Scalable High-Quality Graph and Hypergraph Partitioning

June 13, 2022

Lars Gottesbüren, Tobias Heuer, Peter Sanders, Sebastian Schlag
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 \( \omega : E \to \mathbb{R}_{\geq 1} \)
\( \varepsilon \)-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:

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

- 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**:

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

- **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

Speed
Quality
low
high
low
fast
slow
Trade-Off Landscape for Hypergraph Partitioning

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

Sequential
Shared Memory
Distributed
Trade-Off Landscape for Hypergraph Partitioning

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

Sequential, Shared Memory, Distributed
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]
- Mt-KaHyPar-Q-F [SEA'22] [with 10 threads]

Sequential
Shared Memory
Distributed
Trade-Off Landscape for Graph Partitioning

- High Speed
- Low Speed
- High Quality
- Low Quality

Sequential
Shared Memory
Distributed
Trade-Off Landscape for Graph Partitioning
Multilevel Partitioning

Input Hypergraph

Coarsening

cluster

contract
Multilevel Partitioning

Input Hypergraph

Coarsening

cluster

contract

Initial Partitioning
Multilevel Partitioning

- **Input Hypergraph**
- **Coarsening**
  - cluster
  - contract
- **Initial Partitioning**
- **Uncoarsening**
  - local search
  - uncontract
Mt-KaHyPar: Algorithmic Components

Input Hypergraph

Coarsening

cluster

contract

Uncoarsening

local search

uncontract

Initial Partitioning
Mt-KaHyPar: Algorithmic Components

Parallel Coarsening

Traditional $\log(n)$-level Coarsening (Mt-KaHyPar-D)

$n$-level Coarsening (Mt-KaHyPar-Q)

Input Hypergraph

Uncoarsening

local search

uncontract

Thread 1
Thread 2

Initial Partitioning
Mt-KaHyPar: Algorithmic Components

Parallel Coarsening

Traditional log(n)-level Coarsening (Mt-KaHyPar-D)

n-level Coarsening (Mt-KaHyPar-Q)

Thread 1
Thread 2

Parallel Recursive Bipartitioning based Initial Partitioning with Work-Stealing

$k = 4$

Parallel Recursion

Task Queue

work-stealing

Uncoarsening

local search

uncontract
Mt-KaHyPar: Algorithmic Components

Parallel Coarsening
- Traditional log(n)-level Coarsening (Mt-KaHyPar-D)
- n-level Coarsening (Mt-KaHyPar-Q)

Parallel Direct $k$-Way FM
- Moves vertices greedily

Parallel Flow-Based Refinement

Parallel Recursive Bipartitioning based Initial Partitioning with Work-Stealing

$k = 4$

Input Hypergraph
Traditional Multilevel Partitioning

- contracts matching or clustering on each level
Traditional Multilevel Partitioning

- contracts matching or clustering on each level

![Diagram showing multilevel partitioning with nodes and edges]
Traditional Multilevel Partitioning

- contracts matching or clustering on each level
Traditional Multilevel Partitioning

- contracts matching or clustering on each level
Traditional Multilevel Partitioning

- contracts matching or clustering on each level
Traditional Multilevel Partitioning

- contracts matching or clustering on each level

⇒ approximately $O(\log n)$ levels
Traditional Multilevel Partitioning

- Contracts matching or clustering on each level

Tradeoff:

- More Levels
- Higher Quality
- Higher Running Time

⇒ Approximately $O(\log n)$ levels
$n$-level Partitioning

- contract one vertex at a time
\textit{n-level Partitioning}

- contract one vertex at a time
$n$-level Partitioning

- contract one vertex at a time
\( n \)-level Partitioning

- contract one vertex at a time
\textit{n-level Partitioning}

- contract one vertex at a time
$n$-level Partitioning

- contract one vertex at a time

Coarsening: Almost $n$ levels

Unoarsening: Almost $n$ local search invocations $\Rightarrow$ **High Quality**! (used in KaHyPar)
\( n \)-level Partitioning

- contract one vertex at a time

\[ v_0 \quad v_1 \quad v_2 \quad v_3 \quad v_4 \quad v_5 \quad v_6 \]

\[ e_0 \quad e_1 \quad e_2 \quad e_3 \]

\textbf{Coarsening}: Almost \( n \) levels

\textbf{Unoarsening}: Almost \( n \) local search invocations \( \Rightarrow \) \textbf{High Quality!} (used in KaHyPar)

\( \Rightarrow \) \textbf{Inherently Sequential!}
Contraction Forest

Any sequence of contractions form a forest
Contraction Forest

Any sequence of contractions form a forest
Contraction Forest

Any sequence of contractions form a forest

$v_4$ is contracted onto $v_2$
Contraction Forest

Any sequence of contractions form a forest

Contraction Forest

Roots are the vertices of the coarsest hypergraph
Contraction Forest

Any sequence of contractions form a forest

Contraction Forest

Contraction order:
1. Contract $v_{15}$ onto $v_8$
2. Contract $v_8$ onto $v_4$
3. Contract $v_4$ onto $v_2$
Contraction Forest

Any sequence of contractions form a forest

Contraction Forest

Observations

- There is more than one contraction order leading to the same contraction forest
Contraction Forest

Any sequence of contractions form a forest

Observations
- There is more than one contraction order leading to the same contraction forest

Rules
- Contractions in different subtrees are independent
- Contract $v$ when its children are contracted onto $v$
**Contraction Forest**

Any sequence of contractions form a forest

![Contraction Forest Diagram]

**Observations**
- There is more than one contraction order leading to the same contraction forest

**Rules**
- Contractions in different subtrees are independent
- Contract $v$ when its children are contracted onto $v$

**Parallelization Idea**
- Contract contraction forest bottom-up in parallel
Contraction Forest

Any sequence of contractions form a forest

Observations
- There is more than one contraction order leading to the same contraction forest

Rules
- Contractions in different subtrees are independent
- Contract $v$ when its children are contracted onto $v$

Parallelization Idea
- Contract contraction forest bottom-up in parallel
Contraction Forest

Any sequence of contractions form a forest

\[ T_i = \text{Thead } i \]

Observations

- There is more than one contraction order leading to the same contraction forest

Rules

- Contractions in different subtrees are independent
- Contract \( v \) when its children are contracted onto \( v \)

Parallelization Idea

- Contract contraction forest bottom-up in parallel
Contraction Forest

Any sequence of contractions form a forest

\[ T_i = \text{Thead } i \]

Contraction Forest

Observations

- There is more than one contraction order leading to the same contraction forest

Rules

- Contractions in different subtrees are independent
- Contract \( v \) when its children are contracted onto \( v \)

Parallelization Idea

- Contract contraction forest bottom-up in parallel
## Contraction Forest

Any sequence of contractions form a forest

<table>
<thead>
<tr>
<th>Contraction Forest</th>
<th>Observations</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_1 \quad v_2$</td>
<td>- There is more than one contraction order leading to the same contraction forest</td>
</tr>
<tr>
<td>$T_i = \text{Thead } i$</td>
<td>- Constructions in different subtrees are independent</td>
</tr>
<tr>
<td></td>
<td>- Contract $v$ when its children are contracted onto $v$</td>
</tr>
</tbody>
</table>

### Parallelization Idea
- Contract contraction forest bottom-up in parallel
Contraction Forest

Any sequence of contractions form a forest

\[ T_i = \text{Thead } i \]

**Contraction Forest**

- \( v_1 \)
- \( v_2 \)

**Observations**

- There is more than one contraction order leading to the same contraction forest

**Rules**

- Contractions in different subtrees are independent
- Contract \( v \) when its children are contracted onto \( v \)

**Parallelization Idea**

- Contract contraction forest bottom-up in parallel

**Problem:** Contraction forest is not known in advance
Contraction Forest Construction

**Idea**: Construct contraction forest *on-the-fly*

---

**Algorithm 1**: Parallel $n$-level Coarsening

```plaintext
for each $u \in V$ in parallel
    $v \leftarrow$ find contraction partner for $u$
    if add $(v, u)$ to contraction forest then
        contract $v$ onto $u$
```
Contraction Forest Construction

**Idea:** Construct contraction forest *on-the-fly*

<table>
<thead>
<tr>
<th>Algorithm 1: Parallel $n$-level Coarsening</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>for each</strong> $u \in V$ <strong>in parallel</strong></td>
</tr>
<tr>
<td>$v \leftarrow$ find contraction partner for $u$</td>
</tr>
<tr>
<td><strong>if</strong> <em>add</em> $(v, u)$ <em>to contraction forest</em> <strong>then</strong></td>
</tr>
<tr>
<td>contract $v$ onto $u$</td>
</tr>
</tbody>
</table>
Contraction Forest Construction

**Idea:** Construct contraction forest *on-the-fly*

\[ T_i = \text{Thead } i \]

**Algorithm 1:** Parallel \( n \)-level Coarsening

\[
\text{for each } u \in V \text{ in parallel} \\
\quad v \leftarrow \text{find contraction partner for } u \\
\quad \text{if } \text{add } (v, u) \text{ to contraction forest then} \\
\quad \quad \text{contract } v \text{ onto } u
\]
Contraction Forest Construction

**Idea:** Construct contraction forest *on-the-fly*

$T_i = \text{Thead } i$

Algorithm 1: Parallel $n$-level Coarsening

```plaintext
for each $u \in V$ in parallel
    $v \leftarrow \text{find contraction partner for } u$
    if add $(v, u)$ to contraction forest then
        contract $v$ onto $u$
```

pending contractions on node $v_2$
Contraction Forest Construction

**Idea**: Construct contraction forest *on-the-fly*

\[ T_i = \text{Thead } i \]

**Algorithm 1**: Parallel \( n \)-level Coarsening

```plaintext
for each \( u \in V \) in parallel
    \( v \leftarrow \text{find contraction partner for } u \)
    if add \((v, u)\) to contraction forest then
        contract \( v \) onto \( u \)
```

---

June 13, 2022  Tobias Heuer – Scalable High-Quality Graph and Hypergraph Partitioning  Institute of Theoretical Informatics, Algorithmics II
Contraction Forest Construction

**Idea**: Construct contraction forest on-the-fly

\[ T_i = \text{Thead}_i \]

**Algorithm 1**: Parallel \( n \)-level Coarsening

```plaintext
for each \( u \in V \) in parallel
    \( v \leftarrow \) find contraction partner for \( u \)
    \[ \text{if add} (v, u) \text{ to contraction forest then} \]
    contract \( v \) onto \( u \)
```
Contraction Forest Construction

**Idea:** Construct contraction forest *on-the-fly*

\[(v_2, v_3)\text{ is not eligible for contraction}\]
\[\Rightarrow \text{do something else}\]

\[T_i = \text{Thead } i\]

**Algorithm 1:** Parallel \(n\)-level Coarsening

```plaintext
for each \(u \in V\) in parallel
    \(v \leftarrow \text{find contraction partner for } u\)
    if add \((v, u)\) to contraction forest then
        contract \(v\) onto \(u\)
```

\(v\) is the root vertex.
\(u\) is a child vertex.
\((v, u)\) is the edge connecting them.

The contraction is performed by merging the two vertices into a single vertex.

The contraction forest is a collection of trees, where each tree represents a connected component of the graph.

In this context, the contraction process is performed on-the-fly, meaning that the contraction decisions are made as the algorithm progresses, rather than being determined in advance. This allows for more dynamic and efficient partitioning and coarsening of the graph.
Contraction Forest Construction

Idea: Construct contraction forest on-the-fly

\[ T_i = \text{Thead } i \]

Algorithm 1: Parallel $n$-level Coarsening

for each $u \in V$ in parallel

\[ v \leftarrow \text{find contraction partner for } u \]

if add $(v, u)$ to contraction forest then

contract $v$ onto $u$
Contraction Forest Construction

**Idea:** Construct contraction forest *on-the-fly*

\[ T_i = \text{Thead } i \]

**Algorithm 1:** Parallel \( n \)-level Coarsening

```plaintext
for each \( u \in V \) in parallel
    \( v \leftarrow \text{find contraction partner for } u \)
    if add \((v, u)\) to contraction forest then
        contract \( v \) onto \( u \)
```

\[ v_1 \]
\[ v_2 \]
\[ v_3 \]
\[ v_5 \]
\[ v_8 \]
Contraction Forest Construction

**Idea**: Construct contraction forest *on-the-fly*

Algorithm 1: Parallel $n$-level Coarsening

```plaintext
for each $u \in V$ in parallel
    $v \leftarrow$ find contraction partner for $u$
    if add $(v, u)$ to contraction forest then
        contract $v$ onto $u$
```

$T_i = \text{Thead } i$
Contraction Forest Construction

**Idea:** Construct contraction forest *on-the-fly*

\[ T_i = \text{Thead } i \]

Algorithm 1: Parallel \( n \)-level Coarsening

for each \( u \in V \) in parallel

\[ v \leftarrow \text{find contraction partner for } u \]

if add \((v, u)\) to contraction forest then

contract \( v \) onto \( u \)

Cyclic Dependency
\[ \Rightarrow \text{Discard Contraction} \]
Contraction Forest Construction

Idea: Construct contraction forest on-the-fly

\( T_i = \text{Thead } i \)

Algorithm 1: Parallel \( n \)-level Coarsening

\[
\text{for each } u \in V \text{ in parallel} \\
\quad v \leftarrow \text{find contraction partner for } u \\
\quad \text{if add } (v, u) \text{ to contraction forest then} \\
\quad \quad \text{contract } v \text{ onto } u
\]
Contraction Forest Construction

**Idea:** Construct contraction forest *on-the-fly*

\[ T_i = \text{Thead } i \]

---

**Algorithm 1:** Parallel \( n \)-level Coarsening

```
for each \( u \in V \) in parallel
    \( v \leftarrow \text{find contraction partner for } u \)
    if add \((v, u)\) to contraction forest then
        contract \( v \) onto \( u \)
```

---
Contraction Forest Construction

Idea: Construct contraction forest on-the-fly

$T_i = \text{Thead } i$

Algorithm 1: Parallel $n$-level Coarsening

for each $u \in V$ in parallel

$v \leftarrow \text{find contraction partner for } u$

if add $(v, u)$ to contraction forest then

contract $v$ onto $u$


Contraction Forest Construction

**Idea:** Construct contraction forest *on-the-fly*

Thread $T_3$ decreases pending counter of $v_2$ to zero  
⇒ Recursively continue

$T_i = \text{Thread } i$

Algorithm 1: Parallel $n$-level Coarsening

```
for each $u \in V$ in parallel
  v ← find contraction partner for $u$
  if add $(v, u)$ to contraction forest then
    contract $v$ onto $u$
```

Contraction Forest Construction

Idea: Construct contraction forest on-the-fly

Algorithm 1: Parallel $n$-level Coarsening

for each $u \in V$ in parallel
    $v \leftarrow$ find contraction partner for $u$
    if add ($v$, $u$) to contraction forest then
        contract $v$ onto $u$
Contraction Forest Construction

**Idea:** Construct contraction forest *on-the-fly*

\[ T_i = \text{Thead } i \]

---

**Algorithm 1:** Parallel \( n \)-level Coarsening

for each \( u \in V \) in parallel

\[ v \leftarrow \text{find contraction partner for } u \]

\[
\text{if } \text{add} (v, u) \text{ to contraction forest} \text{ then} \]

\[
\text{contract } v \text{ onto } u
\]
Contraction Forest Construction

**Idea:** Construct contraction forest *on-the-fly*

---

\[ T_i = \text{Thead } i \]

---

**Algorithm 1:** Parallel \( n \)-level Coarsening

```plaintext
for each \( u \in V \) in parallel
    \( v \leftarrow \text{find contraction partner for } u \)
    if add \((v, u)\) to contraction forest then
        contract v onto u
```

- Simple locking protocol used to modify contraction forest
Consistency Requirements

Contraction Consistency

Data Structure Consistency
Consistency Requirements

Contraction Consistency

Data Structure Consistency

see paper
Parallel Uncoarsening

- traditional $n$-level uncontracts only **one** vertex on each level $\Rightarrow$ inherently sequential
Parallel Uncoarsening

- traditional $n$-level uncontracts only one vertex on each level $\Rightarrow$ inherently sequential

Idea
- assemble independent uncontractions in a batch $B$ with $|B| \approx b_{\text{max}}$
- uncontract $B$ in parallel
- then run parallel localized refinement around $B$
- construct batches $\mathcal{B} = \langle B_1, \ldots, B_l \rangle$
- uncontracting $B_i$ enables uncontraction of all vertices in $B_{i+1}$
Parallel Uncoarsening

- Traditional $n$-level uncontracts only one vertex on each level $\Rightarrow$ inherently sequential

**Idea**
- Assemble independent uncontractions in a batch $B$ with $|B| \approx b_{max}$
  - Uncontract $B$ in parallel
  - Then run parallel localized refinement around $B$
- Construct batches $B = \{B_1, \ldots, B_l\}$
- Uncontracting $B_i$ enables uncontraction of all vertices in $B_{i+1}$
- **Top-down traversal** of contraction forest $\mathcal{F}$

\[
\begin{align*}
  b_{max} &= 3 \\
  B &= \langle \text{eligible for uncontraction}, \text{eligible for uncontraction}, \text{eligible for uncontraction}, \text{eligible for uncontraction}, \text{already uncontracted}, \text{already uncontracted} \rangle
\end{align*}
\]
Parallel Uncoarsening

- traditional $n$-level uncontracts only one vertex on each level $\Rightarrow$ inherently sequential

**Idea**
- assemble independent uncontractions in a batch $B$ with $|B| \approx b_{\text{max}}$
- uncontract $B$ in parallel
- then run parallel localized refinement around $B$
- construct batches $B = \langle B_1, \ldots, B_l \rangle$
- uncontracting $B_i$ enables uncontraction of all vertices in $B_{i+1}$
- **top-down traversal** of contraction forest $\mathcal{F}$

$b_{\text{max}} = 3$

$B = \langle v_3, v_7, v_4 \rangle$

- $v_3$ already uncontracted
- $v_4$ eligible for uncontraction
Parallel Uncoarsening

- traditional $n$-level uncontracts only one vertex on each level $\Rightarrow$ inherently sequential

**Idea**
- assemble independent uncontractions in a *batch* $B$ with $|B| \approx b_{\text{max}}$
  - uncontract $B$ in parallel
  - then run parallel localized refinement around $B$
- construct *batches* $\mathcal{B} = \langle B_1, \ldots, B_l \rangle$
- uncontracting $B_i$ enables uncontraction of all vertices in $B_{i+1}$
- **top-down traversal** of contraction forest $\mathcal{F}$

$$b_{\text{max}} = 3$$

$$\mathcal{B} = \langle v_3, v_7, v_4, v_5, v_6, v_{12}, \square, \square, \square \rangle$$
Parallel Uncoarsening

- traditional \( n \)-level uncontracts only one vertex on each level \( \Rightarrow \) inherently sequential

**Idea**
- assemble independent uncontractions in a batch \( B \) with \( |B| \approx b_{\text{max}} \)
  - uncontract \( B \) in parallel
  - then run parallel localized refinement around \( B \)
- construct batches \( \mathcal{B} = \langle B_1, \ldots, B_l \rangle \)
- uncontracting \( B_i \) enables uncontraction of all vertices in \( B_{i+1} \)
- top-down traversal of contraction forest \( \mathcal{F} \)

\[
\begin{align*}
b_{\text{max}} &= 3 \\
\mathcal{B} &= \langle \{v_3, v_7, v_4\}, \{v_5, v_6, v_{12}\}, \{v_8, v_9, v_{10}\}, \ldots \rangle
\end{align*}
\]
Parallel Uncoarsening

- traditional \( n \)-level uncontracts only one vertex on each level \( \Rightarrow \) inherently sequential

**Idea**
- assemble independent uncontractions in a batch \( B \) with \( |B| \approx b_{\text{max}} \)
  - uncontract \( B \) in parallel
  - then run parallel localized refinement around \( B \)
- construct batches \( \mathcal{B} = \langle B_1, \ldots, B_l \rangle \)
- uncontracting \( B_i \) enables uncontraction of all vertices in \( B_{i+1} \)
- **top-down traversal** of contraction forest \( \mathcal{F} \)

\[
\begin{align*}
  b_{\text{max}} &= 3 \\
  \mathcal{B} &= \langle \{v_3, v_7, v_4\}, \{v_5, v_6, v_{12}\}, \{v_8, v_9, v_{10}\}, \{v_{11}, v_{13}, v_{14}\}, \{v_9, v_{10}, v_{11}, v_{12}, v_{13}, v_{14}, \text{ already uncontracted} \} \rangle
\end{align*}
\]
Parallel Uncoarsening

- traditional \( n \)-level uncontracts only one vertex on each level \( \Rightarrow \) inherently sequential

**Idea**

- assemble independent uncontractions in a *batch* \( B \) with \( |B| \approx b_{\text{max}} \)
  - uncontract \( B \) in parallel
  - then run parallel localized refinement around \( B \)
- construct *batches* \( B = \langle B_1, \ldots, B_l \rangle \)
- uncontracting \( B_i \) enables uncontraction of all vertices in \( B_{i+1} \)
- **top-down traversal** of contraction forest \( \mathcal{F} \)

\[
\begin{align*}
b_{\text{max}} &= 3 \\
\mathcal{B} &= \langle \langle \langle v_3, v_7, v_4, v_5, v_6, v_{12}, v_8, v_9, v_{10}, v_{11}, v_{13}, v_{14}, v_{15} \rangle \rangle \rangle
\end{align*}
\]
Parallel Uncoarsening

- traditional $n$-level uncontracts only one vertex on each level $\Rightarrow$ inherently sequential

**Idea**
- assemble independent uncontractions in a batch $B$ with $|B| \approx b_{\text{max}}$
  - uncontract $B$ in parallel
  - then run parallel localized refinement around $B$
- construct batches $\mathcal{B} = \langle B_1, \ldots, B_l \rangle$
- uncontracting $B_i$ enables uncontraction of all vertices in $B_{i+1}$
- top-down traversal of contraction forest $\mathcal{F}$

- $b_{\text{max}} = 3$
- $\mathcal{B} = \langle \langle v_3, v_7, v_4, v_5, v_6, v_{12}, v_8, v_9, v_{10}, v_{11}, v_{13}, v_{14}, v_{15} \rangle \rangle$

### Diagram

- $b_{\text{max}} = 1000$ in practice
- eligible for uncontraction
- already uncontracted
Parallel Uncoarsening

- traditional $n$-level uncontracts only one vertex on each level ⇒ inherently sequential

Idea
- assemble independent uncontractions in a batch $B$ with $|B| \approx b_{\text{max}}$
- uncontract $B$ in parallel
- then run parallel localized refinement around $B$
- construct batches $B = \langle B_1, \ldots, B_l \rangle$
- uncontracting $B_i$ enables uncontraction of all vertices in $B_{i+1}$
- top-down traversal of contraction forest $F$
- $b_{\text{max}} = 3$
- $B = \langle \langle v_3, v_7, v_4 \rangle, \langle v_5, v_6, v_{12} \rangle, \langle v_8, v_9, v_{10} \rangle, \langle v_{11}, v_{13}, v_{14} \rangle, \langle v_{15} \rangle \rangle$

Implementation Detail:
Uncontract siblings in reverse order of contraction ⇒ see paper

Implementation Detail:
Uncontract siblings in reverse order of contraction ⇒ see paper

$\bullet$ eligible for uncontraction
\(\bullet\) already uncontracted

$b_{\text{max}} = 1000$ in practice
Parallel Flow-Based Refinement

The value of a **maximum flow** between two vertices $s$ and $t$ is equal with the **minimum cut** separating $s$ and $t$. 
Parallel Flow-Based Refinement

The value of a maximum flow between vertices \( s \) and \( t \) is equal to the minimum cut separating \( s \) and \( t \).
Parallel Flow-Based Refinement

The value of a **maximum flow** between two vertices $s$ and $t$ is equal with the **minimum cut** separating $s$ and $t$

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

Hypergraph

Cut Hyperedges
Parallel Flow-Based Refinement

The value of a \textbf{maximum flow} between to vertices $s$ and $t$ is equal with the \textbf{minimum cut} seperating $s$ and $t$.

Initial Cut = 539, Target Imbalance = 3%
Parallel Flow-Based Refinement

The value of a **maximum flow** between two vertices $s$ and $t$ is equal with the **minimum cut** separating $s$ and $t$

Initial Cut = 539, Target Imbalance = 3%

Grow region around cut via BFS
Parallel Flow-Based Refinement

The value of a **maximum flow** between vertices $s$ and $t$ is equal with the **minimum cut** separating $s$ and $t$.

Initial Cut = 539, Target Imbalance = 3%
Parallel Flow-Based Refinement

The value of a \textbf{maximum flow} between to vertices $s$ and $t$ is equal with the \textbf{minimum cut} separating $s$ and $t$

Initial Cut = 539, Target Imbalance = 3%

Compute a maximum $(s, t)$-flow
Parallel Flow-Based Refinement

The value of a maximum flow between two vertices $s$ and $t$ is equal with the minimum cut separating $s$ and $t$.

Initial Cut = 539, Target Imbalance = 3%

Current Cut = 250, Current Imbalance = 15% Imbalanced!
Parallel Flow-Based Refinement

The value of a maxium flow between to vertices $s$ and $t$ is equal with the minimum cut seperating $s$ and $t$

Initial Cut = 539, Target Imbalance = 3%

Contract smaller cut onto its terminal plus one additional node

Piercing node ensure that we find a different cut in the next iteration

Current Cut = 250, Current Imbalance = 15% Imbalanced!
Parallel Flow-Based Refinement

The value of a **maximum flow** between to vertices \( s \) and \( t \) is equal with the **minimum cut** seperating \( s \) and \( t \)

Initial Cut = 539, Target Imbalance = 3%

Contract smaller cut onto its terminal plus one additional node

\[ V_1 \quad S \quad V_2 \]
The value of a **maximum flow** between two vertices $s$ and $t$ is equal to the **minimum cut** separating $s$ and $t$.

Initial Cut = 539, Target Imbalance = 3%

Augment flow again to a maximum $(s, t)$-flow
Parallel Flow-Based Refinement

The value of a **maximum flow** between to vertices \( s \) and \( t \) is equal with the **minimum cut** separating \( s \) and \( t \)

Initial Cut = 539, Target Imbalance = 3%

Current Cut = 498, Current Imbalance = 2.5%

Balanced!

Improvement = 539 – 498 = 41
Parallel Flow-Based Refinement

The value of a \textbf{maximum flow} between two vertices \(s\) and \(t\) is equal with the \textbf{minimum cut} separating \(s\) and \(t\).

\[
\text{New Cut} = 498, \text{New Imbalance} = 2.5\%
\]
Parallel Flow-Based Refinement

The value of a \textit{maximum flow} between to vertices $s$ and $t$ is equal with the \textit{minimum cut} seperating $s$ and $t$

New Cut = 498, New Imbalance = 2.5%

Our implementation uses a \textit{parallel} maximum flow algorithm (push-relabel algorithm)
Parallel Flow-Based Refinement

**General Idea:** Schedule parallel flow problems on adjacent block pairs
Parallel Flow-Based Refinement

General Idea: Schedule parallel flow problems on adjacent block pairs

Nodes can overlap
Parallel Flow-Based Refinement

**General Idea:** Schedule parallel flow problems on adjacent block pairs

- Flow computation returns a sequences moves
- What could possibly go wrong?
Parallel Flow-Based Refinement

**General Idea:** Schedule parallel flow problems on adjacent block pairs

- Flow computation returns a sequence of moves
- What could possibly go wrong?
  - Applying the move sequence could violate the balance constraint
Parallel Flow-Based Refinement

**General Idea:** Schedule parallel flow problems on adjacent block pairs

- **Nodes can overlap**

- Flow computation returns a sequence of moves
- What could possibly go wrong?
  - Applying the move sequence could violate the balance constraint
  - Applying the move sequence could worsen the solution quality
Experiments – Large Instances

- for comparison with fast partitioners: Zoltan, PaToH-D, Hype, BiPart
- for scaling 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 42
  - SAT Competition 2014 (3 representations) 14·3 = 42
  - DAC2012 VLSI Circuits 10
- Largest hypergraph $\approx$ 2 billion pins

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

![Graph showing scalability experiments]

- **Total Computation**
- **Coarsening**
- **Initial Partitioning**
- **Batch Uncontractions**
- **Localized Label Propagation**
- **Localized FM**

Legend:
- Mt-KaHyPar-Q 4
- Mt-KaHyPar-Q 16
- Mt-KaHyPar-Q 64
Experiments – Scalability

- Harmonic mean speedup of Mt-KaHyPar-Q:
  - 3.7 with 4 threads
  - 11.7 with 16 threads
  - 22.6 with 64 threads

- Instances $\geq 100s$:
  - 3.7 with 4 threads
  - 12.3 with 16 threads
  - 25 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)

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

\[ \tau \]

Fraction of instances

\[ \begin{align*}
\text{KaHyPar-CA} & \quad \text{PaToH-Q} \\
\text{Mt-KaHyPar-Q 10} & \quad \text{PaToH-D} \\
\text{Mt-KaHyPar-D 10} & \quad \text{PaToH-D}
\end{align*} \]
Experiments – Connectivity Metric (Quality)

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

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

KaHyPar-CA ≈ 50%

Mt-KaHyPar-Q ≈ 37%

Mt-KaHyPar-D ≈ 5%
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)

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Gmean</th>
<th>t [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mt-KaHyPar-D 10</td>
<td>0.95</td>
<td></td>
</tr>
<tr>
<td>PaToH-D</td>
<td>1.17</td>
<td></td>
</tr>
<tr>
<td>Mt-KaHyPar-Q 10</td>
<td>3.19</td>
<td></td>
</tr>
<tr>
<td>PaToH-Q</td>
<td>5.86</td>
<td></td>
</tr>
<tr>
<td>KaHyPar-CA</td>
<td>28.14</td>
<td></td>
</tr>
</tbody>
</table>
\[ p_{\text{Algo}}(\tau) = \frac{|\{ I \in I | \text{Algo}(I) \leq \tau \cdot \text{Best}(I)\}|}{|I|} \]
\[ p_{\text{Algo}}(\tau) = \frac{\left| \{ I \in \mathcal{I} \mid \text{Algo}(I) \leq \tau \cdot \text{Best}(I) \} \right|}{|\mathcal{I}|} \]
\[ p_{\text{Algo}}(\tau) = \frac{|\{ I \in \mathcal{I} \mid \text{Algo}(I) \leq \tau \cdot \text{Best}(I)\}|}{|\mathcal{I}|} \]

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Gmean t[s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mt-KaHyPar-Q 10</td>
<td>3.19</td>
</tr>
<tr>
<td>Mt-KaHyPar-Q-F 10</td>
<td>5.08</td>
</tr>
<tr>
<td>KaHyPar-HFC</td>
<td>48.98</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>$G\text{mean}$ $t[s]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mt-KaHyPar-Q 10</td>
<td>3.19</td>
</tr>
<tr>
<td>Mt-KaHyPar-Q-F 10</td>
<td>5.08</td>
</tr>
<tr>
<td>KaHyPar-HFC</td>
<td>48.98</td>
</tr>
</tbody>
</table>

$p_{\text{Algo}}(\tau) = \left| \left\{ I \in \mathcal{I} \mid \text{Algo}(I) \leq \tau \cdot \text{Best}(I) \right\} \right| / |\mathcal{I}|$
Multilevel vs $n$-Level Partitioning
Multilevel vs $n$-Level Partitioning

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Gmean $t[s]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mt-KaHyPar-D 10</td>
<td>0.89</td>
</tr>
<tr>
<td>Mt-KaHyPar-Q 10</td>
<td>2.99</td>
</tr>
</tbody>
</table>
Multilevel vs $n$-Level Partitioning

Does Mt-KaHyPar-Q have an unfair advantage due to its longer running time?
Effectiveness Tests

- **Idea**: Perform additional runs with the faster algorithm until its expected running time equals the running time of the slower algorithm
Effectiveness Tests

- **Idea**: Perform additional runs with the faster algorithm until its expected running time equals the running time of the slower algorithm.
- Given an instance $I$ and two algorithms $A$ and $B$.
Effectiveness Tests

- **Idea**: Perform additional runs with the faster algorithm until its expected running time equals the running time of the slower algorithm.

- Given an instance $I$ and two algorithms $A$ and $B$

<table>
<thead>
<tr>
<th>Algorithm A</th>
<th>Run</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quality</td>
<td></td>
<td>1232</td>
<td>1123</td>
<td>1621</td>
<td>1345</td>
</tr>
<tr>
<td>Running Time</td>
<td></td>
<td>23.2</td>
<td>24.5</td>
<td>21.0</td>
<td>22.5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Algorithm B</th>
<th>Run</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quality</td>
<td></td>
<td>1532</td>
<td>1103</td>
<td>1287</td>
<td>1845</td>
</tr>
<tr>
<td>Running Time</td>
<td></td>
<td>5.2</td>
<td>8.3</td>
<td>6.0</td>
<td>7.3</td>
</tr>
</tbody>
</table>
Effectiveness Tests

- **Idea**: Perform additional runs with the faster algorithm until its expected running time equals the running time of the slower algorithm

- **Given an instance** $I$ and two algorithms $A$ and $B$

<table>
<thead>
<tr>
<th>Algorithm A</th>
<th>Run</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Quality</td>
<td>1232</td>
<td>1123</td>
<td>1621</td>
<td>1345</td>
</tr>
<tr>
<td></td>
<td>Running Time</td>
<td>23.2</td>
<td>24.5</td>
<td>21.0</td>
<td>22.5</td>
</tr>
</tbody>
</table>

**Algorithm A**
- Best Result 1123
- Total Time 24.5

<table>
<thead>
<tr>
<th>Algorithm B</th>
<th>Run</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Quality</td>
<td>1532</td>
<td>1103</td>
<td>1287</td>
<td>1845</td>
</tr>
<tr>
<td></td>
<td>Running Time</td>
<td>5.2</td>
<td>8.3</td>
<td>6.0</td>
<td>7.3</td>
</tr>
</tbody>
</table>

**Algorithm B**
- Best Result 1845
- Total Time 7.3

Sample one run from each algorithm
Effectiveness Tests

**Idea**: Perform additional runs with the faster algorithm until its expected running time equals the running time of the slower algorithm

Given an instance $I$ and two algorithms $A$ and $B$

<table>
<thead>
<tr>
<th>Algorithm A</th>
<th>Run</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quality</td>
<td>1232</td>
<td>1123</td>
<td>1621</td>
<td>1345</td>
<td></td>
</tr>
<tr>
<td>Running Time</td>
<td>23.2</td>
<td>24.5</td>
<td>21.0</td>
<td>22.5</td>
<td></td>
</tr>
</tbody>
</table>

Algorithm A Best Result 1123
Total Time 24.5

Algorithm B

<table>
<thead>
<tr>
<th>Run</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quality</td>
<td>1532</td>
<td>1103</td>
<td>1287</td>
<td>1845</td>
</tr>
<tr>
<td>Running Time</td>
<td>5.2</td>
<td>8.3</td>
<td>6.0</td>
<td>7.3</td>
</tr>
</tbody>
</table>

Sample additional runs of algorithm B

Algorithm B

<table>
<thead>
<tr>
<th>Run</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quality</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Running Time</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Algorithm B Best Result 1456
Total Time 11.6
Effectiveness Tests

**Idea**: Perform additional runs with the faster algorithm until its expected running time equals the running time of the slower algorithm

**Given an instance** \( I \) **and two algorithms** \( A \) **and** \( B \)

**Algorithm A**

<table>
<thead>
<tr>
<th>Run</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quality</td>
<td>1232</td>
<td>1123</td>
<td>1621</td>
<td>1345</td>
</tr>
<tr>
<td>Running Time</td>
<td>23.2</td>
<td>24.5</td>
<td>21.0</td>
<td>22.5</td>
</tr>
</tbody>
</table>

**Algorithm B**

<table>
<thead>
<tr>
<th>Run</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quality</td>
<td>1532</td>
<td>1103</td>
<td>1287</td>
<td>1845</td>
</tr>
<tr>
<td>Running Time</td>
<td>5.2</td>
<td>8.3</td>
<td>6.0</td>
<td>7.3</td>
</tr>
</tbody>
</table>

**Algorithm A**

- Best Result: 1123
- Total Time: 24.5

**Algorithm B**

- Best Result: 1456
- Total Time: 16.8

Sample additional runs of algorithm B
Effectiveness Tests

- **Idea**: Perform additional runs with the faster algorithm until its expected running time equals the running time of the slower algorithm.

- **Given an instance** $I$ and two algorithms $A$ and $B$

<table>
<thead>
<tr>
<th>Algorithm A</th>
<th>Run</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Quality</td>
<td>1232</td>
<td>1123</td>
<td>1621</td>
<td>1345</td>
</tr>
<tr>
<td></td>
<td>Running Time</td>
<td>23.2</td>
<td>24.5</td>
<td>21.0</td>
<td>22.5</td>
</tr>
</tbody>
</table>

**Algorithm A**

- Best Result: 1123
- Total Time: 24.5

16.8 + 8.3 = 25.1 > 24.5

⇒ accept last sample with probability \((24.5 - 16.8)/8.3 = 92\%\)

<table>
<thead>
<tr>
<th>Algorithm B</th>
<th>Run</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Quality</td>
<td>1532</td>
<td>1103</td>
<td>1287</td>
<td>1845</td>
</tr>
<tr>
<td></td>
<td>Running Time</td>
<td>5.2</td>
<td>8.3</td>
<td>6.0</td>
<td>7.3</td>
</tr>
</tbody>
</table>

**Algorithm B**

- Best Result: 1456
- Total Time: 16.8
**Effectiveness Tests**

**Idea**: Perform additional runs with the faster algorithm until its expected running time equals the running time of the slower algorithm

**Given an instance $I$ and two algorithms $A$ and $B$**

<table>
<thead>
<tr>
<th>Algorithm A</th>
<th>Run</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Quality</td>
<td>1232</td>
<td>1123</td>
<td>1621</td>
<td>1345</td>
</tr>
<tr>
<td></td>
<td>Running Time</td>
<td>23.2</td>
<td>24.5</td>
<td>21.0</td>
<td>22.5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Algorithm A</th>
<th>Best Result</th>
<th>1123</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Time</td>
<td>24.5</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Algorithm B</th>
<th>Run</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Quality</td>
<td>1532</td>
<td>1103</td>
<td>1287</td>
<td>1845</td>
</tr>
<tr>
<td></td>
<td>Running Time</td>
<td>5.2</td>
<td>8.3</td>
<td>6.0</td>
<td>7.3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Algorithm B</th>
<th>Best Result</th>
<th>1103</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Time</td>
<td>25.1</td>
<td></td>
</tr>
</tbody>
</table>
Effectiveness Tests

**Idea**: Perform additional runs with the faster algorithm until its expected running time equals the running time of the slower algorithm

Given an instance $I$ and two algorithms $A$ and $B$

<table>
<thead>
<tr>
<th>Algorithm A</th>
<th>Run</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Quality</td>
<td>1232</td>
<td>1123</td>
<td>1621</td>
<td>1345</td>
</tr>
<tr>
<td></td>
<td>Running Time</td>
<td>23.2</td>
<td>24.5</td>
<td>21.0</td>
<td>22.5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Algorithm B</th>
<th>Run</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Quality</td>
<td>1532</td>
<td>1103</td>
<td>1287</td>
<td>1845</td>
</tr>
<tr>
<td></td>
<td>Running Time</td>
<td>5.2</td>
<td>8.3</td>
<td>6.0</td>
<td>7.3</td>
</tr>
</tbody>
</table>

This is also called a *virtual instance* ⇒ we create 10 virtual instances per instance

<table>
<thead>
<tr>
<th>Algorithm A</th>
<th>Best Result</th>
<th>1123</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Time</td>
<td>24.5</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Algorithm B</th>
<th>Best Result</th>
<th>1103</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Time</td>
<td>25.1</td>
<td></td>
</tr>
</tbody>
</table>
Multilevel vs $n$-Level - Effectiveness Tests
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
- great speedups

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