel, we can still improve running times and speedups by also using the parallel WHFC. This was the case when we compared sequential and parallel Dinic implementations. The parallel Push-Relabel algorithm suffers from the same scalability problems as before. In terms of quality, the flow-based refinement proved to be very effective, by far exceeding the quality of plain PaToH. Recursive bisection performed worse than the k-way refinement (comparing the quality) and is also not considerably faster, making it a bad trade-off in our use case. However, we were not able to test parallel partitioning with recursive bisection due to some limitations of the PaToH library and due to the independence of the parallel steps in the recursive bisection we can expect good speed-ups there.

Considering future work and improvements, there are other things that can be tried. For one, the parallel k-way scheduling of block pairs can be improved. When we schedule new block pairs, we take the first block from the *blocks* list with a maximum number of participations and search for another block that we can schedule it with. We could use a secondary criterion here to decide which of the blocks with a maximum number of participations to take. One could also rethink the criteria for how often and which block pairs to refine and introduce new schemes that use less refinement steps but still result in a partition with good quality. Another tie-braking criteria can be introduced for deciding which endpoint of an edge in the Lawler network wins the edge. In our implementation, the node id is used as the last tie-breaker and the node with the lower one wins the edge. Because of the way we assigned the ids in the Lawler network (first regular nodes, then edge-in nodes, then edge-out nodes) an edge-in node always wins over edge-in and edge-out nodes and edge-in nodes win over edge-out nodes. It is not clear whether this is favorable. We could for example introduce a hash function that lets us make a deterministic coin toss [SUV17] which results in the same node winning no matter from which endpoint we compute it. However, this is a very fine-grained improvement.

A more general improvement could be made by optimizing or using different flow algorithms for the max-flow computation. Maybe there is a way to nicely parallelize the computation of augmenting paths for the Dinic algorithm that we did not think of.

Lastly, it would also be possible to combine the two refinement approaches and first use recursive bisection (together with flow based refinement) to obtain an initial partition and use the k-way refinement afterwards, thus further improving the quality of the partitioning.

Bibliography

[AK95] Charles J. Alpert and Andrew B. Kahng. Recent Directions in Netlist Partitioning: A Survey. *Integration: The VLSI Journal*, 19(1-2):1–81, 1995.

[AMO93] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1993.

[AS95] Richard Anderson and Joao C. Setubal. A parallel implementation of the push-relabel algorithm for the maximum flow problem. *Journal of parallel and distributed computing*, 29(1):17–26, 1995.

[BBS15] Niklas Baumstark, Guy Blelloch, and Julian Shun. Efficient implementation of a synchronous parallel push-relabel algorithm. In *Algorithms-ESA 2015*, pages 106–117. Springer, 2015.

[BDHJ14] Anton Belov, Daniel Diepold, Marijn JH Heule, and Matti Järvisalo. Proceedings of sat competition 2014. 2014.

[BJ92] Thang Nguyen Bui and Curt Jones. Finding good approximate vertex and edge partitions is np-hard. *Information Processing Letters*, 42(3):153–159, 1992.

[BK04] Yuri Boykov and Vladimir Kolmogorov. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *IEEE transactions on pattern analysis and machine intelligence*, 26(9):1124–1137, 2004.

[BM11] Aydin Buluç and Kamesh Madduri. Parallel breadth-first search on distributed memory systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2011.

[BMS$^+$16] Aydın Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. Recent advances in graph partitioning. In *Algorithm Engineering*, pages 117–158. Springer, 2016.

[CA95] Ümit V Catalyürek and Cevdet Aykanat. Hypergraph model for mapping repeated sparse matrix-vector product computations onto multicomputers. 1995.

[CA99] Umit V Catalyurek and Cevdet Aykanat. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Transactions on parallel and distributed systems*, 10(7):673–693, 1999.

[CG95] Boris V Cherkassky and Andrew V Goldberg. On implementing push-relabel method for the maximum flow problem. In *International Conference on Integer Programming and Combinatorial Optimization*, pages 157–171. Springer, 1995.

[Che94a] Boris V. Cherkassky. A Fast Algorithm for Constructing a Maximum Flow Through a Network. In *Selected Topics in Discrete Mathematics: Proceedings of the Moscow Discrete Mathematics Seminar, 197*, volume 158 of *American Mathematica*, pages 23–30. AMS, 1994.

[Che94b] Boris V Cherkassky. A fast algorithm for computing maximum flow in a network. *Collected Papers*, 3:90–96, 1994.

[ÇKU11] Ümit V Çatalyürek, Kamer Kaya, and Bora Uçar. Integrated data placement and task assignment for scientific workflows in clouds. In *Proceedings of the fourth international workshop on Data-intensive distributed computing*, pages 45–54, 2011.

[DH11] Timothy A Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1–25, 2011.

[Din70] Yefim Dinitz. Algorithm for Solution of a Problem of Maximum Flow in a Network with Power Estimation. *Soviet Mathematics-Doklady*, 11(5):1277–1280, September 1970.

[Din06] Yefim Dinitz. Dinitz'algorithm: The original version and even's version. In *Theoretical computer science*, pages 218–240. Springer, 2006.

[DM89] Ulrich Derigs and Wolfgang Meier. Implementing goldberg's max-flow-algorithm—a computational investigation. *Zeitschrift für Operations Research*, 33(6):383–403, 1989.

[DM02] Elizabeth D Dolan and Jorge J Moré. Benchmarking optimization software with performance profiles. *Mathematical programming*, 91(2):201–213, 2002.

[DSW14] Julian Dibbelt, Ben Strasser, and Dorothea Wagner. Customizable contraction hierarchies. In *International Symposium on Experimental Algorithms*, pages 271–282. Springer, 2014.

[EK72] Jack Edmonds and Richard M. Karp. Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems. *Journal of the ACM*, 19(2):248–264, April 1972.

[FF56] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

[FJF15] Lester Randolph Ford Jr and Delbert Ray Fulkerson. *Flows in networks.* Princeton university press, 2015.

[FM82] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proceedings of the 19th ACM/IEEE Conference on Design Automation*, pages 175–181, 1982.

[GHK$^+$11] Andrew V Goldberg, Sagi Hed, Haim Kaplan, Robert E Tarjan, and Renato F Werneck. Maximum flows by incremental breadth-first search. In *European Symposium on Algorithms*, pages 457–468. Springer, 2011.

[GHK$^+$15] Andrew V Goldberg, Sagi Hed, Haim Kaplan, Pushmeet Kohli, Robert E Tarjan, and Renato F Werneck. Faster and more dynamic maximum flow by incremental breadth-first search. In *Algorithms-ESA 2015*, pages 619–630. Springer, 2015.

[GHSS20] Lars Gottesbüren, Tobias Heuer, Peter Sanders, and Sebastian Schlag. Scalable shared-memory hypergraph partitioning. *arXiv preprint arXiv:2010.10272*, 2020.

[GHSW20] Lars Gottesbüren, Michael Hamann, Sebastian Schlag, and Dorothea Wagner. Advanced Flow-Based Multilevel Hypergraph Partitioning. In Simone Faro and Domenico Cantone, editors, *18th International Symposium on Experimental*

Algorithms (SEA 2020), volume 160 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 11:1–11:15, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[GHW19] Lars Gottesbüren, Michael Hamann, and Dorothea Wagner. Evaluation of a Flow-Based Hypergraph Bipartitioning Algorithm. In Michael A. Bender, Ola Svensson, and Grzegorz Herman, editors, *27th Annual European Symposium on Algorithms (ESA 2019)*, volume 144 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 52:1–52:17, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[Glo86] Fred Glover. Future paths for integer programming and links to ar tifi cial intelli g en ce. *Computers operations research*, 13(5):533–549, 1986.

[GT88] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM*, 35(4):921–940, 1988.

[GT14] Andrew V Goldberg and Robert E Tarjan. Efficient maximum flow algorithms. *Communications of the ACM*, 57(8):82–89, 2014.

[GTT89] Andrew V Goldberg, Éva Tardos, and Robert E Tarjan. Network flow algorithms. Technical report, PRINCETON UNIV NJ DEPT OF COMPUTER SCIENCE, 1989.

[HH10] Bo Hong and Zhengyu He. An asynchronous multithreaded algorithm for the maximum network flow problem with nonblocking global relabeling heuristic. *IEEE Transactions on Parallel and Distributed Systems*, 22(6):1025–1033, 2010.

[HL95] Bruce Hendrickson and Robert Leland. A Multilevel Algorithm for Partitioning Graphs. In *Proceedings of the 1995 ACM/IEEE conference on Supercomputing (SC'05)*, page 28. ACM Press, 1995.

[Hoc08] Dorit S Hochbaum. The pseudoflow algorithm: A new algorithm for the maximum-flow problem. *Operations research*, 56(4):992–1009, 2008.

[Hon08] Bo Hong. A lock-free multi-threaded algorithm for the maximum flow problem. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–8. IEEE, 2008.

[HS18] Michael Hamann and Ben Strasser. Graph bisection with pareto optimization. *Journal of Experimental Algorithmics (JEA)*, 23:1–34, 2018.

[HSS19] Tobias Heuer, Peter Sanders, and Sebastian Schlag. Network flow-based refinement for multilevel hypergraph partitioning. *Journal of Experimental Algorithmics (JEA)*, 24(1):1–36, 2019.

[HYW11] Felix Halim, Roland HC Yap, and Yongzheng Wu. A mapreduce-based maximum-flow algorithm for large small-world network graphs. In *2011 31st International Conference on Distributed Computing Systems*, pages 192–202. IEEE, 2011.

[HZY15] Jin Huang, Rui Zhang, and Jeffrey Xu Yu. Scalable hypergraph learning and processing. In *2015 IEEE International Conference on Data Mining*, pages 775–780. IEEE, 2015.

[KAKS99] George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. Multilevel hypergraph partitioning: applications in vlsi domain. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 7(1):69–79, 1999.

[Kar74] Alexander V. Karzanov. Determining the Maximal Flow in a Network by the Method of Preflows. *Soviet Mathematics-Doklady*, 15(2):434–437, 1974.

[KKP$^+$17] Igor Kabiljo, Brian Karrer, Mayank Pundir, Sergey Pupyrev, Alon Shalita, Alessandro Presta, and Yaroslav Akhremtsev. Social hash partitioner: a scalable distributed hypergraph partitioner. *arXiv preprint arXiv:1707.06665*, 2017.

[KL70] Brian W. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *Bell System Technical Journal*, 49(2):291–307, February 1970.

[Law73] Eugene L. Lawler. Cutsets and Partitions of hypergraphs. *Networks*, 3:275–285, 1973.

[Len90] Thomas Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout*. Wiley, 1990.

[LK13] Dominique LaSalle and George Karypis. Multi-threaded graph partitioning. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 225–236. IEEE, 2013.

[LK16] Dominique LaSalle and George Karypis. A parallel hill-climbing refinement algorithm for graph partitioning. In *2016 45th International Conference on Parallel Processing (ICPP)*, pages 236–241. IEEE, 2016.

[Phe08] Chuck Pheatt. Intel® threading building blocks. *Journal of Computing Sciences in Colleges*, 23(4):298–298, 2008.

[QD84] Michael J Quinn and Narsingh Deo. Parallel graph algorithms. *ACM Computing Surveys (CSUR)*, 16(3):319–348, 1984.

[RAK07] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Physical review E*, 76(3):036106, 2007.

[RC78] Eshrat Reghbati and Derek G. Corneil. Parallel computations in graph theory. *SIAM Journal on Computing*, 7(2):230–237, 1978.

[RPG96] Erik Rolland, Hasan Pirkul, and Fred Glover. Tabu search for graph partitioning. *Annals of operations research*, 63(2):209–232, 1996.

[SHH$^+$16] Sebastian Schlag, Vitali Henne, Tobias Heuer, Henning Meyerhenke, Peter Sanders, and Christian Schulz. K-way hypergraph partitioning via n-level recursive bisection. In *2016 Proceedings of the Eighteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 53–67. SIAM, 2016.

[SS11] Peter Sanders and Christian Schulz. Engineering multilevel graph partitioning algorithms. In *European Symposium on Algorithms*, pages 469–480. Springer, 2011.

[ST06] Peter Sanders and Jesper Larsson Träff. Parallel prefix (scan) algorithms for mpi. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pages 49–57. Springer, 2006.

[SUV17] Alexander Shen, Vladimir A Uspensky, and Nikolay Vereshchagin. *Kolmogorov complexity and algorithmic randomness*, volume 220. American Mathematical Soc., 2017.

[SV82] Yossi Shiloach and Uzi Vishkin. An o (n2log n) parallel max-flow algorithm. *Journal of Algorithms*, 3(2):128–146, 1982.

[Tri06] Aleksandar Trifunovic. *Parallel algorithms for hypergraph partitioning.* PhD thesis, University of London, 2006.

[UB13] Johan Ugander and Lars Backstrom. Balanced label propagation for partitioning massive graphs. In *Proceedings of the sixth ACM international conference on Web search and data mining*, pages 507–516, 2013.

[VAS+12] Natarajan Viswanathan, Charles Alpert, Cliff Sze, Zhuo Li, and Yaoguang Wei. The dac 2012 routability-driven placement contest and benchmark suite. In *DAC Design Automation Conference 2012*, pages 774–782. IEEE, 2012.

Appendix

A. Set of hypergraphs used for the experiments

```
af_shell10.mtx.hgr
af_shell6.mtx.hgr
boneS10.mtx.hgr
Bump_2911.mtx.hgr
circuit5M.mtx.hgr
CurlCurl_4.mtx.hgr
dac2012_superblue11.hgr
dac2012_superblue12.hgr
dac2012_superblue14.hgr
dac2012_superblue16.hgr
dac2012_superblue19.hgr
dac2012_superblue2.hgr
dac2012_superblue3.hgr
dac2012_superblue6.hgr
dac2012_superblue7.hgr
dac2012_superblue9.hgr
dgreen.mtx.hgr
dielFilterV2clx.mtx.hgr
dielFilterV2real.mtx.hgr
dielFilterV3clx.mtx.hgr
dielFilterV3real.mtx.hgr
Emilia_923.mtx.hgr
fem_hifreq_circuit.mtx.hgr
Flan_1565.mtx.hgr
Freescale1.mtx.hgr
FullChip.mtx.hgr
Ga41As41H72.mtx.hgr
Geo_1438.mtx.hgr
gsm_106857.mtx.hgr
inline_1.mtx.hgr
ldoor.mtx.hgr
msdoor.mtx.hgr
nlpkkt80.mtx.hgr
```

```
rajat31.mtx.hgr
RM07R.mtx.hgr
sat14_11pipe_k.cnf.dual.hgr
sat14_11pipe_k.cnf.hgr
sat14_11pipe_k.cnf.primal.hgr
sat14_9vliw_m_9stages_iq3_C1_bug8.cnf.dual.hgr
sat14_9vliw_m_9stages_iq3_C1_bug8.cnf.hgr
sat14_9vliw_m_9stages_iq3_C1_bug8.cnf.primal.hgr
sat14_atco_enc3_opt1_04_50.cnf.dual.hgr
sat14_atco_enc3_opt1_04_50.cnf.hgr
sat14_atco_enc3_opt1_04_50.cnf.primal.hgr
sat14_atco_enc3_opt2_05_21.cnf.dual.hgr
sat14_atco_enc3_opt2_05_21.cnf.hgr
sat14_atco_enc3_opt2_05_21.cnf.primal.hgr
sat14_blocks-blocks-37-1.130-NOTKNOWN.cnf.dual.hgr
sat14_blocks-blocks-37-1.130-NOTKNOWN.cnf.hgr
sat14_blocks-blocks-37-1.130-NOTKNOWN.cnf.primal.hgr
sat14_SAT_dat.k100-24_1_rule_1.cnf.dual.hgr
sat14_SAT_dat.k100-24_1_rule_1.cnf.hgr
sat14_SAT_dat.k100-24_1_rule_1.cnf.primal.hgr
sat14_SAT_dat.k100-24_1_rule_2.cnf.dual.hgr
sat14_SAT_dat.k100-24_1_rule_2.cnf.hgr
sat14_SAT_dat.k100-24_1_rule_2.cnf.primal.hgr
sat14_SAT_dat.k95-24_1_rule_3.cnf.dual.hgr
sat14_SAT_dat.k95-24_1_rule_3.cnf.hgr
sat14_SAT_dat.k95-24_1_rule_3.cnf.primal.hgr
sat14_velev-npe-1.0-9dlx-b71.cnf.dual.hgr
sat14_velev-npe-1.0-9dlx-b71.cnf.hgr
sat14_velev-npe-1.0-9dlx-b71.cnf.primal.hgr
ss.mtx.hgr
StocF-1465.mtx.hgr
vas_stokes_1M.mtx.hgr
vas_stokes_2M.mtx.hgr
vas_stokes_4M.mtx.hgr
wb-edu.mtx.hgr
```