Shared-Memory n-level Hypergraph Partitioning

December 17, 2021
Lars Gottesbüren, Tobias Heuer, Peter Sanders, Sebastian Schlag
Hypergraph Partitioning

- Hypergraph $H = (V, E, c, \omega)$
  - vertex/node set $V = \{1, \ldots, n\}$
  - hyperedge/net set $E \subset \mathcal{P}(V) \setminus \emptyset$
  - incident hyperedges $I(u) = \{e \in E \mid u \in e\}$
  - vertex weights $c : V \to \mathbb{R}_{\geq 1}$
  - hyperedge weights $\omega : E \to \mathbb{R}_{\geq 1}$
Hypergraph Partitioning

- partition $V$ into $k$ blocks $\Pi : V \rightarrow \{ V_1, \ldots, V_k \}$
- blocks $V_i$ with **equal size**
  (tolerated **imbalance** $\varepsilon$):

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

- **minimize connectivity** metric:
  
  \[
  con = \sum_{e \in E} (\lambda(e) - 1)\omega(e)
  \]

  \[
  \lambda(e) = \left| \{ V_i \mid e \cap V_i \neq \emptyset \} \right|
  \]
  
  ($\sim$ number of blocks overlapping with $e$)
Applications

- Distributed Databases
- Route Planning
- VLSI Design
- HPC

Institute of Theoretical Informatics, Algorithmics II
Trade-Off Landscape

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

Sequential
Shared Memory
Distributed
Trade-Off Landscape

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

Sequential
Shared Memory
Distributed

Speed
low...fast

Quality
low...high

what could go here?
Trade-Off Landscape

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

Sequential
Shared Memory
Distributed
Trade-Off Landscape

- KaHyPar-HFC
- KaHyPar-CA
- hMetis-R
- Mt-KaHyPar-Q
- Mt-KaHyPar-D

Similar quality and components as KaHyPar-CA
Faster through parallelism

Sequential
Shared Memory
Distributed

Speed

Quality

Low

High

Slow

Fast
Heuristics: Local Moving

- iterative improvement heuristics KL, FM
- move vertices to improve initial partition
Heuristics: Local Moving

- iterative improvement heuristics KL, FM
- move vertices to improve initial partition
- important class: localized refinement
- initially consider only few seed vertices
- expand around moved vertices
Heuristics: Local Moving

- iterative improvement heuristics KL, FM
- move vertices to improve initial partition
- important class: localized refinement
- initially consider only few seed vertices
- expand around moved vertices
- localization = form of parallelism
- expand non-overlapping
Heuristics: Local Moving

- iterative improvement heuristics KL, FM
- move vertices to improve initial partition
- important class: localized refinement
- initially consider only few seed vertices
- expand around moved vertices

- localization = form of parallelism
- expand non-overlapping
- where to get initial partition?
Multilevel

Input Hypergraph

cluster

contract
Multilevel

Input Hypergraph

cluster

contract

initial partition
Multilevel

Input Hypergraph

cluster

close

contract

uncontract

local search

initial partition

Input Hypergraph

local search

initial partition
Multilevel

- move multiple nodes at once
- more levels $\sim$ more refinement $\sim$ better quality
- traditionally $O(\log(n))$ levels $n = |V|$
Multilevel

- move multiple nodes at once
- more levels → more refinement → better quality
- traditionally $O(\log(n))$ levels $n = |V|$
- what if we did $n$ levels?
- (un)contract one vertex on each level
- trade off for maximum quality
Multilevel

- move multiple nodes at once
- more levels $\sim$ more refinement $\sim$ better quality
- traditionally $O(\log(n))$ levels $n = |V|$
- what if we did $n$ levels?
  - (un)contract one vertex on each level
  - trade off for maximum quality

**Ingredients**
- sub-linear work per level
  $\sim$ localized refinement
- (un)coarsening with dynamic data structures
Multilevel

- move multiple nodes at once
- more levels $\sim$ more refinement $\sim$ better quality
- traditionally $O(\log(n))$ levels $n = |V|$
- what if we did $n$ levels?
- (un)contract one vertex on each level
- trade off for maximum quality

- ingredients
  - sub-linear work per level $\sim$ localized refinement
  - (un)coarsening with dynamic data structures

this talk

Plug in existing work
Multilevel

- move multiple nodes at once
- more levels $\sim$ more refinement $\sim$ better quality
- traditionally $O(\log(n))$ levels $n = |V|$
- what if we did $n$ levels?
- (un)contract one vertex on each level
- trade off for maximum quality

- big challenge: inherently sequential?
- relaxe strict ordering
- tree precedence constraints $\sim$ schedule
- sibling constraints for uncoarsening

ingredients
- sub-linear work per level
  $\sim$ localized refinement
- (un)coarsening with dynamic data structures

this talk
plug in existing work

sync on each level
Sequential Coarsening

Algorithm 1: Sequential Coarsening

\[
\text{while } |V| > C_{\text{max}} \cdot k \text{ do } \\
\quad \text{for } v \in V \text{ in random order do } \\
\quad \quad u \leftarrow \text{findMate}(v) \\
\quad \quad \text{contract } v \text{ onto } u
\]
Sequential Coarsening

Algorithm 1: Sequential Coarsening

while $|V| > C_{\text{max}} \cdot k$ do
  for $v \in V$ in random order do
    $u \leftarrow \text{findMate}(v)$
    contract $v$ onto $u$
  update hypergraph
  $u$ remains, $v$ removed
  $\forall e \in I(v)$
    if $u \notin e$ replace $v$ by $u$ and add $e$ to $I(u)$
    otherwise remove $v$ from $e$
Sequential Coarsening

Algorithm 1: Sequential Coarsening

\[\text{while } |V| > C_{\text{max}} \cdot k \text{ do} \]
\[\text{for } v \in V \text{ in random order do} \]
\[u \leftarrow \text{findMate}(v)\]
\[\text{contract } v \text{ onto } u\]

- update hypergraph
- \(u\) remains, \(v\) removed
- \(\forall e \in I(v)\)
  - if \(u \notin e\) replace \(v\) by \(u\) and add \(e\) to \(I(u)\)
  - otherwise remove \(v\) from \(e\)
Sequential Coarsening

Algorithm 1: Sequential Coarsening

while $|V| > C_{\text{max}} \cdot k$ do
  for $v \in V$ in random order do
    $u \leftarrow \text{findMate}(v)$
    contract $v$ onto $u$
  update hypergraph
  $u$ remains, $v$ removed
  $\forall e \in I(v)$
    if $u \not\in e$ replace $v$ by $u$ and add $e$ to $I(u)$
    otherwise remove $v$ from $e$

- lots of race conditions
- editing pin lists $\Rightarrow$ locks are fine
Sequential Coarsening

Algorithm 1: Sequential Coarsening

while $|V| > C_{\text{max}} \cdot k$ do
  for $v \in V$ in random order do (in parallel?)
    $u \leftarrow \text{findMate}(v)$
    contract $v$ onto $u$

  update hypergraph
  $u$ remains, $v$ removed
  $\forall e \in I(v)$
    if $u \notin e$ replace $v$ by $u$ and add $e$ to $I(u)$
    otherwise remove $v$ from $e$

- lots of race conditions
- editing pin lists $\Rightarrow$ locks are fine

more difficult
- $I(v)$ changes
- while or after $v$ is contracted
Sequential Coarsening

**Algorithm 1: Sequential Coarsening**

while $|V| > C_{\text{max}} \cdot k$ do
  for $v \in V$ in random order do
    $u \leftarrow \text{findMate}(v)$
    contract $v$ onto $u$

**Algorithm 2: Parallel Dependencies**

while $|V| > C_{\text{max}} \cdot k$ do
  for $v \in V$ in random order in parallel
    parent[$v$] $\leftarrow \text{findMate}(v)$
**Sequential Coarsening**

**Algorithm 1: Sequential Coarsening**

```
while |V| > C_{max} \cdot k do
    for v \in V in random order do
        u ← findMate(v)
        contract v onto u
```

**Algorithm 2: Parallel Dependencies**

```
while |V| > C_{max} \cdot k do
    for v \in V in random order in parallel
        parent[v] ← findMate(v)
```

induces forest $\mathcal{F}$
Sequential Coarsening

**Algorithm 1: Sequential Coarsening**

```latex
while |V| > C_{\text{max}} \cdot k do
    for v \in V \text{ in random order do}
        u \leftarrow \text{findMate}(v)
        contract v onto u
```

**Algorithm 2: Parallel Dependencies**

```latex
while |V| > C_{\text{max}} \cdot k do
    for v \in V \text{ in random order in parallel}
        parent[v] \leftarrow \text{findMate}(v)
```

- contract leaves in parallel
- all children finished \( \Rightarrow I(v) \) stable

induces forest \( \mathcal{F} \)
Sequential Coarsening

**Algorithm 1: Sequential Coarsening**

while $|V| > C_{\text{max}} \cdot k$ do
  for $v \in V$ in random order do
    $u \leftarrow \text{findMate}(v)$
    contract $v$ onto $u$

**Algorithm 2: Parallel Dependencies**

while $|V| > C_{\text{max}} \cdot k$ do
  for $v \in V$ in random order in parallel
    parent[$v$] $\leftarrow \text{findMate}(v)$

induces forest $\mathcal{F}$

- contract leaves in parallel
- all children finished $\sim I(v)$ stable
- asynchronous
- dynamic rooted forest
- fancy locking protocol $\Rightarrow$ paper
Parallel Coarsening

Algorithm 3: Parallel Coarsening

while $|V| > C_{\text{max}} \cdot k$ do
  for $v \in V$ in random order in parallel
    $u \leftarrow \text{findMate}(v)$
    $u, \text{parent}[v] \leftarrow \text{findSafeAncestor}(u)$
    pending[$u$] += 1
  while $u \neq v$ and pending[$v$] = 0 do
    contract $v$ onto $u$
    pending[$u$] -= 1
    $v \leftarrow u, u \leftarrow \text{parent}[u]$
Parallel Coarsening

Algorithm 3: Parallel Coarsening

\begin{algorithm}
\begin{algorithmic}
\WHILE{$|V| > C_{\text{max}} \cdot k$}
    \FOR{$v \in V$ \textit{in random order} \textbf{in parallel}}
        \STATE $u \leftarrow \text{findMate}(v)$
        \STATE $u, \text{parent}[v] \leftarrow \text{findSafeAncestor}(u)$
        \STATE $\text{pending}[u] += 1$
    \WHILE{$u \neq v$ \textbf{and} $\text{pending}[v] = 0$}
        \STATE contract $v$ onto $u$
        \STATE $\text{pending}[u] -= 1$
        \STATE $v \leftarrow u, u \leftarrow \text{parent}[u]$
    \ENDWHILE
    \STATE $v \leftarrow u, u \leftarrow \text{parent}[u]$
\ENDFOR
\ENDWHILE
\end{algorithmic}
\end{algorithm}
Parallel Coarsening

Algorithm 3: Parallel Coarsening

while $|V| > C_{\text{max}} \cdot k$ do
  for $v \in V$ in random order in parallel
    $u \leftarrow \text{findMate}(v)$
    $u, \text{parent}[v] \leftarrow \text{findSafeAncestor}(u)$
    pending[$u$] $\leftarrow 1$
    while $u \neq v$ and pending[$v$] $= 0$ do
      contract $v$ onto $u$
      pending[$u$] $\leftarrow 1$
      $v \leftarrow u, u \leftarrow \text{parent}[u]$

walk up tree
parent already contracted $\Rightarrow$ replace with ancestor

walk up tree
parent already contracted $\Rightarrow$ replace with ancestor
avoid cycles $\Rightarrow$ may discard
in most cases $u$ is the suggested mate
Parallel Coarsening

Algorithm 3: Parallel Coarsening

while $|V| > C_{\text{max}} \cdot k$ do
  for $v \in V$ in random order in parallel
    $u \leftarrow \text{findMate}(v)$
    $u, \text{parent}[v] \leftarrow \text{findSafeAncestor}(u)$
    pending[$u$] += 1
    while $u \neq v$ and pending[$v$] = 0 do
      contract $v$ onto $u$
      pending[$u$] -= 1
      $v \leftarrow u, u \leftarrow \text{parent}[u]$
    end while
  end for
end while

- walk up tree
- parent already contracted $\Rightarrow$ replace with ancestor
- already contracted
- want to contract now
Parallel Coarsening

Algorithm 3: Parallel Coarsening

\[
\text{while } |V| > C_{\text{max}} \cdot k \text{ do}
\]

\[
\text{for } v \in V \text{ in random order in parallel}
\]

\[
\quad u \leftarrow \text{findMate}(v)
\]

\[
\quad u, \text{parent}[v] \leftarrow \text{findSafeAncestor}(u)
\]

\[
\quad \text{pending}[u] += 1
\]

\[
\text{while } u \neq v \text{ and pending}[v] = 0 \text{ do}
\]

\[
\quad \text{contract } v \text{ onto } u
\]

\[
\quad \text{pending}[u] -= 1
\]

\[
\quad v \leftarrow u, u \leftarrow \text{parent}[u]
\]

- walk up tree
- parent already contracted \(\Rightarrow\) replace with ancestor
- avoid cycles \(\Rightarrow\) may discard
- in most cases \(u\) is the suggested mate
Algorithm 3: Parallel Coarsening

\[
\text{while } |V| > C_{\text{max}} \cdot k \text{ do}
\]

\[
\text{for } v \in V \text{ in random order in parallel}
\]

\[
u \leftarrow \text{findMate}(v)
\]

\[
u, \text{parent}[v] \leftarrow \text{findSafeAncestor}(u)
\]

\[
pending[u] += 1
\]

\[
\text{while } u \neq v \text{ and pending}[v] = 0 \text{ do}
\]

\[
\text{contract } v \text{ onto } u
\]

\[
pending[u] -= 1
\]

\[
v \leftarrow u, u \leftarrow \text{parent}[u]
\]

- walk up tree
- parent already contracted ⇒ replace with ancestor
- avoid cycles ⇒ may discard
- in most cases \(u\) is the suggested mate
- detect potential contraction start
- pending\([v]\) = 0 ∧ parent\([v]\) ≠ \(v\)
Parallel Coarsening

Algorithm 3: Parallel Coarsening

\[
\text{while } |V| > C_{\text{max}} \cdot k \text{ do} \\
\text{for } v \in V \text{ in random order in parallel} \\
\quad u \leftarrow \text{findMate}(v) \\
\quad u, \text{parent}[v] \leftarrow \text{findSafeAncestor}(u) \\
\quad \text{pending}[u] += 1 \\
\quad \textbf{while } u \neq v \text{ and pending}[v] = 0 \text{ do} \\
\quad \quad \text{contract } v \text{ onto } u \\
\quad \quad \text{pending}[u] -= 1 \\
\quad \quad v \leftarrow u, u \leftarrow \text{parent}[u]
\]
Algorithm 3: Parallel Coarsening

\[\text{while } |V| > C_{\text{max}} \cdot k \text{ do} \]
\[\quad \text{for } v \in V \text{ in random order in parallel} \]
\[\quad \quad u \leftarrow \text{findMate}(v) \]
\[\quad \quad u, \text{parent}[v] \leftarrow \text{findSafeAncestor}(u) \]
\[\quad \quad \text{pending}[u] += 1 \]
\[\quad \textbf{while } u \neq v \text{ and pending}[v] = 0 \textbf{ do} \]
\[\quad \quad \quad \text{contract } v \text{ onto } u \]
\[\quad \quad \quad \text{pending}[u] -= 1 \]
\[\quad \quad \quad v \leftarrow u, u \leftarrow \text{parent}[u] \]

- enforce bottom-up
- pending[v] = 0 \Rightarrow we can go
- pending[v] \neq 0 \Rightarrow transfer responsibility
Parallel Coarsening

Algorithm 3: Parallel Coarsening

while $|V| > C_{\text{max}} \cdot k$ do
  for $v \in V$ in random order in parallel
    $u \leftarrow \text{findMate}(v)$
    $u, \text{parent}[v] \leftarrow \text{findSafeAncestor}(u)$
    pending[$u$] += 1
    while $u \neq v$ and pending[$v$] = 0 do
      contract $v$ onto $u$
      pending[$u$] -= 1
      $v \leftarrow u, u \leftarrow \text{parent}[u]$

- enforce bottom-up
- pending[$v$] = 0 $\Rightarrow$ we can go
- pending[$v$] $\neq 0$ $\Rightarrow$ transfer responsibility
- repeat for parent if none pending
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
- no strict uncontraction order imposed by coarsening
Parallel Uncoarsening

- traditional \( n \)-level uncontracts only one vertex on each level \( \Rightarrow \) inherently sequential
- no strict uncontraction order imposed by coarsening

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 \( B_{i+1} \)
Parallel Uncoarsening

- traditional $n$-level uncontracts only one vertex on each level $\Rightarrow$ inherently sequential
- no strict uncontraction order imposed by coarsening

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 $\mathcal{F}$

$$b_{\text{max}} = 3$$
$$\mathcal{B} = \langle \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rangle$$
Parallel Uncoarsening

- traditional \( n \)-level uncontracts only one vertex on each level \( \Rightarrow \) inherently sequential
- no strict uncontraction order imposed by coarsening

**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 \( \mathcal{F} \)

\[
\begin{align*}
b_{\text{max}} = 3 \\
B = \langle [v_3, v_7, v_4], [v_5], [v_6], [v_8], \rangle
\end{align*}
\]
Parallel Uncoarsening

- Traditional $n$-level uncontracts only one vertex on each level ⇒ inherently sequential
- No strict uncontraction order imposed by coarsening

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 $\mathcal{F}$

$b_{\text{max}} = 3$
$\mathcal{B} = \langle V_3, V_7, V_4, V_5, V_6, V_{12}, \ldots \rangle$
Parallel Uncoarsening

- traditional \( n \)-level uncontracts only one vertex on each level \( \Rightarrow \) inherently sequential
- no strict uncontraction order imposed by coarsening

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 \( \mathcal{F} \)

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

\( \bullet \) eligible for uncontraction
\( \bullet \) already uncontracted
Parallel Uncoarsening

- traditional $n$-level uncontracts only one vertex on each level $\Rightarrow$ inherently sequential
- no strict uncontraction order imposed by coarsening

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 $\mathcal{F}$

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

- traditional $n$-level uncontracts only one vertex on each level ⇒ inherently sequential
- no strict uncontraction order imposed by coarsening

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 $\mathcal{F}$

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

- traditional *n*-level uncontracts only **one** vertex on each level ⇒ inherently sequential
- no strict uncontraction order imposed by coarsening

**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 \( \mathcal{F} \)

\[
b_{\text{max}} = 3
\]

\[
B = \langle v_3, v_7, v_4, v_5, v_6, v_{12}, v_8, v_9, v_{10}, v_{11}, v_{13}, v_{14}, v_{15} \rangle
\]

\( b_{\text{max}} = 1000 \) in practice

- green: eligible for uncontraction
- red: already uncontracted
Sibling Ordering Constraints

- sequential n-level: uncontractions in reverse order
- parallel n-level: parents before children + siblings in reverse order
Sibling Ordering Constraints

- sequential n-level: uncontractions in reverse order
- parallel n-level: parents before children + **siblings in reverse order**
- first pin is replaced by the parent, second is removed
- \(\sim\) different ordering messes up data structure
Sibling Ordering Constraints

- sequential n-level: uncontractions in reverse order
- parallel n-level: parents before children + siblings in reverse order
- first pin is replaced by the parent, second is removed
- different ordering messes up data structure
- contracted at the same time and $\geq 2$ shared hyperedges $e_1, e_2$
- $u$ replaced in $e_1$, removed in $e_2$, $v$ removed in $e_1$, replaced in $e_2$
- must go in the same batch

Diagram:

- Nodes: $u, v, w, y, e, u, x$
- Hyperedges: $e$
- Ordering: $u \rightarrow v \rightarrow w$
Sibling Ordering Constraints

- sequential n-level: uncontractions in reverse order
- parallel n-level: parents before children + siblings in reverse order
- first pin is replaced by the parent, second is removed
- → different ordering messes up data structure
Sibling Ordering Constraints

- sequential n-level: uncontractions in reverse order
- parallel n-level: parents before children + **siblings in reverse order**
- first pin is replaced by the parent, second is removed
- ~ different ordering messes up data structure

![Diagram showing sibling ordering constraints](image-url)
Sibling Ordering Constraints

- sequential n-level: uncontractions in reverse order
- parallel n-level: parents before children + **siblings in reverse order**
- first pin is replaced by the parent, second is removed
- $\sim$ different ordering messes up data structure

```
contract v onto u  contract w onto u
```

![Diagrams showing the process of contract v onto u and contract w onto u.](image)
Sibling Ordering Constraints

- sequential n-level: uncontractions in reverse order
- parallel n-level: parents before children + siblings in reverse order
- first pin is replaced by the parent, second is removed
- \( \sim \) different ordering messes up data structure
Sibling Ordering Constraints

- sequential n-level: uncontractions in reverse order
- parallel n-level: parents before children + siblings in reverse order
- first pin is replaced by the parent, second is removed
- \( \sim \) different ordering messes up data structure

![Diagram showing sibling ordering constraints](image)
Sibling Ordering Constraints

- sequential n-level: uncontractions in reverse order
- parallel n-level: parents before children + siblings in reverse order
- first pin is replaced by the parent, second is removed
- \( \sim \) different ordering messes up data structure
Sibling Ordering Constraints

- sequential n-level: uncontractions in reverse order
- parallel n-level: parents before children + siblings in reverse order
- first pin is replaced by the parent, second is removed
- \( \sim \) different ordering messes up data structure

![Diagram of sibling ordering constraints with nodes and edges illustrating the process of contracting and uncontracting nodes, moving elements between partitions, and maintaining the ordering constraints.]
Sibling Ordering Constraints

- sequential n-level: uncontractions in reverse order
- parallel n-level: parents before children + siblings in reverse order
- first pin is replaced by the parent, second is removed
- different ordering messes up data structure

uncontracting w increases cut
⇒ fundamental multilevel property violated
Sibling Ordering Constraints

Solution

- track sibling order with atomic counters
Sibling Ordering Constraints

Solution

- track sibling order with atomic counters
- \( \sim \) time interval \([s_v, e_v]\) for each contracted vertex
Sibling Ordering Constraints

Solution

- track sibling order with atomic counters
- $\sim$ time interval $[s_v, e_v]$ for each contracted vertex
- transitive closure of overlapping time intervals

![Diagram showing sibling ordering constraints with vertices and time intervals]


- The vertices must be uncontracted in the same batch.
- Uncontract in reverse order.
Sibling Ordering Constraints

Solution
- track sibling order with atomic counters
- $\sim$ time interval $[s_v, e_v]$ for each contracted vertex
- transitive closure of overlapping time intervals

uncontract in reverse order

must be uncontracted in the same batch

very few nodes have siblings in $\mathcal{F}$ $\Rightarrow$ occurs rarely in practice $\Rightarrow$ small overhead
Additional Challenges

- gain table for parallel localized FM
- updated when nodes are moved
- now requires updates when uncontracting

<table>
<thead>
<tr>
<th>node</th>
<th>target block</th>
<th>gain</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Additional Challenges

- Gain table for parallel localized FM
- Updated when nodes are moved
- Now requires updates when uncontracting

- Dynamic hypergraph data structure
- Edit pin arrays
- Link and edit incident hyperedge arrays

Gain Table

<table>
<thead>
<tr>
<th>node</th>
<th>target block</th>
<th>gain</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>1</th>
<th>3</th>
<th>2</th>
<th>4</th>
<th>0</th>
<th>3</th>
<th>6</th>
<th>4</th>
<th>2</th>
<th>5</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>e0</td>
<td>1</td>
<td>0</td>
<td></td>
<td>3</td>
<td>0</td>
<td></td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>e1</td>
<td>0</td>
<td>1</td>
<td></td>
<td>0</td>
<td>1</td>
<td></td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>e2</td>
<td>0</td>
<td>0</td>
<td></td>
<td>2</td>
<td>0</td>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>e3</td>
<td>0</td>
<td>0</td>
<td></td>
<td>0</td>
<td>0</td>
<td></td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>
```

Gain Table

`t_u,e`

```
<table>
<thead>
<tr>
<th></th>
<th>v0</th>
<th>v1</th>
<th>v2</th>
<th>v3</th>
<th>v4</th>
<th>v5</th>
<th>v6</th>
</tr>
</thead>
<tbody>
<tr>
<td>t_u</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>L_u</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Additional Challenges

- Gain table for parallel localized FM
- Updated when nodes are moved
- Now requires updates when uncontracting

- Dynamic hypergraph data structure
- Edit pin arrays
- Link and edit incident hyperedge arrays

- Identical hyperedge detection + removal

Gain Table

<table>
<thead>
<tr>
<th>node</th>
<th>target block</th>
<th>gain</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Gain Table:

- Gain table for parallel localized FM
- Updated when nodes are moved
- Now requires updates when uncontracting

- Dynamic hypergraph data structure
- Edit pin arrays
- Link and edit incident hyperedge arrays

- Identical hyperedge detection + removal
Experiments – Big Bench

- 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 \cdot 3 = 42
  - DAC2012 VLSI Circuits 10
- 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
Experiments – Scalability

![Graphs showing scalability results for different components and their speedup](image_url)
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 \( \geq 100 \text{s} \):
  - 3.7 with 4 threads
  - 12.3 with 16 threads
  - 25 with 64 threads
Experiments – Connectivity Metric (Quality)
Experiments – Connectivity Metric (Quality)

\[ \tau = 1 \Leftrightarrow \text{fraction of instances for which algorithm finds the best partition} \]

\[ \text{Mt-KaHyPar-Q} \geq 70\% \]
Experiments – Connectivity Metric (Quality)

\[ \tau = 1 \leftrightarrow \text{fraction of instances for which algorithm finds the best partition} \]

- Mt-KaHyPar-Q $\geq 70\%$
- Zoltan $\leq 5\%$

\[ \tau \]

- Mt-KaHyPar-Q 64
- PaToH-D
- Mt-KaHyPar-D 64
- Hype
- Zoltan 64
- BiPart 64
Experiments – Connectivity Metric (Quality)

\[ \tau = 1.1 \Leftrightarrow \text{fraction for which algorithm performs at most 10\% worse than the best} \]
Experiments – Connectivity Metric (Quality)

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Gmean t[s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mt-KaHyPar-D 64</td>
<td>4.89</td>
</tr>
<tr>
<td>Zoltan 64</td>
<td>12.63</td>
</tr>
<tr>
<td>Hype</td>
<td>25.56</td>
</tr>
<tr>
<td>BiPart 64</td>
<td>29.19</td>
</tr>
<tr>
<td>Mt-KaHyPar-Q 64</td>
<td>30.7</td>
</tr>
<tr>
<td>PaToH-D</td>
<td>51.2</td>
</tr>
</tbody>
</table>

\( \tau \)
- Mt-KaHyPar-Q 64
- PaToH-D
- Mt-KaHyPar-D 64
- Hype
- Zoltan 64
- BiPart 64
Experiments – Medium Bench

- 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 42
  - SAT Competition 2014 (3 representations) $14 \cdot 3 = 42$
  - DAC2012 VLSI Circuits 10

- $k \in \{2, 4, 8, 16, 32, 64, 128\}$ with imbalance: $\varepsilon = 3\%$
- 10 random seeds
- 1, 10, 20 threads
Experiments – Connectivity Metric (Quality)
Experiments – Connectivity Metric (Quality)

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Gmean t[s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mt-KaHyPar-D 10</td>
<td>0.95</td>
</tr>
<tr>
<td>PaToH-D</td>
<td>1.17</td>
</tr>
<tr>
<td>Mt-KaHyPar-Q 10</td>
<td>3.19</td>
</tr>
<tr>
<td>PaToH-Q</td>
<td>5.86</td>
</tr>
<tr>
<td>KaHyPar-CA</td>
<td>28.14</td>
</tr>
</tbody>
</table>
Lars Gottesbüren – Shared-Memory n-level Hypergraph Partitioning

Institute of Theoretical Informatics, Algorithmics II

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

Teaser
<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><strong>Mt-KaHyPar-Q-F 10</strong></td>
<td><strong>5.08</strong></td>
</tr>
<tr>
<td>KaHyPar-HFC</td>
<td>48.98</td>
</tr>
</tbody>
</table>

![Graph showing performance comparison]

**Teaser**
Conclusion

Mt-KaHyPar-Q

- parallel $n$-level (un)coarsening
- contraction forest $\sim$ schedule
- similar quality as KaHyPar-CA, 10x faster with 10 threads
- great speedups

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

Mt-KaHyPar-Q
- parallel $n$-level (un)coarsening
- contraction forest $\rightarrow$ schedule
- similar quality as KaHyPar-CA, 10x faster with 10 threads
- great speedups

Future Work (already done)
- asynchronous uncoarsening
- flow-based refinement

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

Mt-KaHyPar-Q
- parallel $n$-level (un)coarsening
- contraction forest $\sim$ schedule
- similar quality as KaHyPar-CA, 10x faster with 10 threads
- great speedups

Future Work (already done)
- asynchronous uncoarsening
- flow-based refinement

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

no batches
- simultaneously uncontract and refine
- slightly better speedups, slightly worse quality
- more difficult constraints
- more difficult interference
Conclusion

Mt-KaHyPar-Q

- parallel \( n \)-level (un)coarsening
- contraction forest \( \sim \) schedule
- similar quality as KaHyPar-CA, 10x faster with 10 threads
- great speedups

Future Work (already done)

- asynchronous uncoarsening
- flow-based refinement
- parallel max flow is difficult
- \( \log(n) \)-level + flows beats \( n \)-level
- and almost as good as \( n \)-level + flows
- as good as sequential KaHyPar-HFC