Scalable Shared-Memory Hypergraph Partitioning

ALENEX’21 · December 30, 2020
Lars Gottesbüren, Tobias Heuer, Peter Sanders, Sebastian Schlag
Hypergraphs

- generalization of graphs ➔ hyperedges connect ≥ 2 nodes

- graphs ➔ dyadic (2-ary) relationships
- hypergraphs ➔ (d-ary) relationships

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

- generalization of graphs
  ⇒ hyperedges connect ≥ 2 nodes

- graphs ⇒ dyadic (2-ary) relationships
- hypergraphs ⇒ (d-ary) relationships

- hypergraph $H = (V, E, c, \omega)$
  - vertex set $V = \{1, \ldots, n\}$
  - 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 \textbf{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:**
ε-Balanced Hypergraph Partitioning Problem

Partition hypergraph $H = (V, E, c, ω)$ into $k$ disjoint blocks $Π = \{ V_1, \ldots, V_k \}$ 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**:
  \[
  \sum_{e \in \text{cut}} (\lambda - 1) \omega(e)
  \]

**imbalance parameter**

**connectivity:**

# blocks connected by net $e$
$\varepsilon$-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:
  \[ \sum_{e \in \text{cut}} (\lambda - 1) \omega(e) = 12 \]

**Imbalance parameter**

**Connectivity**: # blocks connected by net $e$
Applications

- VLSI Design
- Warehouse Optimization
- Complex Networks
- Route Planning
- Simulation
- Scientific Computing

\[ \mathbb{R}^{n \times n} \ni Ax = b \in \mathbb{R}^n \]
Hypergraph Partitioner Landscape

Why do we need yet another hypergraph partitioner?
Why do we need yet another hypergraph partitioner?
Why do we need yet another hypergraph partitioner?
Hypergraph Partitioner Landscape

Why do we need yet another hypergraph partitioner?
Hypergraph Partitioner Landscape
Why do we need yet another hypergraph partitioner?

- KaHyPar-HFC
- KaHyPar-CA
- hMetis-R
- PaToH-Q
- PaToH-D
- Zoltan

- 2 times slower
- 4 times slower
- 5 times slower
Hypergraph Partitioner Landscape
Why do we need yet another hypergraph partitioner?

First Shared-Memory Hypergraph Partitioner

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

Quality

Speed

low

high

slow

fast
Multilevel Paradigm

Input Hypergraph

match /

cluster

contract

Coarsening
Multilevel Paradigm

Input Hypergraph

Coarsening

match /

cluster

contract

Initial Partitioning
Multilevel Paradigm

Input Hypergraph

Coarsening

match /

cluster

contract

output partition

Uncoarsening

local search

uncontract

Initial Partitioning
Mt-KaHyPar: Algorithmic Overview

Input Hypergraph

Coarsening

match / cluster

contract

Uncoarsening

local search

uncontract

Output Partition

Initial Partitioning
Mt-KaHyPar: Algorithmic Overview

Parallel Community Detection

Parallel Coarsening based on Clustering

Output Partition

local search

uncontract

Initial Partitioning
Mt-KaHyPar: Algorithmic Overview

Parallel Community Detection

- bipartite graph representation

Parallel Coarsening based on Clustering

- on-the-fly clustering
- conflict resolution
- parallel contraction

Parallel Recursive Bipartitioning based Initial Partitioning with Work-Stealing

- $k = 4$
- $V_1$ $V_2$

Task Queue

- Thread 1 $C_1 B_2$
- Thread 2 $C_1 B_2 B_2 B_2$
- Thread 3 $B_2 B_2$
- Thread 4 $B_2 B_2$

Task Queue

- Thread 1 $C_1 B_2$
- Thread 2 $C_1 B_2$
- Thread 3 $B_2 B_2$
- Thread 4 $B_2 B_2$
Mt-KaHyPar: Algorithmic Overview

Parallel Community Detection

Parallel Coarsening based on Clustering

Parallel Recursive Bipartitioning based Initial Partitioning with Work-Stealing

Scalable Label Propagation Refinement + First Fully-Parallel Direct $k$-Way FM Algorithm
Parallel Community Detection

- restrict contractions to densely connected areas. KaHyPar-CA [HS18]
Parallel Community Detection

- restrict contractions to densely connected areas. KaHyPar-CA [HS18]

![Diagram of parallel community detection](image)

- modularity maximization on the bipartite graph representation
- parallel Louvain method [MS16]
Parallel Coarsening based on Clustering

- Initially each vertex is in its own cluster.
Parallel Coarsening based on Clustering

- Initially each vertex is in its own cluster
- Iterate over vertices in parallel
- Vertex $u$ joins cluster $C$ that maximizes heavy-edge rating:

$$r(u, C) := \sum_{e \in I(u) \cap I(C)} \frac{\omega(e)}{|e| - 1}$$
Parallel Coarsening based on Clustering

- Initially each vertex is in its own cluster.

- Iterate over vertices in parallel.

- Vertex $u$ joins cluster $C$ that maximizes heavy-edge rating:
  \[
  r(u, C) := \sum_{e \in I(u) \cap I(C)} \frac{\omega(e)}{|e| - 1}
  \]

- Contract clusters in parallel:
  - Removes single-vertex nets and aggregates the weight of identical nets.
Parallel Coarsening based on Clustering

- Initially each vertex is in its own cluster

- Iterate over vertices in parallel
- Vertex $u$ joins cluster $C$ that maximizes heavy-edge rating:
  
  $$ r(u, C) := \sum_{e \in I(u) \cap I(C)} \frac{\omega(e)}{|e| - 1} $$

- Contract clusters in parallel
  - Removes single-vertex nets and aggregates the weight of identical nets

- Repeat until the number of vertices is $\approx 160k$ or no vertex changed its cluster
Parallel Clustering
Conflict Resolution

- issue: oscillation

I want to join $v$ in $T_1$
I want to join $u$ in $T_2$
Parallel Clustering
Conflict Resolution

- issue: oscillation

- vertex states:
  - **SINGLETON**: initial state of each vertex
  - **JOINING**: vertex wants to join a cluster
  - **CLUSTERED**: final state. vertex cannot change its cluster

- transition with atomic compare-and-swap operations
Parallel Clustering
Conflict Resolution

- issue: oscillation

- vertex states:
  - **SINGLETON**: initial state of each vertex
  - **JOINING**: vertex wants to join a cluster
  - **CLUSTERED**: final state. vertex cannot change its cluster

- transition with atomic compare-and-swap operations

- **SINGLETON** vertex $u$ can join cluster of leader $v$ if
  - $v$ is already **CLUSTERED** or
  - successful compare-and-swap on state of $u$ and $v$ from **SINGLETON** to **JOINING**
Parallel Clustering
Conflict Resolution

JOINING $T_1$

JOINING $T_2$

SINGLETON
Parallel Clustering
Conflict Resolution
Parallel Clustering

Conflict Resolution

Joining $T_1$ busy waiting $T_2$

Joining $T_1$ busy waiting $T_2$ busy waiting $T_3$

Intentional deadlock!
Parallel Clustering
Conflict Resolution

- perform cycle detection during busy waiting
  ⇒ smallest vertex ID in cycle resolves the conflict
Parallel Clustering
Conflict Resolution

- perform cycle detection during busy waiting
  ⇒ smallest vertex ID in cycle resolves the conflict

 intentional deadlock!

\[ U \rightarrow V \rightarrow W \quad \text{JOINING} \]
\[ T_1 \quad \text{busy waiting} \quad T_2 \]

\[ U \rightarrow V \rightarrow W \quad \text{JOINING} \]
\[ T_1 \quad \text{busy waiting} \quad T_2 \quad \text{busy waiting} \quad T_3 \]
Parallel Clustering
Conflict Resolution

- perform cycle detection during busy waiting
  ⇒ smallest vertex ID in cycle resolves the conflict

intentional deadlock!
Overview

Coarsening

- match /
- cluster
- contract

Uncoarsening

- local search
- uncontract

Input Hypergraph

Output Partition

Initial Partitioning
Parallel Initial Partitioning

- Parallel Multilevel Recursive Bipartitioning
  - use our parallel coarsening and refinement
  - bipartition with a portfolio of 9 different algorithms
    - Random
    - BFS-based
    - Greedy Partition Growing (6 variants)
    - Label Propagation

Partition $\Pi = \{V_1, V_2, V_3, V_4\}$
Parallel Initial Partitioning

- Parallel Multilevel Recursive Bipartitioning
  - use our parallel coarsening and refinement
  - bipartition with a portfolio of 9 different algorithms
    - Random
    - BFS-based
    - Greedy Partition Growing (6 variants)
    - Label Propagation

- work-stealing to deal with load imbalance in recursion
- implement everything with work-stealing based parallel primitives of tbb library
  ⇒ task scheduler takes care of everything

Input Hypergraph

Partition $\Pi = \{V_1, V_2, V_3, V_4\}$
Overview

Input Hypergraph

Coarsening

match / cluster

contract

Uncoarsening

Output Partition

local search

uncontract

Initial Partitioning

13 Lars Gottesbüren, Tobias Heuer, Peter Sanders, Sebastian Schlag – Scalable Shared-Memory Hypergraph Partitioning
Institute of Theoretical Informatics
Label Propagation

Algorithm 1: Label Propagation

for each \( u \in V \) in random order in parallel
- calculate gains to all blocks in neighborhood of \( u \)
- move \( u \) to best block if still balanced and gain \( > 0 \)
Label Propagation

Algorithm 1: Label Propagation

for each \( u \in V \) in random order in parallel
\[
\begin{align*}
&\text{calculate gains to all blocks in neighborhood of } u \\
&\text{move } u \text{ to best block if still balanced and gain } > 0
\end{align*}
\]

- not particularly sophisticated. why still use it?
- faster than FM, inherently parallel
- catches easy improvements \( \Rightarrow \) FM converges faster
Label Propagation

Algorithm 1: Label Propagation

for each $u \in V$ in random order in parallel
  calculate gains to all blocks in neighborhood of $u$
  move $u$ to best block if still balanced and gain $> 0$

- not particularly sophisticated. why still use it?
- faster than FM, inherently parallel
- catches easy improvements $\Rightarrow$ FM converges faster
- issue: individually correct gains become wrong when combined

$u \quad v$

I want to be $T_1$
I want to be $T_2$
Label Propagation

Algorithm 1: Label Propagation

\[ \text{for each } u \in V \text{ in random order in parallel} \]
\[ \text{calculate gains to all blocks in neighborhood of } u \]
\[ \text{move } u \text{ to best block if still balanced and gain } > 0 \]

- not particularly sophisticated. why still use it?
- faster than FM, inherently parallel
- catches easy improvements \( \Rightarrow \) FM converges faster

- issue: individually correct gains become wrong when combined
- our contribution: attributed gains
  - tracks improvement through atomic updates to \( |e \cap V_i| \)
  - secondary check: revert move if attributed gain \(< 0\)
Label Propagation

Algorithm 1: Label Propagation

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

\begin{itemize}
  \item calculate gains to all blocks in neighborhood of \( u \)
  \item move \( u \) to best block if still balanced and gain > 0
\end{itemize}

- not particularly sophisticated. why still use it?
- faster than FM, inherently parallel
- catches easy improvements \( \Rightarrow \) FM converges faster

- issue: individually correct gains become wrong when combined

- our contribution: attributed gains
  \begin{itemize}
    \item tracks improvement through atomic updates to \( |e \cap V_i| \)
    \item secondary check: revert move if attributed gain < 0
  \end{itemize}

- common techniques / folklore
  \begin{itemize}
    \item active node set: only consider node if neighbor was moved in previous round
    \item guarantee balance constraint with atomic fetch-and-add
  \end{itemize}
Fiduccia-Mattheyses

Algorithm 2: FM Local Search

while improvement found do
    while not done do
        find best move
        lock vertex
        perform best move
        rollback to best solution
    pass

connectivity

vertex moves
Algorithm 2: FM Local Search

while improvement found do
  while not done do
    find best move
    lock vertex
    perform best move
    rollback to best solution
  pass

connectivity vs. vertex moves

pass 1
pass 2
rollback
Algorithm 2: FM Local Search

\[
\text{while improvement found do} \\
\quad \text{while not done do} \\
\qquad \text{find best move} \\
\qquad \text{lock vertex} \\
\qquad \text{perform best move} \\
\quad \text{rollback to best solution}
\]

- gain table
- priority queue(s)
Algorithm 2: FM Local Search

while improvement found do
  while not done do
    find best move
    lock vertex
    perform best move
  rollback to best solution

- gain table
- priority queue(s)
- start with boundary nodes and expand
Algorithm 2: FM Local Search

while improvement found do
    while not done do
        find best move
        lock vertex
        perform best move
    rollback to best solution

- gain table
- priority queue(s)
- start with boundary nodes and expand
- each thread expands around a few boundary vertices
- searches do not overlap. threads acquire and own their vertices
Algorithm 2: FM Local Search

while improvement found do
  while not done do
    find best move
    lock vertex
    perform best move
  rollback to best solution

- gain table
- priority queue(s)
- start with boundary nodes and expand
- each thread expands around a few boundary vertices
- searches do not overlap. threads acquire and own their vertices

← maintain with atomic fetch-and-add
← handles shared across threads
Algorithm 2: FM Local Search

while improvement found do
    while not done do
        find best move
        lock vertex
        perform best move
    rollback to best solution

- gain table
- priority queue(s)
- start with boundary nodes and expand
- each thread expands around a few boundary vertices
- searches do not overlap. threads acquire and own their vertices
- avoid synchronization
- instead lazily adapt to changes on global partition

← maintain with atomic fetch-and-add
← handles shared across threads
Algorithm 3: Parallel Localized FM

while improvement found do
  do on each thread
    while seeds ← workqueue.tryPop(x) do
      insert seeds into thread-local PQs
    while not done do
      find best move in PQs
      perform best move
      claim and insert/update neighbors into PQs
      rollback to best solution of thread-local search
  rollback to best solution

Algorithm 3: Parallel Localized FM

while improvement found do
  do on each thread
    while seeds ← workqueue.tryPop(x) do
      insert seeds into thread-local PQs
    while not done do
      find best move in PQs
      perform best move
      claim and insert/update neighbors into PQs
    rollback to best solution of thread-local search
  rollback to best solution
  retry if gain table does not match PQ entry
  gain table look up
Algorithm 3: Parallel Localized FM

while improvement found do
  do on each thread
    while seeds ← workqueue.tryPop(x) do
      insert seeds into thread-local PQs
      while not done do
        find best move in PQs
        perform best move
        claim and insert/update neighbors into PQs
        rollback to best solution of thread-local search
    apply locally
  apply to shared partition
  rollback to best solution
Parallel Fiduccia-Mattheyses

**Algorithm 3: Parallel Localized FM**

```plaintext
while improvement found do
    do on each thread
        while seeds ← workqueue.tryPop(x) do
            insert seeds into thread-local PQs
        while not done do
            find best move in PQs
            perform best move
            claim and insert/update neighbors into PQs
            rollback to best solution of thread-local search
        rollback to best solution
```

- **apply to shared partition**
  - cheaper
  - local rollback may confuse other searches

- **apply locally**
  - gain table changes in hash tables
  - rollback applies to shared partition
  - attributed gains → better checks
Parallel Fiduccia-Mattheyses

Algorithm 3: Parallel Localized FM

while improvement found do
  do on each thread
    while seeds ← workqueue.tryPop(x) do
      insert seeds into thread-local PQs
      while not done do
        find best move in PQs
        perform best move
        claim and insert/update neighbors into PQs
        rollback to best solution of thread-local search

Challenges
  - move order?
  - balance constraint
  - incorrect gains
Algorithm 3: Parallel Localized FM

while improvement found do
  do on each thread
    while seeds ← workqueue.tryPop(x) do
      insert seeds into thread-local PQs
    while not done do
      find best move in PQs
      perform best move
      claim and insert/update neighbors into PQs
      rollback to best solution of thread-local search

Challenges
- move order?
- balance constraint
- incorrect gains

← as applied to shared partition. use atomics
Parallel Fiduccia-Mattheyses

Algorithm 3: Parallel Localized FM

\[
\text{while improvement found do}
\]
\[
\text{do on each thread}
\]
\[
\text{while seeds} \leftarrow \text{workqueue}.\text{tryPop}(x) \text{ do}
\]
\[
\text{insert seeds into thread-local PQs}
\]
\[
\text{while not done do}
\]
\[
\text{find best move in PQs}
\]
\[
\text{perform best move}
\]
\[
\text{claim and insert/update neighbors into PQs}
\]
\[
\text{rollback to best solution of thread-local search}
\]

Challenges

- move order?
- balance constraint
- incorrect gains

\[\text{← as applied to shared partition. use atomics}\]

\[\text{← parallel prefix sum}\]
Parallel Fiduccia-Mattheyses

Algorithm 3: Parallel Localized FM

while improvement found do
do on each thread
  while seeds ← workqueue.tryPop(x) do
    insert seeds into thread-local PQs
  while not done do
    find best move in PQs
    perform best move
    claim and insert/update neighbors into PQs
  rollback to best solution of thread-local search

roll back to best solution

Challenges

- move order?
- balance constraint
- incorrect gains

← as applied to shared partition. use atomics
← parallel prefix sum
← recalculate in parallel. next slide
Recalculate Gains

\[ \text{gain}(u \rightarrow t) = \sum_{e \in \Gamma(u) : e \cap \Pi(u) = \{u\}} \omega(e) - \sum_{e \in \Gamma(u) : e \cap t = \emptyset} \omega(e) \]

current part of \( u \)   target part
Recalculate Gains

Algorithm 4: Gain Calculation Sketch

Let $u$ be moved from $s$ to $t$
gain ← 0

for $e \in I(u)$ do
  if $u$ is the last pin of $e$ still left in $s$ then
    gain += $\omega(e)$
  if $u$ is the first pin of $e$ to move into $t$ then
    gain -= $\omega(e)$
Recalculate Gains

**Algorithm 4: Gain Calculation Sketch**

Let $u$ be moved from $s$ to $t$

$\text{gain} \leftarrow 0$

for $e \in I(u)$ do

- **if** $u$ is the last pin of $e$ still left in $s$ **then**
  - $\text{gain} += \omega(e)$

- **if** $u$ is the first pin of $e$ to move into $t$ **then**
  - $\text{gain} -= \omega(e)$

were all pins of $e \cap s$ from beginning of the pass removed?

is $u$ the last one?

was any pin of $e$ moved into $s$ before $u$ leaves?
Recalculate Gains

**Algorithm 4: Gain Calculation Sketch**

Let $u$ be moved from $s$ to $t$

gain $\leftarrow 0$

for $e \in I(u)$ do

- **if** $u$ is the last pin of $e$ still left in $s$ **then**
  
gain $+$= $\omega(e)$

- **if** $u$ is the first pin of $e$ to move into $t$ **then**
  
gain $-$= $\omega(e)$

- **were all pins of** $e \cap s$ **from beginning of the pass removed?**
- **is** $u$ **the last one?**
- **was any pin of** $e$ **moved into** $s$ **before** $u$ **leaves?**

- **were all pins of** $e \cap t$ **from beginning of the pass removed?**
- **before** $u$ **joined?**
- **is** $u$ **the first pin of** $e$ **moved into** $t$ **during this pass?**
Recalculate Gains

Algorithm 4: Gain Calculation Sketch

Let $u$ be moved from $s$ to $t$

gain $\leftarrow 0$

for $e \in I(u)$ do

\begin{itemize}
  \item \textbf{if} $u$ is the last pin of $e$ still left in $s$ \textbf{then} \\
      gain $\leftarrow$ gain $+\omega(e)$
  \item \textbf{if} $u$ is the first pin of $e$ to move into $t$ \textbf{then} \\
      gain $\leftarrow$ gain $-\omega(e)$
\end{itemize}

\begin{itemize}
  \item were all pins of $e \cap s$ from beginning of the pass removed? \\
  \item is $u$ the last one? \\
  \item was any pin of $e$ moved into $s$ before $u$ leaves?
\end{itemize}

\begin{itemize}
  \item were all pins of $e \cap t$ from beginning of the pass removed? \\
  \item before $u$ joined? \\
  \item is $u$ the first pin of $e$ moved into $t$ during this pass? \\
  \item before/after = lower/higher position in move sequence
\end{itemize}
Recalculate Gains

Algorithm 5: Compute Auxiliary Information

\[
\text{rem-pins}[e][i] \leftarrow |e \cap V_i| : \forall e \in E, i \in \{1, \ldots, k\}
\]

\textbf{for each} move index \( i \in \langle 1, \ldots, r \rangle \) \textbf{in parallel}

\hspace{1em} Let \( m_i \) move \( u \) from \( s \) to \( t \)

\hspace{2em} \textbf{for} \( e \in \text{l}(u) \) \textbf{do}

\hspace{3em} first-in\[e]\[t\] \leftarrow \min(i, \text{first-in}[e][t]) \quad // \text{compare-and-swap loop}

\hspace{3em} last-out\[e][s] \leftarrow \max(i, \text{last-out}[e][s]) \quad // \text{compare-and-swap loop}

\hspace{3em} \text{rem-pins}[e][s] \leftarrow \text{rem-pins}[e][s] - 1 \quad // \text{atomic fetch-and-add}
Recalculate Gains

**Algorithm 5: Compute Auxiliary Information**

| rem-pins[e][i] ← |e ∩ Vi| : ∀e ∈ E, i ∈ {1, . . ., k} |
|-------------------|
| for each move index i ∈ {1, . . ., r} in parallel |
| Let m_i move u from s to t |
| for e ∈ l(u) do |
| first-in[e][t] ← min(i, first-in[e][t]) // compare-and-swap loop |
| last-out[e][s] ← max(i, last-out[e][s]) // compare-and-swap loop |
| rem-pins[e][s] ← rem-pins[e][s] − 1 // atomic fetch-and-add |

\[ O(\sum_{i=1}^{r} |l(u_j)|) \Rightarrow \text{linear time if no contention} \]
Recalculate Gains

Algorithm 5: Compute Auxiliary Information

\[
\text{rem-pins}[e][i] \leftarrow |e \cap V_i| : \forall e \in E, i \in \{1, \ldots, k\}
\]

for each move index \( i \in \langle 1, \ldots, r \rangle \) in parallel

Let \( m_i \) move \( u \) from \( s \) to \( t \)

for \( e \in I(u) \) do

1. first-in[e][t] \( \leftarrow \) min\((i, \text{first-in}[e][t])\) \hfill // compare-and-swap loop
2. last-out[e][s] \( \leftarrow \) max\((i, \text{last-out}[e][s])\) \hfill // compare-and-swap loop
3. rem-pins[e][s] \( \leftarrow \) rem-pins[e][s] \(-\) 1 \hfill // atomic fetch-and-add

- \( O(\sum_{1}^{r} |I(u_j)|) \Rightarrow \) linear time if no contention
- \( k \cdot |E| \) memory is a lot \( \Rightarrow \) use only for large edges
- iterate directly over pins for small edges \( \Rightarrow O(|e| \cdot |e \cap \text{moved vertices}|) \)
- common inputs have many small and a few large edges
Recalculate Gains

Algorithm 5: Compute Auxiliary Information

\[ \text{rem-pins}[e][i] \leftarrow |e \cap V_i| : \forall e \in E, i \in \{1, \ldots, k\} \]

for each move index \( i \in \langle 1, \ldots, r \rangle \) in parallel

Let \( m_i \) move \( u \) from \( s \) to \( t \)

for \( e \in I(u) \) do

\[ \begin{align*}
\text{first-in}[e][t] &\leftarrow \min(i, \text{first-in}[e][t]) & \text{\textit{\textendash\ compare-and-swap loop}} \\
\text{last-out}[e][s] &\leftarrow \max(i, \text{last-out}[e][s]) & \text{\textit{\textendash\ compare-and-swap loop}} \\
\text{rem-pins}[e][s] &\leftarrow \text{rem-pins}[e][s] - 1 & \text{\textit{\textendash\ atomic fetch-and-add}} 
\end{align*} \]

- \( O(\sum_1^r |I(u_j)|) \Rightarrow \text{linear time if no contention} \)
- \( k \cdot |E| \) memory is a lot \( \Rightarrow \) use only for \textit{large} edges
- iterate directly over pins for \textit{small} edges \( \Rightarrow \ O(|e| \cdot |e \cap \text{moved vertices}|) \)
- common inputs have many small and a few large edges
- new variant: no CAS loops, no extra memory, simple
Experiments on Big Instances

- for comparison with fast partitioners: Zoltan, PaToH-D
- for scaling experiments

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

- $k \in \{2, 8, 16, 64\}$ with imbalance: $\varepsilon = 3\%$

- 5 random seeds
- 1,4,16,64 threads
Self-Relative Speedups

![Graphs showing self-relative speedups for different components and algorithms.](image-url)
Quality versus Fast Partitioners

fraction of instances for which $\tau \times \text{algo} \leq \text{best}$

- getting high early is good
- value at $\tau = 1$ gives percentage of best solutions
Relative Slowdowns

- higher means that algorithm is slower
- Mt-KaHyPar with 4 threads slightly faster than PaToH-D
Experiments on Smaller Instances

- for comparison with sequential partitioners: KaHyPar, hMetis, PaToH-Q
- Intel Xeon Gold, 2 sockets, 20 cores @ 2.1 Ghz, 96 GB RAM
- # Hypergraphs: [publicly available]
  - SuiteSparse Matrix Collection | 184
  - SAT Competition 2014 (3 representations) | 92.3
  - ISPD98 & DAC2012 VLSI Circuits | 28
- $k \in \{2, 4, 8, 16, 32, 64, 128\}$ with imbalance: $\varepsilon = 3\%$
- 10 random seeds
- 1, 10, 20 threads
Smaller Instances

<table>
<thead>
<tr>
<th>Fraction of instances</th>
<th>$\tau$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>0.80</td>
<td>1.10</td>
</tr>
<tr>
<td>0.60</td>
<td>1.15</td>
</tr>
<tr>
<td>0.40</td>
<td>1.20</td>
</tr>
<tr>
<td>0.20</td>
<td>1.25</td>
</tr>
<tr>
<td>0.10</td>
<td>1.30</td>
</tr>
</tbody>
</table>

- Mt-KaHyPar 10
- hMetis-R
- KaHyPar-HFC
- PaToH-D
- KaHyPar-CA
- PaToH-Q
Smaller Instances

![Graph showing the fraction of instances vs \( \tau \)]

- Red line: Mt-KaHyPar 10
- Orange line: PaToH-Q
- Purple line: PaToH-D

**Axes:**
- Y-axis: Fraction of instances
- X-axis: \( \tau \)
Smaller Instances

![Graph showing the performance of different algorithms across various instances. The x-axis represents the number of instances, ranging from 0 to 3416. The y-axis represents the relative time to Mt-KaHyPar 10, ranging from $10^{-1}$ to $10^3$. The graph includes lines for hMetis-R, KaHyPar-HFC, PaToH-Q, KaHyPar-CA, PaToH-D, and PaToH-S, each with a distinct color and marker.](image-url)
Conclusion

Mt-KaHyPar

- thorough engineering and parallelization of existing techniques
- better quality than comparable multilevel partitioners
- great scalability
- solution quality independent of parallelism
- first fully parallel FM due to parallel gain recalculation
- first shared-memory hypergraph partitioner

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

Mt-KaHyPar

- thorough engineering and parallelization of existing techniques
- better quality than comparable multilevel partitioners
- great scalability
- solution quality independent of parallelism
- first fully parallel FM due to parallel gain recalculation
- first shared-memory hypergraph partitioner

![GitHub Repository](https://github.com/kahypar/mt-kahypar)

Future Work

- n-level coarsening and uncoarsening (coming soon)
- flow-based refinement (in the works)
- improve coarsening for skewed inputs and corner cases