Parallel Flow-Based Hypergraph Partitioning

July 25, 2022
Lars Gottesbüren, Tobias Heuer, Peter Sanders
Hypergraphs

- generalization of graphs
  ⇒ hyperedges connect ≥ 2 nodes

- graphs ⇒ dyadic (2-ary) relationships

- hypergraphs ⇒ (d-ary) relationships

- hypergraph $H = (V, E, c, \omega)$
  - vertex set $V = \{1, ..., n\}$
  - edge set $E \subseteq \mathcal{P}(V) \setminus \emptyset$
  - node weights $c : V \to \mathbb{R}_{\geq 1}$
  - edge weights $\omega : E \to \mathbb{R}_{\geq 1}$

- graphs ⇒ dyadic (2-ary) relationships
- hypergraphs ⇒ (d-ary) relationships
- hyperedges connect ≥ 2 nodes

- generalization of graphs
\( \varepsilon \)-Balanced Hypergraph Partitioning Problem

Partition hypergraph \( H = (V, E, c, \omega) \) into \( k \) disjoint blocks \( \Pi = \{ V_1, \ldots, V_k \} \) such that:

- blocks \( V_i \) are roughly equal-sized:

\[
\forall i \, c(V_i) \leq (1 + \varepsilon) \left\lceil \frac{c(V)}{k} \right\rceil
\]
ε-Balanced Hypergraph Partitioning Problem

Partition hypergraph $H = (V, E, c, \omega)$ into $k$ disjoint blocks $\Pi = \{ V_1, \ldots, V_k \}$ such that:

- blocks $V_i$ are roughly equal-sized:
  \[ c(V_i) \leq (1 + \varepsilon) \left\lceil \frac{c(V)}{k} \right\rceil \]

Connectivity objective is minimized.
**ε-Balanced Hypergraph Partitioning Problem**

Partition hypergraph $H = (V, E, c, \omega)$ into $k$ disjoint blocks $\Pi = \{ V_1, \ldots, V_k \}$ such that:

- blocks $V_i$ are **roughly equal-sized**:
  
  $$c(V_i) \leq (1 + \epsilon) \left\lceil \frac{c(V)}{k} \right\rceil$$

- **connectivity** objective is **minimized**:
ε-Balanced Hypergraph Partitioning Problem

Partition hypergraph $H = (V, E, c, \omega)$ into $k$ disjoint blocks $\Pi = \{V_1, \ldots, V_k\}$ such that:

- blocks $V_i$ are **roughly equal-sized:**
  \[ c(V_i) \leq (1 + \varepsilon) \left\lceil \frac{c(V)}{k} \right\rceil \]

- **connectivity** objective is **minimized**:
  \[ \sum_{e \in E} (\lambda(e) - 1) \omega(e) = 12 \]
Applications

Distributed Databases

Route Planning

VLSI Design

HPC
Trade-Off Landscape for Hypergraph Partitioning

- KaHyPar-HFC
- KaHyPar-CA
- hMetis-R
- PaToH-Q
- PaToH-D
- Zoltan
- BiPart
- Social Hash

Sequential
Shared Memory
Distributed

Quality

Speed
Trade-Off Landscape for Hypergraph Partitioning

- KaHyPar-HFC
- KaHyPar-CA
- hMetis-R
- PaToH-Q
- PaToH-D
- Mt-KaHyPar-D [ALENEX'21] [with 10 threads]
- Zoltan
- BiPart
- Social Hash

Sequential, Shared Memory, Distributed

- Speed
- Quality

low

high

slow

fast
Trade-Off Landscape for Hypergraph Partitioning

- KaHyPar-HFC
- KaHyPar-CA
- hMetis-R
- PaToH-Q
- PaToH-D
- Zoltan
- BiPart
- Social Hash
- Mt-KaHyPar-D [ALENEX'21] [with 10 threads]
- Mt-KaHyPar-Q [ALENEX'22] [with 10 threads]

Sequential
Shared Memory
Distributed
Trade-Off Landscape for Hypergraph Partitioning

- **Sequential**
  - BiPart
  - Social Hash
- **Shared Memory**
  - PaToH-Q
  - PaToH-D
  - Zoltan
- **Distributed**
  - Mt-KaHyPar-D [ALENEX'21]
  - Mt-KaHyPar-Q [ALENEX'22]
  - Mt-KaHyPar-Q-F [SEA'22]

- KaHyPar-HFC
- KaHyPar-CA
- hMetis-R

The diagram illustrates the trade-off between speed and quality for various hypergraph partitioning tools, with a focus on sequential, shared memory, and distributed methods.
Multilevel Partitioning

Input Hypergraph

Coarsening

cluster

contract

...
Multilevel Partitioning

- Input Hypergraph
- Coarsening
  - Cluster
  - Contract
- Initial Partitioning
Multilevel Partitioning

Coarsening

Input Hypergraph

cluster

contract

Initial Partitioning

Uncoarsening

local search

uncontract

5
Mt-KaHyPar: Algorithmic Components

Coarsening

Input Hypergraph

contract

cluster

Initial Partitioning

Uncoarsening

local search

uncontract

local

search
Mt-KaHyPar: Algorithmic Components

Input Hypergraph

Parallel Coarsening
Traditional log(n)-level Coarsening [ALENEX’21]

n-level Coarsening [ALENEX’22]

Thread 1

Thread 2

Initial Partitioning

Uncoarsening

local search

uncontract

Thread 1

Thread 2

n-level Coarsening [ALENEX’22]
Mt-KaHyPar: Algorithmic Components

Parallel Coarsening
Traditional log(n)-level Coarsening [ALENEX’21]

n-level Coarsening [ALENEX’22]

Parallel Recursive Bipartitioning based Initial Partitioning with Work-Stealing [ALENEX’21]

$k = 4$

Uncoarsening
Mt-KaHyPar: Algorithmic Components

Parallel Coarsening
Traditional log(n)-level Coarsening [ALENEX’21]

n-level Coarsening [ALENEX’22]

Parallel Recursive Bipartitioning based Initial Partitioning with Work-Stealing [ALENEX’21]

Parallel Direct $k$-Way FM [ALENEX’21]
Moves vertices greedily

Parallel Flow-Based Refinement [SEA’22]

$V_1$, $V_2$, $V_3$

$k = 4$

$V_1$, $V_2$, $V_3$, $V_4$

Task Queue
Thread 1: 1, 2, 2, 2, 2
Thread 2: 4, 2, 2, 2, 2
Thread 3: 4, 2, 2, 2, 2
Thread 4: 4, 2, 2, 2, 2

Work-Stealing

Task Queue
Thread 1: 1, 2, 2, 2, 2
Thread 2: 4, 2, 2, 2, 2
Thread 3: 4, 2, 2, 2, 2
Thread 4: 4, 2, 2, 2, 2
Mt-KaHyPar: Algorithmic Components

Parallel Coarsening
- Traditional log(n)-level Coarsening [ALENEX’21]
- n-level Coarsening [ALENEX’22]

Parallel Direct k-Way FM [ALENEX’21]
- Moves vertices greedily

Parallel Flow-Based Refinement [SEA’22]
- Improves
- Moves vertices greedily

Parallel Recursive Bipartitioning based Initial Partitioning with Work-Stealing [ALENEX’21]

$k = 4$

Input Hypergraph

Parallel Coarsening

Parallel Flow-Based Refinement

Thread 1

Thread 2
Maximum Flows

Flow $f$ / Capacity $c$
Maximum Flows

Flow Network
- Directed graph $\mathcal{N} = (\mathcal{V}, \mathcal{E}, c)$ with dedicated source $s \in \mathcal{V}$ and sink $t \in \mathcal{V}$
- Capacity Function: $c : \mathcal{V} \times \mathcal{V} \rightarrow \mathbb{R}$

Flow $f$ / Capacity $c$
Maximum Flows

Flow Network
- Directed graph $\mathcal{N} = (\mathcal{V}, \mathcal{E}, c)$ with dedicated source $s \in \mathcal{V}$ and sink $t \in \mathcal{V}$
- Capacity Function: $c : \mathcal{V} \times \mathcal{V} \to \mathbb{R}$

Flows
An valid $(s, t)$-flow $f : \mathcal{V} \times \mathcal{V} \to \mathbb{R}$ satisfies:
- $\forall u, v \in \mathcal{V} : f(u, v) \leq c(u, v)$ (capacity constraint)
- $\forall u, v \in \mathcal{V} : f(u, v) = -f(v, u)$ (skew symmetry constraint)
- $\forall u \in \mathcal{V} \setminus \{s, t\} : \sum_{v \in \mathcal{V}} f(u, v) = 0$ (flow conservation constraint)
Maximum Flows

Flow Network

- Directed graph $\mathcal{N} = (\mathcal{V}, \mathcal{E}, c)$ with dedicated source $s \in \mathcal{V}$ and sink $t \in \mathcal{V}$
- Capacity Function: $c : \mathcal{V} \times \mathcal{V} \rightarrow \mathbb{R}$

Flows
An valid $(s, t)$-flow $f : \mathcal{V} \times \mathcal{V} \rightarrow \mathbb{R}$ satisfies:

- $\forall u, v \in \mathcal{V} : f(u, v) \leq c(u, v)$ (capacity constraint)
- $\forall u, v \in \mathcal{V} : f(u, v) = -f(v, u)$ (skew symmetry constraint)
- $\forall u \in \mathcal{V} \setminus \{s, t\} : \sum_{v \in \mathcal{V}} f(u, v) = 0$ (flow conservation constraint)

Maximum $(s, t)$-Flow

- $|f| = \sum_{v \in \mathcal{V}} f(s, v) = \sum_{v \in \mathcal{V}} f(v, t)$ (flow value)
- For all other $(s, t)$-flows $f' : |f'| \leq |f|$
Maximum Flows

Flow Network
- Directed graph $\mathcal{N} = (\mathcal{V}, \mathcal{E}, c)$ with dedicated source $s \in \mathcal{V}$ and sink $t \in \mathcal{V}$
- Capacity Function: $c : \mathcal{V} \times \mathcal{V} \to \mathbb{R}$

Flows
- An valid $(s, t)$-flow $f : \mathcal{V} \times \mathcal{V} \to \mathbb{R}$
  - $\forall u, v \in \mathcal{V} : f(u, v) \leq c(u, v)$ (capacity constraint)
  - $\forall u, v \in \mathcal{V} : f(u, v) = -f(v, u)$ (skew symmetry constraint)
  - $\forall u \in \mathcal{V} \setminus \{s, t\} : \sum_{v \in \mathcal{V}} f(u, v) = 0$ (flow conservation constraint)

Max-Flow Min-Cut Theorem
- Maximum $(s, t)$-flow = Minimum $(s, t)$-Cut

Maximum $(s, t)$-Flow
- $|f| = \sum_{v \in \mathcal{V}} f(s, v) = \sum_{v \in \mathcal{V}} f(v, t)$ (flow value)
- For all other $(s, t)$-flows $f'$: $|f'| \leq |f|$
Push-Relabel Algorithm

Flow $f$ / Capacity $c$
Push-Relabel Algorithm

A node \( u \) is active if \( \text{exc}(u) > 0 \).

Distance label \( d(u) \) is a lower bound for the distance of \( u \) to \( t \).

Flow \( f \) / Capacity \( c \)
Push-Relabel Algorithm

push\((u, v)\)
- Sends $\delta = \min(exc(u), c(u, v) - f(u, v))$ over $(u, v)$
- Applicable if $u$ is active and $d(u) = d(v) + 1$

Flow $f$ / Capacity $c$
Push-Relabel Algorithm

push\((u, v)\)
- Sends $\delta = \min(\text{exc}(u), c(u, v) - f(u, v))$ over $(u, v)$
- Applicable if $u$ is active and $d(u) = d(v) + 1$

relabel\((u)\)
- Set $d(u) = \min\{d(v) + 1 \mid c(u, v) - f(u, v) > 0\}$
- Applicable if $u$ is active and there exists no edge over which we can push the remaining excess of $u$

Flow $f$ / Capacity $c$
Push-Relabel Algorithm

- **push**\((u, v)\) sends \(\delta = \min(\text{exc}(u), c(u, v) - f(u, v))\) over \((u, v)\).
  - Applicable if \(u\) is active and \(d(u) = d(v) + 1\).

- **relabel**\((u)\) sets \(d(u) = \min\{d(v) + 1 \mid c(u, v) - f(u, v) > 0\}\).
  - Applicable if \(u\) is active and there exists no edge over which we can push the remaining excess of \(u\).

- **discharge**\((u)\) performs all applicable push operations for node \(u\).
  - Relabel node \(u\).

Flow \(f\) / Capacity \(c\)

![Diagram showing the Push-Relabel Algorithm](image)
Push-Relabel Algorithm

- **push**($u, v$)
  - Sends $\delta = \min(\text{exc}(u), c(u, v) - f(u, v))$ over $(u, v)$
  - Applicable if $u$ is active and $d(u) = d(v) + 1$

- **relabel**($u$)
  - Set $d(u) = \min\{d(v) + 1 | c(u, v) - f(u, v) > 0\}$
  - Applicable if $u$ is active and there exists no edge over which we can push the remaining excess of $u$

- **discharge**($u$)
  - Perform all applicable push operations for node $u$
  - Relabel node $u$

---

Flow $f$ / Capacity $c$
Push-Relabel Algorithm

push\((u, v)\)
- Sends $\delta = \min(\text{exc}(u), c(u, v) - f(u, v))$ over \((u, v)\)
- Applicable if \(u\) is active and \(d(u) = d(v) + 1\)

relabel\((u)\)
- Set \(d(u) = \min\{d(v) + 1 \mid c(u, v) - f(u, v) > 0\}\)
- Applicable if \(u\) is active and there exists no edge over which we can push the remaining excess of \(u\)

discharge\((u)\)
- Perform all applicable push operations for node \(u\)
- Relabel node \(u\)

Flow \(f / \text{Capacity} \ c\)
Push-Relabel Algorithm

- **push**$(u, v)$
  - Sends $\delta = \min(\text{exc}(u), c(u, v) - f(u, v))$ over $(u, v)$
  - Applicable if $u$ is active and $d(u) = d(v) + 1$

- **relabel**$(u)$
  - Set $d(u) = \min\{d(v) + 1 | c(u, v) - f(u, v) > 0\}$
  - Applicable if $u$ is active and there exists no edge over which we can push the remaining excess of $u$

- **discharge**$(u)$
  - Perform all applicable push operations for node $u$
  - Relabel node $u$

**Flow $f$ / Capacity $c$**
Push-Relabel Algorithm

push($u, v$)
- Sends $\delta = \min(\text{exc}(u), c(u, v) - f(u, v))$ over $(u, v)$
- Applicable if $u$ is active and $d(u) = d(v) + 1$

relabel($u$)
- Set $d(u) = \min\{d(v) + 1 \mid c(u, v) - f(u, v) > 0\}$
- Applicable if $u$ is active and there exists no edge over which we can push the remaining excess of $u$

discharge($u$)
- Perform all applicable push operations for node $u$
- Relabel node $u$
Push-Relabel Algorithm

push(u, v)
- Sends $\delta = \min(\text{exc}(u), c(u, v) - f(u, v))$ over $(u, v)$
- Applicable if $u$ is active and $d(u) = d(v) + 1$

relabel(u)
- Set $d(u) = \min\{d(v) + 1 \mid c(u, v) - f(u, v) > 0\}$
- Applicable if $u$ is active and there exists no edge over which we can push the remaining excess of $u$

discharge(u)
- Perform all applicable push operations for node $u$
- Relabel node $u$
Push-Relabel Algorithm

push($u, v$)
- Sends $\delta = \min(\text{exc}(u), c(u, v) - f(u, v))$ over $(u, v)$
- Applicable if $u$ is active and $d(u) = d(v) + 1$

relabel($u$)
- Set $d(u) = \min\{d(v) + 1 | c(u, v) - f(u, v) > 0\}$
- Applicable if $u$ is active and there exists no edge over which we can push the remaining excess of $u$

discharge($u$)
- Perform all applicable push operations for node $u$
- Relabel node $u$

Flow $f$ / Capacity $c$
Push-Relabel Algorithm

flow $f$ / capacity $c$

push($u$, $v$)
- Sends $\delta = \min(\text{exc}(u), c(u, v) - f(u, v))$ over $(u, v)$
- Applicable if $u$ is active and $d(u) = d(v) + 1$

relabel($u$)
- Set $d(u) = \min\{d(v) + 1 \mid c(u, v) - f(u, v) > 0\}$
- Applicable if $u$ is active and there exists no edge over which we can push the remaining excess of $u$

discharge($u$)
- Perform all applicable push operations for node $u$
- Relabel node $u$
Push-Relabel Algorithm

\[\text{Flow } f \text{ / Capacity } c\]

\section*{Push-Relabel Algorithm}

\begin{itemize}
  \item **push**\((u, v)\)
    \begin{itemize}
      \item Sends \(\delta = \min(\text{exc}(u), c(u, v) - f(u, v))\) over \((u, v)\)
      \item Applicable if \(u\) is active and \(d(u) = d(v) + 1\)
    \end{itemize}
  
  \item **relabel**\((u)\)
    \begin{itemize}
      \item Set \(d(u) = \min\{d(v) + 1 \mid c(u, v) - f(u, v) > 0\}\)
      \item Applicable if \(u\) is active and there exists no edge over which we can push the remaining excess of \(u\)
    \end{itemize}
  
  \item **discharge**\((u)\)
    \begin{itemize}
      \item Perform all applicable push operations for node \(u\)
      \item Relabel node \(u\)
    \end{itemize}
\end{itemize}
Push-Relabel Algorithm

push($u, v$)
- Sends $\delta = \min(\text{exc}(u), c(u, v) - f(u, v))$ over $(u, v)$
- Applicable if $u$ is active and $d(u) = d(v) + 1$

relabel($u$)
- Set $d(u) = \min\{d(v) + 1 \mid c(u, v) - f(u, v) > 0\}$
- Applicable if $u$ is active and there exists no edge over which we can push the remaining excess of $u$

discharge($u$)
- Perform all applicable push operations for node $u$
- Relabel node $u$

Flow $f$ / Capacity $c$
Push-Relabel Algorithm

Push($u, v$)
- Sends $\delta = \min(\text{exc}(u), c(u, v) - f(u, v))$ over $(u, v)$
- Applicable if $u$ is active and $d(u) = d(v) + 1$

Relabel($u$)
- Set $d(u) = \min\{d(v) + 1 \mid c(u, v) - f(u, v) > 0\}$
- Applicable if $u$ is active and there exists no edge over which we can push the remaining excess of $u$

Discharge($u$)
- Perform all applicable push operations for node $u$
- Relabel node $u$

Flow $f$ / Capacity $c$
Flows on Hypergraph

Hypergraph $H$

Find Minimum $(s, t)$-Cut
Flows on Hypergraph

Hypergraph $H$

Find Minimum $(s, t)$-Cut

Lawler Expansion
Flow-Based Refinement – FlowCutter Algorithm

Bipartition $\Pi = \{V_1, V_2\}$

Cut Hyperedges

Hypergraph
Flow-Based Refinement – FlowCutter Algorithm

\[ c(B_1) \leq U_1 \]
Flow-Based Refinement – FlowCutter Algorithm
Flow-Based Refinement – FlowCutter Algorithm
Flow-Based Refinement – FlowCutter Algorithm
Flow-Based Refinement – FlowCutter Algorithm

Source-Side Cut \( \{ S_r, B \setminus S_r \} \)

Sink-Side Cut \( \{ T_r, B \setminus T_r \} \)

\[ B = B_1 \cup B_2 \]

Compute Minimum \((s, t)\)-Cut

\[ V_1 \]

\[ V_2 \]
Flow-Based Refinement – FlowCutter Algorithm

Assume \( \{ S_r, B \setminus S_r \} \) and \( \{ T_r, B \setminus T_r \} \) both induce an **imbalance**d bipartition on the original hypergraph.
Flow-Based Refinement – FlowCutter Algorithm

Contract smaller side onto corresponding terminal (assuming $c(S_r) \leq c(T_r)$)
Flow-Based Refinement – FlowCutter Algorithm

Additionally, we add one piercing node to the source
⇒ ensures that we find a different cut with better balance in the next iteration (potentially larger cut)

Contract smaller side onto corresponding terminal (assuming $c(S_r) \leq c(T_r)$)
Flow-Based Refinement – FlowCutter Algorithm
Flow-Based Refinement – FlowCutter Algorithm

Compute maximum \((s, t)\)-flow (initialized with previous flow assignment)
Flow-Based Refinement – FlowCutter Algorithm

New bipartition is **balanced** and **improved** cut from 7 to 5

Compute maximum $(s, t)$-flow (initialized with previous flow assignment)
Flow-Based Refinement – FlowCutter Algorithm
Parallelization
Parallelization

- Plugging in an existing parallel push-relabel algorithm [Baumstark et al. 2015]
- Discharge all active nodes in parallel
- Update flow globally, relabel nodes locally, excess deltas are aggregated using atomic instructions
- Fix an undocumented bug in the original algorithm
Parallelization

- Plugging in an existing parallel push-relabel algorithm [Baumstark et al. 2015]
  - Discharge all active nodes in parallel
  - Update flow globally, relabel nodes locally, excess deltas are aggregated using atomic instructions
  - Fix an undocumented bug in the original algorithm

Implementation Details

- Bulk Piercing
Parallelization

- Plugging in an existing parallel push-relabel algorithm [Baumstark et al. 2015]
- Discharge all active nodes in parallel
- Update flow globally, relabel nodes locally, excess deltas are aggregated using atomic instructions
- Fix an undocumented bug in the original algorithm

Implementation Details

- Bulk Piercing
- Restricting Capacities
Parallelization

- Plugging in an existing parallel push-relabel algorithm [Baumstark et al. 2015]
- Discharge all active nodes in parallel
- Update flow globally, relabel nodes locally, excess deltas are aggregated using atomic instructions
- Fix an undocumented bug in the original algorithm

Implementation Details

- Bulk Piercing
- Restricting Capacities
Parallelization

- Plugging in an existing parallel push-relabel algorithm [Baumstark et al. 2015]
- Discharge all active nodes in parallel
- Update flow globally, relabel nodes locally, excess deltas are aggregated using atomic instructions
- Fix an undocumented bug in the original algorithm

Implementation Details

- Bulk Piercing
- Restricting Capacities
- Implement push-relabel algorithm directly on hypergraph representation
**Parallelization**

- Plugging in an existing parallel push-relabel algorithm [Baumstark et al. 2015]
  - Discharge all active nodes in parallel
  - Update flow globally, relabel nodes locally, excess deltas are aggregated using atomic instructions
  - Fix an undocumented bug in the original algorithm

**Implementation Details**

- Bulk Piercing
- Restricting Capacities
- Implement push-relabel algorithm directly on hypergraph representation
- *many other optimizations*
Parallelization

- Plugging in an existing parallel push-relabel algorithm [Baumstark et al. 2015]
- Discharge all active nodes in parallel
- Update flow globally, relabel nodes locally, excess deltas are aggregated using atomic instructions
- Fix an undocumented bug in the original algorithm

Implementation Details

- Bulk Piercing
- Restricting Capacities
- Implement push-relabel algorithm directly on hypergraph representation
- many other optimizations

explained in our paper in more detail
Improving $k$-Way Partitions
Improving $k$-Way Partitions

**Idea:** Schedule flow computations on adjacent block pairs in parallel.
Improving $k$-Way Partitions

**Idea:** Schedule flow computations on adjacent block pairs in parallel

Schedule **Overlapping Flow Computations**
Improving $k$-Way Partitions

**Idea:** Schedule flow computations on adjacent block pairs in parallel

Schedule **Overlapping Flow Computations**
- We process $\min(t, k)$ block pairs in parallel
- Remaining threads are used for parallel flow computations
Improving $k$-Way Partitions

**Idea:** Schedule flow computations on adjacent block pairs in parallel

Schedule Overlapping Flow Computations
- We process $\min(t, k)$ block pairs in parallel
- Remaining threads are used for parallel flow computations
Improving $k$-Way Partitions

**Idea:** Schedule flow computations on adjacent block pairs in parallel

Schedule **Overlapping Flow Computations**
- We process $\min(t, k)$ block pairs in parallel
- Remaining threads are used for parallel flow computations

**Conflict Resolution Scheme**
- detects balance violations and flow computation that worsen the connectivity metric
- see paper
Experiments – Large Instances

- for comparison with fast sequential and parallel partitioners
- for scalability experiments

- 1st gen Epyc Rome, 1 socket, 64 cores @ 2.0-3.35 Ghz, 1024 GB RAM

- 94 large hypergraphs: [publicly available]
  - SuiteSparse Matrix Collection
  - SAT Competition 2014 (3 representations)
  - DAC2012 VLSI Circuits
- Largest hypergraph \( \approx 2 \text{ billion pins} \)

- \( k \in \{2, 8, 16, 64\} \) with imbalance: \( \varepsilon = 3\% \)
- 5 random seeds
- 1,4,16,64 threads
Scalability

![Graphs showing scalability](image)
Scalability

- Geometric Mean Speedup:
  - 3.1 with 4 Threads
  - 7.4 with 16 Threads
  - 10.6 with 64 Threads
- Instances with single-threaded time $\geq 100$s
  - 14.5 with 64 Threads
Scalability

- For $k = 2$, all parallelism is leveraged by the maximum flow algorithm
- Mediocre speedups for instances $< 100$s
- Instances with single-threaded time $\geq 100$s:
  - 13.3 with 64 Threads
- On par with speedups of Ref. [Baumstark et al., 2015]
Scalability

- For $k = 64$, all parallelism is leveraged by the scheduler
- Geometric Mean Speedups
  - 3.4 with 4 Threads
  - 10.7 with 16 Threads
  - 18.5 with 64 Threads
Experiments – Medium-Sized Instances

- for comparison with sequential partitioners: KaHyPar, hMetis, PaToH
- Intel Xeon Gold, 2 sockets, 20 cores @ 2.1 Ghz, 96 GB RAM

- 488 hypergraphs: [publicly available]
  - SuiteSparse Matrix Collection 184
  - SAT Competition 2014 (3 representations) 92·3 = 276
  - DAC2012 VLSI Circuits 10
  - ISPD98 18

- \( k \in \{2, 4, 8, 16, 32, 64, 128\} \) with imbalance: \( \varepsilon = 3\% \)
- 10 random seeds
- 10 threads
Experiments - Connectivity Metric (Quality)
Experiments - Connectivity Metric (Quality)

\[ p_{\text{Algo}}(\tau) = \frac{|\{I \in \mathcal{I} \mid \text{Algo}(I) \leq \tau \cdot \text{Best}(I)\}|}{|\mathcal{I}|} \]

Graph showing the fraction of instances as a function of quality relative to the best, with different algorithms represented by different lines.
Experiments - Connectivity Metric (Quality)

$$p_{\text{Algo}}(\tau) = \left| \left\{ I \in \mathcal{I} \mid \text{Algo}(I) \leq \tau \cdot \text{Best}(I) \right\} \right| / |\mathcal{I}|$$

For $\tau = 1 \Rightarrow$ fraction of instances for which an algorithm finds the best partition

Mt-KaHyPar-Q-F finds for 70% of the instances the best solution
Experiments - Connectivity Metric (Quality)

\[ p_{\text{Algo}}(\tau) = \frac{|\{ I \in \mathcal{I} | \text{Algo}(I) \leq \tau \cdot \text{Best}(I)\}|}{|\mathcal{I}|} \]

For \( \tau = 1 \Rightarrow \) fraction of instances for which an algorithm finds the best partition

- Mt-KaHyPar-Q-F finds for 70% of the instances the best solution

- The partitions produced by Mt-KaHyPar-Q-F are better than those of ...
  - Mt-KaHyPar-Q by 2.7% ...
  - hMetis by 3% ...
  - PaToH-Q by 6.4% ...
  - PaToH-D by 13% ...

... in the median
Experiments - Connectivity Metric (Quality)

\[ p_{\text{Algo}}(\tau) = \frac{|\{ I \in \mathcal{I} \mid \text{Algo}(I) \leq \tau \cdot \text{Best}(I)\}|}{|\mathcal{I}|} \]
Experiments - Connectivity Metric (Quality)

\[ p_{\text{Algo}}(\tau) = \frac{|\{I \in I \mid \text{Algo}(I) \leq \tau \cdot \text{Best}(I)\}|}{|I|} \]

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Gmean t [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>PaToH-D</td>
<td>1.17</td>
</tr>
<tr>
<td>Mt-KaHyPar-Q 10</td>
<td>2.98</td>
</tr>
<tr>
<td><strong>Mt-KaHyPar-Q-F 10</strong></td>
<td><strong>5.08</strong></td>
</tr>
<tr>
<td>PaToH-Q</td>
<td>5.86</td>
</tr>
<tr>
<td>kKaHyPar</td>
<td>48.97</td>
</tr>
<tr>
<td>hMetis-R</td>
<td>93.21</td>
</tr>
</tbody>
</table>

Quality Relative to Best [\(\tau\)]

Fraction of Instances
Conclusion

Mt-KaHyPar

- achieves the **same solution quality** as the highest quality sequential system in fast parallel code
- **order of magnitude faster** than its sequential counterparts with only 10 threads

https://github.com/kahypar/mt-kahypar
Conclusion

Mt-KaHyPar

- achieves the **same solution quality** as the highest quality sequential system in fast parallel code
- **order of magnitude faster** than its sequential counterparts with only 10 threads

Future Work

- How much quality is enough?
- Distributed-Memory Partitioning
- Large $k$ Partitioning
References

[Baumstark et al. 2015]
Trade-Off Landscape for Graph Partitioning
Trade-Off Landscape for Graph Partitioning