Parallel Flow-Based Hypergraph Partitioning

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

- generalization of graphs
  - hyperedges connect $\geq 2$ nodes

- graphs $\Rightarrow$ dyadic (2-ary) relationships

- hypergraphs $\Rightarrow$ (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 $\omega : E \to \mathbb{R}_{\geq 1}$
**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:

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

imbalance parameter
\(\varepsilon\)-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:
  \[c(V_i) \leq (1 + \varepsilon) \left\lceil \frac{c(V)}{k} \right\rceil\]

- connectivity objective is minimized:

---

2 July 25, 2022

Institute of Theoretical Informatics, Algorithmics II
\(\varepsilon\)-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 \textit{roughly equal-sized}:
  \[
  c(V_i) \leq (1 + \varepsilon) \left\lceil \frac{c(V)}{k} \right\rceil
  \]

- \textit{connectivity} objective is \textit{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

Speed
Quality

low
high

slow
fast
Trade-Off Landscape for Hypergraph Partitioning

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

Sequential
Shared Memory
Distributed

Quality
high
low

Speed
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]
- Mt-KaHyPar-Q [ALENEX'22] [with 10 threads]
Trade-Off Landscape for Hypergraph Partitioning

- KaHyPar-HFC
- hMetis-R
- PaToH-Q
- PaToH-D
- Zoltan
- BiPart
- Social Hash
- Mt-KaHyPar-D [ALENEX'21]
- Mt-KaHyPar-Q [ALENEX'22]
- Mt-KaHyPar-Q-F [SEA'22]

Sequential
Shared Memory
Distributed

Speed
- low
- fast

Quality
- high
- low
Multilevel Partitioning

Input Hypergraph

Coarsening

cluster

contract

...
Multilevel Partitioning

Input Hypergraph

Coarsening

cluster

contract

Initial Partitioning
Multilevel Partitioning

Coarsening

Input Hypergraph

cluster

contract

Uncoarsening

local search

uncontract

Initial Partitioning
Mt-KaHyPar: Algorithmic Components

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

- **Uncoarsening**
  - Local search
  - Uncontract

Diagram showing the process of coarsening and uncoarsening with initial partitioning.
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
Mt-KaHyPar: Algorithmic Components

Input Hypergraph

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]

$k = 4$

Parallel Recursion

Task Queue

Thread 1: f1, f2
Thread 2: f3, f5, f6
Thread 3: f4
Thread 4: f7

Task Queue

Thread 1: f1, f2
Thread 2: f3, f5, f6
Thread 3: f4
Thread 4: f7

Work-Stealing
Mt-KaHyPar: Algorithmic Components

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

**Parallel Flow-Based Refinement** [SEA'22]
- Moves vertices greedily

**Parallel Recursive Bipartitioning based Initial Partitioning with Work-Stealing** [ALENEX'21]
- k = 4
- Parallel Recursion

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

Improvement

Best Prefix

Moves

k = 4

Parallel Recursion
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: $c : \mathcal{V} \times \mathcal{V} \to \mathbb{R}$

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: $c : \mathcal{V} \times \mathcal{V} \rightarrow \mathbb{R}$

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 Flows

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

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

Maximum $(s, t)$-Flow
- $|f| = \sum_{v \in V} f(s, v) = \sum_{v \in V} f(v, t)$ (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: $c : \mathcal{V} \times \mathcal{V} \rightarrow \mathbb{R}$

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)
  - $\forall u \in \mathcal{V} \backslash \{s, t\} : \sum_{v \in \mathcal{V}} f(u, v) = 0$ (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 $\delta = \min(\text{exc}(u), c(u,v) - f(u,v))$ 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 $f / \text{Capacity } c$**
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 \(d(u) = \min\{d(v) + 1 \mid c(u, v) - f(u, v) > 0\}\)
- 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 $c$
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 $u$
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 $d(u) = \min\{d(v) + 1 | c(u, v) - f(u, v) > 0\}$
- 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 \(u\)

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**

**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

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 $u$
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 \(u\)
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 $u$
Flows on Hypergraph

![Diagram of Hypergraph H with vertices s, t, and edges e₁, e₂, e₃]

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

$V_1$

$V_2$
Flow-Based Refinement – FlowCutter Algorithm

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

\[ V_1 \quad B_1 \quad B_2 \quad V_2 \]
Flow-Based Refinement – FlowCutter Algorithm
Flow-Based Refinement – FlowCutter Algorithm

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

\[ B = B_1 \cup B_2 \]

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

Compute Minimum \((s, t)\)-Cut
Flow-Based Refinement – FlowCutter Algorithm

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

Contract smaller side onto corresponding terminal (assuming $c(S_t) \leq c(T_t)$)
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

Potential Conflicts when we apply min-cut
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) \( \cdot 3 = 42 \)
  - 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-url)
Scalability

- Geometric Mean Speedup:
  - 3.1 with 4 Threads
  - 7.4 with 16 Threads
  - 10.6 with 64 Threads

- Instances with single-threaded time $\geq 100s$
  - 14.5 with 64 Threads
Scalability

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

![Graph showing scalability results](image)
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)

![Graph showing the connectivity metric quality]

- Fraction of Instances
- Quality Relative to Best $[\tau]$

Key:
- Mt-KaHyPar-Q 10
- Mt-KaHyPar-Q-F 10
- hMetis-R
- PaToH-D
- PaToH-Q
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)

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

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

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

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

- 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)

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

| Algorithm            | Gmean | t [s] |
|----------------------|-------|-------|
| PaToH-D              |       | 1.17  |
| Mt-KaHyPar-Q 10      |       | 2.98  |
| **Mt-KaHyPar-Q-F 10**|       | 5.08  |
| PaToH-Q              |       | 5.86  |
| kKaHyPar             |       | 48.97 |
| hMetis-R             |       | 93.21 |
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

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

[Baumstark et al. 2015]
Niklas Baumstark, Guy E. Blelloch, and Julian Shun. Efficient Implementation of a Synchronous Parallel Push-Relabel Algorithm. In: *23rd European Symposium on Algorithms (ESA)*. Volume 9294. Springer, 2015, pages 106–117.
Institute of Theoretical Informatics, Algorithmics II

Trade-Off Landscape for Graph Partitioning

- KaFFPa-StrongS
- KaFFPa-Strong
- KaFFPa-EcoS
- ParHIP-Eco
- Mt-KaHIP
- KaMinPar
- Mt-Metis
- ParHIP-Fast
- ParMetis
- Scotch
- Metis-Fast
- Metis-K
- Metis-R
- KaFFPa-FastS

Sequential
Shared Memory
Distributed
Trade-Off Landscape for Graph Partitioning

- KaFFPa-StrongS
- KaFFPa-Strong
- KaFFPa-EcoS
- Mt-KaHyPar-Q-F [SEA'22]
- Mt-KaHyPar-Q [ALENEX'22]
- Mt-KaHyPar-D [ALENEX'21]
- ParHIP-Eco
- Mt-KaHIP
- KaMinPar
- ParMetis
- Scotch
- Metis-K
- KaFFPa-FastS
- Metis-R
- Mt-Metis
- ParHIP-Fast

Sequential
Shared Memory
Distributed

Speed
Quality