High Quality Hypergraph Partitioning via Max-Flow-Min-Cut Computations Presentation-Script

Tobias Heuer

Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany.

Slide 1 Hello and welcome, I am Tobias Heuer and today I am here to talk about my master thesis with the topic *High Quality Hypergraph Partitioning via Max-Flow-Min-Cut Computations*. More precisely, my task was to develop a refinement algorithm to improve partitions of hypergraphs based on flow computations and integrate it into the n-level hypergraph partitioner *KaHyPar*. The well known *Max-Flow-Min-Cut* theorem establish an analogy between the minimum cut separating two vertices in a network and the maximum flow between them. Algorithms to solve the maximum flow problem are usually computationally expensive. A major goal of my work was to outperform the latest version of *KaHyPar* on most of our benchmark instances and simultaneously ensure that the running time remaining competitive.

Slide 2 Hypergraphs are generalizations of graphs where each hyperedge can connect more than two hypernodes. A hypergraph consists of a set of vertices, a set of hyperedges, where each hyperedge is a subset of the vertex set and a node and edge weight function. A hyperedge is also called net and vertex inside a net is called pin. The number of pins is the sum of the sizes of each net or the sum of the degree of each vertex.

Slide 3 The hypergraph partitioning problem is to partition the vertices of a hypergraph into k non-empty disjoint blocks such that each block is smaller than $1 + \epsilon$ times the average block size. Simulaneously we want to minimize an objective function. In this talk we focus on the connectivity metric. Each hyperedge contributes $(\lambda - 1)$ times the edge weight to that function, where λ is the number of blocks connected by net e. Consequently, the goal is to minimize the number of blocks inside a net e.

Slide 4 Applications of hypergraph partitioning can be found in the area of VLSI Design. The goal in VLSI design is to partition a circuit into smaller units such that the wires between the gates are as short as possible. A wire can connect more than two gates, therefore a hypergraph models a circuit more accurately than a graph.
Another application can be found in the parallization of the sparse matrix-vector multiplication. We can interprete a sparse matrix as a hypergraph. The goal is to partition the sparse matrix into equal sized parts such that the communication during parallel computation of the multiplication is minimized.

Slide 5 The hypergraph partitioning problem is NP-hard. The most common heuristic used in all modern hypergraph partitioners is the *multilevel paradigm*. First, a sequence of smaller hypergraphs is generated by contracting matchings or clusterings between vertices. If the hypergraph is small enough, expensive heuristics are used to calculate an *initial partition* of the hypergraph into k blocks. Afterwards, the sequence of smaller hypergraphs is uncontracted in reverse order of contraction and the partition is projected to the next level finer hypergraph. After each uncontraction an *local search* algorithm is used to improve the hypergraph partition. Our *Max-Flow-Min-Cut* framework will work as a local search algorithm in this multilevel context.

Slide 6 All modern hypergraph partitioners implements varations of the *FM* algorithm as local search algorithm. In general, this is a move-based heuristic that greedily move vertices between blocks based on local informations of incident nets. In this example we want to move the red hypernode from block 4 to block 3. The move reduces the cut of the partition by 1, because the light blue net is removed from cut. The reduction of the objective function when moving a node is also called gain. The *FM* heuristics moves a vertex with maximum gain in each step.

Slide 7 The *FM* heuristics is often critized, because it only incorparates local informations about the problem structure. If we want to move the red node in this example only the state of the colored nets is considered in the gain calculation. The quality of the final partition therefore heavily depends on the initial partition. In the multilevel context this problem is partially solved, because a move of a node corresponds to a move of a subset of the nodes in the original hypergraph, but the quality depends more on the quality of the coarsening phase.
Moreover, large hyperedges induce zero-gain moves. In this example, we have to perform many zero-gain moves until a single move of a vertex finally contributes to the gain function. In such situation the quality of the partition mainly depends on random choices made within the algorithm.
The motivation behind the usage of flow computations is that such an approach considers the global structure of the problem by finding a minimum-cut separating two vertices.

Slide 8 Before we finally start with our main work, we need some basic terminology about network flows. Given a graph with a node and edge set and a capacity function u and two vertices s and t which are also called source and sink. A flow is a function f which satisfy the following two constraints:

- The flow on an edge has to be less or equal to the capacity of that edge
- And the amount of flow entering a node is the same as leaving the node.

The value of the flow is the amount of flow entering the sink resp. leaving the source. The maximum flow problem is to find a maximum flow f from s to t such that no other flow function exists which value is greater than our maximum flow. An important concept to solve the maximum flow problem is the concept of the residual graph, which contains each edge with an residual capacity greater than zero. The residual capacity is the remaining amount of flow which we can send over an edge. On the reverse edge we say that the residual capacity is equal with the flow on the forward edge. An augmenting path is a path from s to t in the residual graph. One can show that f is a maximum flow, if there exists no augmenting path.

Slide 9 We can calculate the corresponding minimum bipartition separating s and t by assinging each node reachable from the source s in the residual graph to block 1 and all remaining to block 2.

Slide 10 Our flow-based reinfement framework is motivated by the results of Sanders and Schulz, who successfully integrate such an approach in their multilevel graph partitioner *KaFFPa*. We generalize many results of their work such that they applicable to hypergraph partitioning. In the following, we explain their basic ideas and highlight our contributions in different parts of the framework.
Our framework consists of four basic building blocks. We perform a flow-based refinement on two adjacent blocks of a k-way partition. Therefore, we need a block selection strategy, which is the first part of our framework. In the second part, we have to build a flow problem for our selected bipartition. Therefore, we have to add hypernodes to the flow problem and configure the source and sink set such that a *Max-Flow-Min-Cut* computations improves the cut of the corresponding

bipartition. In the next step, we have to solve the flow problem. Therefore, we have to transform the subhypergraph into a corresponding flow network such that each mininimum-capacity cutset of the flow network corresponds to a minimum-weight cutset of the hypergraph. Finally, we use informations of the maximum flow function to automatically balance our bipartition according to our balanced constraints. One can show that one maximum flow computation has enough informations to enumerate all minimum-capacity cutsets.
We will now start to explain more in detail our block selection strategy.

Slide 11 Our block selection strategy is called *active block scheduling.* Each block of the partition has a state *active* or *inactive.* Initially all blocks are *active.* The algorithm works in rounds and starts by building the quotient graph of the k-way partition, which contains an edge between each adjacent pair of blocks. Then, we use our flow-based refinement algorithm to improve the quality of the bipartition induced by two adjacent *active* blocks. We visit the edges in random order. If a refinement yield an improvement, we mark the two blocks as *active* for the next round. In the first round, we found e.g., an improvement on block 1 and 2 and on all remaining pairs of adjacent blocks, there is no improvement. If the boundary of a block did not change, we mark them as inactive for the next round. In the next round, we only consider edges where one of the two blocks is *active.* In this example, no refinement yield an improvement in the second round. Consequently, all blocks become *inactive* and the algorithm terminates.

Slide 12 Once we select two adjacent blocks, we have to build a flow problem, which we use to improve the corresponding bipartition.

Slide 13 Our flow problem is build around the cut of the bipartition induced by the two selected blocks. We start a BFS, initialized with all vertices of a cut net of block 1. The BFS stops if the weight of the nodes would exceed a predefined upper bound. The upper bound is chosen in such a way that if all hypernodes of B_1 are assigned to block 2 the bipartition is still feasible according to our balanced contraint. Afterwards, we start a second BFS initialized with all vertices of a cut net of block 2. Let us consider all nets which are now partially contained in the flow problem. To ensure that non of the non-cut edges on the boundary becomes a cut edge we add the vertices of the nets, which are contained into flow problem, to the source resp. sink. Thus, we ensure that a *Max-Flow-Min-Cut* computation never increase the cut of the bipartition.
An extension of that approach is the *adaptive flow iteration* strategy. The size of the flow problem depends now on ε' rather than on ε. Then the algorithm works as follows: If a flow computation yields an improvement we double the flow problem size, where α' is a predifined upper bound. If a flow computation yields no improvement, we half the flow problem size. If $\alpha = 1$ yields no improvement the algorithm terminates.

Slide 14 In the next step, we have to transform the subhypergraph extracted in the flow problem build step into a flow network.

Slide 15 Therefore, we consider the bipartite graph representation of that hypergraph. Each hypernode is connected to its incident hyperedges via an undirected edge. Hu and Moerder introduce node capacities in this graph representation. Each hypernode as a weight equal to infinity and each hyperedge node has weight equal to its corresponding weight in the hypergraph. The observation is that a minimum-cutset separating two vertices of the hypergraph is equal with a minimum-weight vertex separator. We can use the known vertex separator transformation to obtain the corresponding flow network. Each node of the bipartite graph is splitted into an incoming and outgoing node, which are connected via an directed edge with the weight of that node. Lawler showed that it is not neccasary to split the hypernodes because their weight is infinity. The node containing all incoming edges is called incoming hyperedge node and the node containing all outgoing edges is called outgoing hyperedge node.

Slide 16 If we go a step back and consider again the bipartite graph representation with its node capacities, we can make the following observation that a hypernode will never be part of a minimum-weight vertex separator, because their weight is infinity. Therefore, we remove all hypernodes by adding a clique between all incident hyperedges. Using the vertex separator transformation results in this network. A simple observation is that a hypernode induces 2 times the degree of that hypernode edges in the Lawler Network and the degree times the degree minus 1 edges in our network. We can reduce the number of nodes if the degree is smaller than 3. Therefore, we remove all low degree vertices in our flow network.

Slide 17 Summing up the flow network construction, we can find a minimum-weight cutset in the hypergraph with the Lawler-Network. There also exists an other optimization due to Liu and Wong, which models each hyperedge of size 2 as a graph edge. In our network we remove each hypernode with a degree smaller than 3 by adding a clique between the corresponding hypernodes. Finally, we combine our and the Wong Network in a Hybrid Network to combine the advantages of both.

Slide 18 We are now more familiar with the hypergraph flow network. Therefore, we consider again the flow problem modeling approach of Sanders and Schulz in *KaFFPa*. Each hypernode contained in a border hyperedges are added to the source resp. sink, which ensure that a non-cut edge did not become a cut edge after a *Max-Flow-Min-Cut* computation. This restricts the space of possible solutions, because non of the source resp. sinks vertices is moveable after a flow computation. We suggest an other flow problem modeling approach such that all vertices in the flow problem are moveable. Therefore, we extend the flow problem with all vertices contained in a border hyperedge and add those vertices to the source resp. sink. Now all vertices of our original flow problem are moveable, but the corresponding flow problem is significantly larger. Let us take a closer look at the structure of our modeling approach. All incoming and outgoing edges of a source vertex have a capacity equal to infinity. Therefore, we can remove all previously added vertices and add all incident incoming hyperedge nodes to the source. We can do the same for the sink nodes and add all incident outgoing hyperedge nodes to the sink.

Slide 19 In the last building block of our framework we try to find a feasible solution according to our balanced constraint by enumerating all minimum-cutsets.

Slide 20 This heuristic is called *most balanced minimum cut* and also used in *KaFFPa*. Picard and Queryanne showed that one maximum flow f has enough information to enumerate all minimum (s, t)-cuts. In the example, we have a flow graph on the left side and the residual graph of a maximum flow of that network on the right side. The algorithm works as follows: First we contract all *strongly connected components*. Afterwards, we find a topological order of the so called Picard-Queryanne *directed acyclic graph* and afterwards we sweep through the reverse topological order. The key observation is that each bisection of the topological order induce a minimum-cutset in the flow network with the same weight. Thus, we can use this algorithm to balance the bipartition according to our balanced constraints.

Slide 21 We use this framework as *local search* algorithm in one level of the multilevel hierarchy in *KaHyPar*. *KaHyPar* is a n-level hypergraph partitioner taking the multilevel paradigm to its extreme by contracting only a single vertex in each level. Consequently, during uncontraction n local searches are instantiated. Executing our flow-based refinement in each level would be too expensive. Therefore we introduce flow execution policies. One is to execute flows in each level i where i is a power of 2 or in each level which is multiple of β. Note, regardless of the policy we use flow-based refinement on the last level. In each level where no flow is executed we can use the classical *FM* algorithm.

Slide 22 Additionally, we implement several speed-up heuristics. During active block scheduling we only execute flow-based refinement between two adjacent block, if the cut has a weight greater than 10 except on the last level. Furthermore, we incorporate the improvement history of previous flow computations. The first round of active block scheduling works as before. In all remaining rounds we only use a pairwise flow-based refinment if a previous flow computation on the corresponding blocks yield an improvement. Finally, if no hypernode changed its block after a flow computation, we break the adaptive flow iteration strategy.

Slide 23 The last part of the presentation is the experimental evaluation. Our experiments a executed on the following system compiled with *g++ 5.2*. We integrated several maximum flow algorithms. We use two own and two third-party implementations. The fastest in our experiments was the IBFS algorithm of Goldberg et al. Therefore, all presented experiments use the IBFS maximum flow algorithm in the following. We use 3 different bechmark sets to evaluate our partitioner. The first consist of 25 hypergraphs and is used if we have to evaluate the performance of a large number of different configurations. If only a few configurations remain, we use our benchmark subset with 165 hypergraphs. If we compare our final configuration against other hypergraph partitioner we use our full benchmark set with 488 hypergraphs. The benchmarks can be splitted into 6 different benchmark types from 3 different application areas. More precisely, we have benchmarks from VLSI Design, SAT Solving and Sparse Matrices. In our general experiments, we partition a hypergraph instance into 7 different types of k with an imbalance of 3% and 10 different seeds.

Slide 24 We evaluate the impact of our sparsification techniques on our benchmark subset. In this plot we can see the speed-up of the IBFS maximum flow algorithm on the Wong and Hybrid Network relative to the execution on the Lawler Network on the different benchmark types. Remember, in the Wong Network we model each hyperedge of size 2 as graph edge and the Hybrid Network is a combination of the Wong and our network where we remove each hypernode with a degree smaller than 3. We observe that the IBFS algorithm is up to factor between 1.5 and 2 faster on the Wong Network than on the Lawler Network except on the DUAL and SPM instances. The DUAL instances are low degree hypergraphs. Therefore, our Hybrid Network is up to factor of 1.7 faster. Our Hybrid Network performs at least as good as the Wong Network. Therefore, we choose the Hybrid Network for further experiments.

Slide 25 In the next experiment we evaluate the quality of different configurations of our framework. A configuration is denoted with a triple, which indicate the heuristics enabled or disabled. We have three different heuristics: Flows, *Most Balanced Minimum Cut* and the FM heuristic. All configurations are executed with the exponential flow execution policy except CONSTANT128 which uses the constant flow execution policy on each 128-th level with all heuristics enabled. This configuration should give us a lower bound on the quality achievable with *Max-Flow-Min-Cut* computations. Each configuration is executed for different values of α' which determine the maximum flow problem size. All values are relative to the lastest configuration of *KaHyPar*. First, we observe that flow on its own are not strong enough to outperform our baseline configuration. Using the most balanced minimum cut heuristic gives us a configuration which significantly outperform the baseline configuration for large α'. The improvement is more significant for large α, because the larger the flow problem the larger is the number of minimum cuts. Enabling the FM heuristic improves the quality of our strongest configuration by 0.5% and also speed-ups the whole framework, because more work is transferred to the FM heuristic and therefore, the flow computations converge faster. Comparing the FM configuration with CONSTANT128, we observe that the quality is only improved between 0.3% and 0.5% with the drawback of an impractical running time. As our final configuration we choose the one with all heuristics enabled and $\alpha' = 16$.

Slide 26 We implemented several speed-up heuristics which might prevent unpromosing flow executions during our flow-based refinement. You don't have to remember exactly what the heuristics do. The take away message from this experiment is that enabling all speed-up heuristics produce comparable running time and speed-up the flow computations by a factor of 2. Finally, the slow-down compared to KaHyPar-CA is 1.72.

Slide 27 Finally, we compare the quality of our final configuration against other hypergraph partitioner on our full benchmark set. To do so, we use performance plots which represents the quality ratios of an partitioner for an instance compared to best quality achieved for that instance by a partitioner. On the x-axis, we have the instances and on the y-axis we have the quality ratio which is 1 minus the quality of the best partition produced by an partitioner divided by the quality produced by an specific partitioner. Before, we draw the points into the grid we sort the quality ratio of each partitioner in decreasing order. A value of zero means that the partitioner produces the best quality for an specific instance. A value greater than one means that the partitioner produces an infeasible solution according to our balanced constraints. The faster a partitioner intersects the zero line the better is the quality achieved by that partitioner in comparison.
Our new configuration KaHyPar-MF significantly outperforms all other hypergraph partitioner. It produces on 73% of the benchmarks the best partitions and improves the quality of KaHyPar-CA by 2.5%.

Slide 28 Comparing the running time of the different partitioner, we observe that the overall running time of KaHyPar-MF is only slower by a factor of 1.8 than KaHyPar-CA and the running time is comparable to the running time of the direct k-way version of *hMetis*. Moreover, the performance slow-down is less significantly on small flow network instances like DAC or DUAL, than on instances which induce large flow networks like PRIMAL or LITERAL.

Slide 29 To sum up the presentation I want to outline our main contributions and experimental results. Our framework is based on the ideas of Sanders and Schulz generalized to the hypergraph partitioning context. We develop techniques to sparsify the hypergraph flow network which results in a speedup of maximum flow algorithms up to a factor of 2 on some instances. We propose an optimized flow problem modeling approach which significantly improves the solution quality. Finally, we integrate speedup heuristics in our framework which ensure comparable quality and a speedup of the framework by a factor of 2.
Our main result is that we outperform 5 different state-of-the-art hypergraph partitioner on 73% of the benchmark instances. We improve the quality of *KaHyPar* by 2.5%, while only incurring a slowdown by a factor of 1.8. Additionally, our new version of *KaHyPar* has a comparable running time to *hMetis* and outperforms it on 84% of the benchmark instances.

Appendix **TODO 1:** *Add all experiments in Appendix*