FastPart: A Powerful Portfolio-based Hypergraph Partitioner

Amur Ghose
CSE Department, UC San Diego
aghose@ucsd.edu

Andrew B. Kahng
CSE and ECE Departments, UC San Diego
abk@ucsd.edu

Abstract

VLSI physical design relies on high-quality hypergraph partitioning. Multilevel partitioners often lose quality at small K and under tight balance, and no single top-level flow consistently yields the strongest results across all K. We present *K-adaptive Ensembled Partitioning (KEP)*, which combines recursive bisection and direct K-way search and replaces hard-feasibility *Fiduccia–Mattheyses (FM)* with an augmented-Lagrangian refinement step *(AL-FM)* that can cross short infeasible regions before repairing balance. We then build FastPart, a fixed-budget portfolio controller that coordinates KEP with complementary solver flows and returns the best feasible partition it finds. On Titan23 and IBM cases under two-sided balance constraints, FastPart outperforms most public references, has no loss cases in the recent-method comparisons we could evaluate and converges to strong solutions substantially faster than TritonPart, with median speedups up to 8.6×.

CCS Concepts

• **Hardware → Partitioning**; • **Mathematics of computing →** *Hypergraphs.*

Keywords

hypergraph partitioning, physical design, multilevel methods, balance constraints, ensemble methods

ACM Reference Format:
Amur Ghose and Andrew B. Kahng. 2026. FastPart: A Powerful Portfolio-based Hypergraph Partitioner. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD '26), November 08–12, 2026, San Jose, CA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3831252.3834238

1 Introduction

Hypergraph partitioning (HGP) is a core primitive in VLSI physical design, shaping hierarchy construction, floorplanning, congestion and timing closure [4, 6]. Multilevel partitioning remains the dominant approach because it scales to large netlists while exploiting locality [22]. However, modern physical-design flows stress HGP solvers along three axes at once: low-K quality for hierarchy construction, stable quality across multiple partition counts K and strict multi-constraint balance. A single fixed multilevel pipeline rarely handles all three well, leading to poor cutsizes and/or runtimes.

Two coupled weaknesses motivate our design. The first weakness is local-search brittleness under tight balance. KL/FM-style refinement and its K-way generalizations enforce feasibility through hard balance checks, so they reject moves that temporarily violate balance even when those moves would unlock significantly better cuts [11, 17, 30]. A partition can therefore look locally optimal with respect to single-vertex feasible moves even though a short excursion through mild imbalance, or a small multi-vertex exchange, would expose a substantially better cut. Flow-based escapes help in some cases, but they still usually inherit a hard-feasibility approach during move selection [20, 36]. The second weakness is that no single top-level flow works well across all K values. *Recursive bisection (RB)* is often strong for K=2, 3, while direct K-way and spectrally seeded runs can preserve structure that recursive splitting may damage at larger K [5, 18]. In practice, this is also a runtime problem: the solver must decide how to spend one fixed budget across complementary search biases without turning the run into an uncontrolled set of restarts.

Our paper solves the HGP problem on two levels. KEP is the main solver in this paper. For $K \in \{2, 3\}$, it uses a deep multilevel pipeline with Augmented-Lagrangian FM (AL-FM) refinement. For $K \geq 4$, it runs both a recursive-bisection path and a direct spectrally seeded K-way path, then tries a local merge with *Consensus-and-Clean (C&C).* **FastPart** is a controller built around KEP. It divides a given fixed time budget across KEP and other complementary solver flows, then returns the best feasible partition it finds. Throughout the paper, *feasible* means that the partition passes a two-sided balance check used in the experiments. Figure 1 shows the full flow.

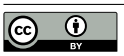
Our contributions.

- *KEP solver.* A K-adaptive solver whose small-K path combines deep multilevel search with AL-FM refinement and a larger-K path merging recursive and direct candidates via localized C&C.
- *FastPart controller.* A fixed-time controller that coordinates KEP with complementary solver flows, keeps a pool of feasible candidates plus a small repair queue and improves coverage through short refinement and local recombination.
- *Evaluation versus public references and recent work.* Results on Titan23 and IBM under tight balance, including comparison to the current HypergraphPartitioning leaderboard [24], a rerun against SHyPar [34], comparisons to HyperEF 2.0 and EasyPart in their published settings [35, 41] and backend/runtime results for both CPLEX and OR-Tools.

The rest of the paper reviews the closest prior work in Section 2, then describes FastPart, KEP, AL-FM and C&C in Section 3. Finally, we present our experimental protocol, main results, recent-method comparisons and the tight-balance stress test in Section 4.

2 Related Work

This section reviews four topics most relevant to this paper: multilevel HGP baselines, balance handling during refinement, the tradeoff between recursive bisection and direct K-way partitioning, and earlier work on combining multiple candidate partitions. Multilevel hypergraph partitioning remains the dominant VLSI HGP



ICCAD '26, San Jose, CA, USA

ACM ISBN 979-8-4007-2873-0/2026/11
https://doi.org/10.1145/3831252.3834238

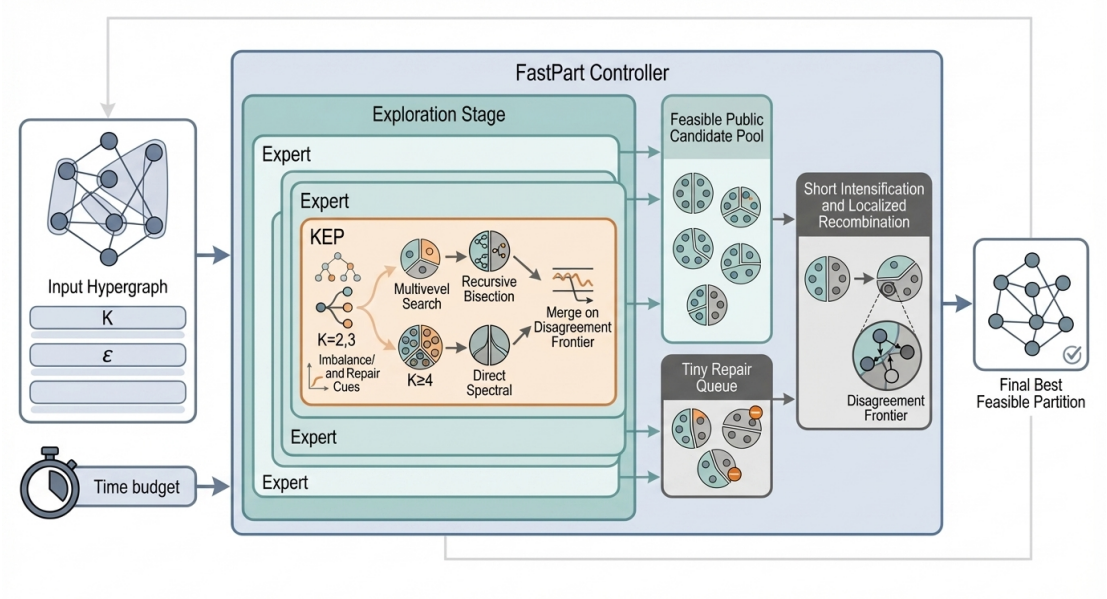


Figure 1: Overview of FastPart and KEP. FastPart receives an input hypergraph and time budget, runs several solver flows and highlights KEP as the main solver. Inside KEP, the small-K path uses deep multilevel AL-FM, while the larger-K path combines recursive and direct flows. A local merge then repairs a small disagreement region and returns the final feasible partition.

framework, with strong baselines including hMETIS, PaToH and TritonPart [8, 12, 25–28]. The long-standing tradeoff between Recursive Bisection (RB) and direct K-way partitioning is well documented [5, 28, 39]: RB often gives clean small-K separators, while direct K-way methods preserve global structure that recursion may lose. There is no clearly dominating method for all problems.

Constraints, determinism and scalability. VLSI HGP flows routinely require multi-dimensional balance and other side constraints, not just unconstrained cut minimization. TritonPart explicitly supports these constraints and integrates *integer linear programming (ILP)* solvers at critical stages [8]; in parallel HGP, Mt-KaHyPar likewise emphasizes scalability together with deterministic operating modes [19]. Our target is not best cuts at any cost, but rather strong cuts under hard balance constraints with reproducible control flow and a fixed time budget.

FastPart combines KEP with a tuned Mt-KaHyPar portfolio. The artifact's `shipped-src` branch bundles a default KEP-disabled controller over pinned, unpatched Mt-KaHyPar [2]. At 300 s on 112 vCPUs, this mode matches or improves 52/66 FastPart cuts and is 0.27% higher in geometric mean (three-seed median), underscoring Mt-KaHyPar's underreported strength. The `main` branch provides 66 CORD partitions, 65 meeting or improving the KEP/FastPart target; this GPU-accelerated variant targets large hypergraphs within and beyond VLSI.

Balance handling and local improvement. Classical KL/FM refinement uses hard feasibility checks, while later methods add pairwise moves, direct K-way refinement or flow-based escapes [15, 17, 20, 29, 36]. Related heuristics have also explored temporary balance violations in other partitioning settings [16]. Augmented-Lagrangian (AL) methods provide a principled way to relax hard constraints through dual prices and quadratic penalties [23, 31, 33]. Our AL-FM step brings this idea directly into FM-style hypergraph refinement for tight-balance VLSI partitioning.

Spectral methods, recent systems and multi-run controllers. Spectral methods and learned spectral refiners provide complementary structure-sensitive signals, including SpecPart and K-SpecPart [9, 10, 14, 21, 32]. Multilevel spectral formulations that explicitly account for vertex sizes have also been developed for HGP [43]. More recent VLSI-oriented partitioners such as EasyPart, BLASPart, HyperEF 2.0 and SHyPar continue the trend of pairing multilevel search with stronger initialization, coarsening or refinement mechanisms [34, 35, 41, 42]. At the systems level, some methods run several solver flows under one budget instead of relying on one flow alone. FastPart follows that idea, but uses a fixed template menu and a border-only merge step instead of broad stochastic restart or global evolutionary search.

Combining candidate partitions. Several earlier methods also combine multiple candidate partitions. Their shared idea is simple: keep the regions where the candidates already agree, and focus extra work on the smaller set of vertices where they disagree [13, 37, 40]. Our sub-method, C&C, follows this idea, but aligns labels and solves a small ILP only on the disagreement border.

What is different here. Earlier multi-start, evolutionary and recombination schemes already use complementary seeds and search biases [18, 19, 36, 38]. However, as of yet it is unclear how to combine these mechanisms to produce a solver that can efficiently search over multiple candidate methods and partitions and produce partitions of high quality (i.e., low cutsize) in a much smaller timeframe than competing solvers. This is the gap that we fill with FastPart and KEP as a pair.

3 FastPart and KEP

In this section, we first define the constrained HGP problem, then describe FastPart and KEP and finally explain the two main KEP components: AL-FM for small K and C&C for larger K.

Problem formulation. We address the standard constrained hypergraph partitioning problem [8]. Given a hypergraph $H(V, E)$, vertices $v \in V$ with m-dimensional weights w_v and hyperedges $e \in E$ with weights w_e, the goal is to find a K-way partition $S = \{V_1, \ldots, V_K\}$ that minimizes total cut

$$\Phi_{\text{cut}}(S) = \sum_{e \in E} w_e \cdot \text{Cut}(e), \tag{1}$$

with Cut(e)=1 if e spans multiple blocks. Let $\omega_i(j)$ denote the weight of block i in dimension j, and let W_j denote the total weight in that dimension. We require two-sided balance:

$$\left(\tfrac{1}{K} - \epsilon\right) W_j \le \omega_i(j) \le \left(\tfrac{1}{K} + \epsilon\right) W_j, \tag{2}$$

for all blocks i and dimensions j, where ϵ is the allowed imbalance factor (e.g., 2%). When only the right-hand side is respected, this is termed upper-bounded balance, a weaker ("UBfactor") constraint.

3.1 FastPart: Fixed-Budget Controller

FastPart is the outer execution policy. For a fixed quartet of Hypergraph, Partition count, balance factor, and time budget, i.e. $\equiv (H, K, \epsilon, T)$ and CPU budget, it spends one wall-clock budget across a predefined library of run templates and returns the lowest-cut partition in its *strictly feasible* pool. It uses three kinds of runs:

(1) **KEP.** The strongest expert: deep small-K multilevel search and larger-K recursive/direct fusion.
(2) **Global probes.** Cheaper direct or recursive runs, via spectral and bootstrap methods, that expose alternative large-scale structure quickly.
(3) **Follow-on runs.** Short warm-start or repair runs from one incumbent solution.

Core intuition. Previous studies in HGP-partitioning reveal that we may efficiently model the process of partitioning as having two phases - a **global** phase where the coarse partitioning is obtained, and a **local** phase where the partition is refined for a final solution, with the local refinement called **V-cycle iteration.** FastPart uses many parallel probes to explore the **global** space extremely quickly (feasible within 1 minute even on the largest VLSI hypergraphs via spectral methods), and narrows down the list of candidates to a set which is likely to contain the final best partition after a stronger but slower method like **KEP** is applied atop these candidates.

Controller structure. FastPart has three stages: *exploration* launches independent template instances until threads are full; *intensification* warm-starts short direct refinements from selected incumbents; and *recombination* solves only the small frontier where strong parents disagree. KEP supports all three stages. The primary driver is spectral cuts for exploration and KEP for refinement, with optional modifiers such as extra V-cycles, tighter balance or rebalancer presets. Extra candidates are primarily RB-based; $K \ge 3$ integrates an additional balance check for lopsided intermediate partitions. Selection for each stage is modulated by cut quality and prior attempts to repair the cut. The public pool $\mathcal{P}$ contains only strictly feasible partitions and is the only source of reported results; the repair queue $\mathcal{R}$ keeps only a few low-cut relaxed partitions for possible repair. For $K \ge 3$, the opening search window is biased toward balanced candidates of low cutsize. Once any strictly feasible incumbent exists, new launches use only the target ϵ (not relaxed tolerances), and underperforming templates are dropped.

Downselection, recombination and reproducibility. When enough candidates accumulate, FastPart keeps the current best candidates from each base method (e.g., KEP), and then fills remaining refinement slots by cutsize. (Balance-violating candidates serve only as repair fallbacks.) These surviving candidates receive warm-start direct refinement waves: a one-thread repair with one V-cycle and, if time remains, consequent two/four-thread repairs. Recombination uses only a few parents: the global best plus V-cycle refined RB and spectral cuts. After recombination, the resulting partition may receive a short cleanup refinement via V-cycles for either cutsize or balance. Each reported run is with a fixed seed. The controller can terminate by wall time, completed work, target cut or stagnation; the target-cut rule supersedes its time budget.

3.2 KEP: K-Adaptive Solver

KEP is the main solver inside FastPart. Its top-level routing rule is intentionally simple: for $K \in \{2, 3\}$ it runs one deeper multilevel search tuned for low cut under tight balance, while for $K \ge 4$ it builds two complementary candidate partitions and then reconciles them. Figure 2 shows this split. The key point is that KEP does *not* change this controller online: the path is fixed by K, which makes the execution policy easy to understand and reproduce.

Small-K path. For bisection and trisection, partition quality is driven mainly by how much work is done inside one multilevel hierarchy. KEP therefore spends its budget on a *deep* small-K pipeline rather than on many shallow alternatives. Each start executes repeated V-cycles. In one V-cycle, KEP coarsens the hypergraph, constructs an initial partition on the coarsest level, uncoarsens back

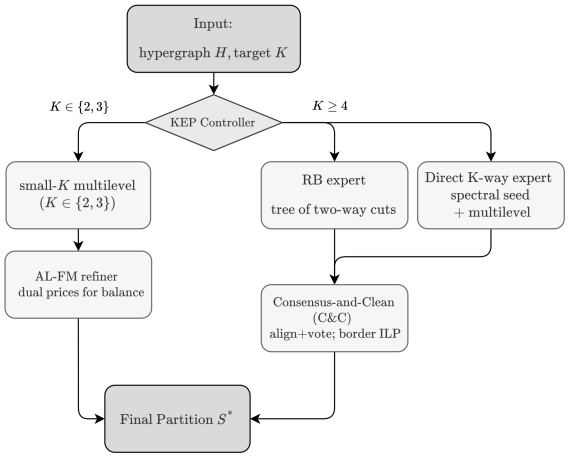


Figure 2: KEP workflow. For $K \in \{2, 3\}$, KEP uses deep multilevel search with AL-FM refinement. For $K \ge 4$, it runs recursive-bisection and direct spectrally seeded flows, then applies C&C on the disagreement frontier.

toward the original instance, and applies AL-FM refinement during uncoarsening (Section 3.3). At the end of the pass it applies the standard V-cycle balance repair [38] and updates the incumbent. The final output is the lowest-cut partition that is feasible under the target two-sided balance check. In short, the small-K path is a deep multilevel solver that is permissive during local search but strict about the partition it finally reports.

Larger-K path. For $K \geq 4$, KEP does not rely on one global search bias. Instead, it constructs two candidate partitions and lets a final local merge decide where each one is more trustworthy.

(1) **Recursive-bisection candidate S_{RB}.** This arm recursively decomposes the instance into balanced subproblems and invokes the small-K machinery at each split. It tends to preserve clean separators and hierarchy-consistent structure.
(2) **Direct K-way candidate S_{dir}.** This arm runs a direct multilevel solver. Its coarsest-level seed is spectral: a low-dimensional embedding is clustered into K blocks and then refined by the multilevel solver. This path can preserve global cross-block structure that recursive splitting may damage.

The two candidates complement each other: S_{RB} is usually stronger on separator hierarchy, while S_{dir} can preserve large-scale structure that a sequence of recursive splits may never recover. KEP therefore does not simply return whichever candidate has lower cut. Instead, it aligns their block labels, keeps the regions where they already agree, and invokes localized Consensus-and-Clean only on the relatively small frontier where they disagree. The larger-K path is thus best read as a three-step pipeline: build S_{RB}, build S_{dir}, then spend the remaining optimization effort only on the disputed region.

3.3 Augmented-Lagrangian Refinement (AL-FM)

Standard Fiduccia–Mattheyses (FM) refinement treats Equation (2) as a hard filter: if a move would make the current partition infeasible, FM rejects it immediately, even when that move would reduce the cut enough to unlock a better solution a few steps later. This is especially limiting when ϵ is small. AL-FM keeps the same multilevel scaffold as ordinary FM, but changes how moves are scored during uncoarsening. Instead of forbidding all temporary imbalance, it assigns an explicit penalty to imbalance and lets cut reduction trade off against that penalty [31].

Setup and notation. Let the current partition be $S = (V_1, \ldots, V_K)$, where V_i is the current vertex set of block i. For each weight dimension j, let W_j be the total instance weight in that dimension and let $\omega_i(j)$ be the current weight of block i in dimension j. The feasible interval $[L_j, U_j]$ is delimited by

$$L_j = (1/K - \epsilon)W_j, \qquad U_j = (1/K + \epsilon)W_j.$$

We measure balance violation by

$$r_i(j) = \max\big(0,\ \omega_i(j) - U_j,\ L_j - \omega_i(j)\big). \tag{3}$$

Thus $r_i(j) = 0$ when block i is feasible in dimension j, and $r_i(j) > 0$ only when the block lies outside the interval $[L_j, U_j]$. Let r_i denote the full violation vector of block i across all dimensions.

AL objective. AL-FM augments the cut objective with a linear dual-price term and a quadratic penalty:

$$\mathcal{L}(S, \lambda, \rho) = \Phi_{\mathrm{cut}}(S) + \sum_{i=1}^{K} \lambda_i^\top r_i + \frac{\rho}{2} \sum_{i=1}^{K} \|r_i\|_2^2, \tag{4}$$

where λ_i is a nonnegative dual-price vector for block i and $\rho > 0$ is the penalty scale. The linear term remembers which blocks have been persistently hard to balance, while the quadratic term makes large violations increasingly expensive.

Move evaluation. Consider moving one vertex u from a source block a to a destination block b. Standard FM would score that move only by the cut change $\Delta\Phi_{\mathrm{cut}}$. AL-FM instead scores the total change in the augmented objective:

$$\Delta\mathcal{L} = \Delta\Phi_{\mathrm{cut}} + \Delta\left(\sum_{i=1}^{K} \lambda_i^\top r_i\right) + \frac{\rho}{2}\,\Delta\left(\sum_{i=1}^{K} \|r_i\|_2^2\right). \tag{5}$$

Only the two incident blocks, a and b, change under a single-vertex move, so the additional bookkeeping remains local. A move that worsens balance is therefore still admissible, but only if its cut improvement is large enough to offset the extra AL penalty. This lets local search cross short infeasible valleys without letting the partition drift arbitrarily far from feasibility.

Dual update and penalty schedule. After each refinement pass, AL-FM updates the dual prices using the current violation vectors:

$$\lambda_i \leftarrow \max\big(0,\ \lambda_i + \rho\, r_i\big). \tag{6}$$

This increases the price of blocks that remain hard to repair. The penalty parameter ρ is also adjusted online: if violations remain large, or if repeated passes fail to improve quality while staying near-feasible, then ρ is increased; if several passes finish feasibly, then ρ is allowed to decay so the search can explore more freely. The dual variables and ρ therefore play different roles: λ_i records *where* the search keeps breaking balance, while ρ controls *how strongly* those violations are penalized overall.

Where AL-FM sits in the multilevel loop. AL-FM does not replace the surrounding multilevel solver. Each V-cycle still follows the usual coarsen-initialize-uncoarsen pattern. The change is local to uncoarsening: FM-style moves are ranked by $\Delta\mathcal{L}$ instead of by cut alone, then a repair step brings the candidate back toward strict feasibility before the next pass. Within a multi-start run, each candidate keeps its own AL state (λ, ρ), so there is no cross-talk across restarts and the overall schedule remains deterministic under a fixed execution policy.

3.4 Consensus-and-Clean (C&C)

C&C (Algorithm 1) is a deterministic method to reconcile a small set of full candidate partitions $\mathcal{C}=\{S_1, \ldots, S_N\}$, such as the recursive and direct larger-K outputs, by focusing optimization effort only on regions where those candidates disagree.

1. Candidate Alignment. We select the lowest-cut candidate as the baseline, S_{base}. All other candidates are aligned to S_{base} by finding the block-label permutation that maximizes agreement; with the seed fixed, this alignment and the resulting ILP are reproducible.

2. Border Identification via Prioritization. We identify vertices where the aligned candidates disagree with the baseline. A priority

Algorithm 1 Consensus-and-Clean (C&C)

procedure ConsensusAndClean(C=$\{S_1, \ldots, S_N\}, H, \epsilon, B_{\max}$)
  $S_{\text{base}} \leftarrow \arg\min_{X \in C} \Phi_{\text{cut}}(X)$
  $V_i^{\text{base}} \leftarrow \{v \in V : S_{\text{base}}(v) = i\}$ for each block i
// **1. Alignment**
  $C' \leftarrow \{S_{\text{base}}\} \cup \{\text{AlignLabels}(X, S_{\text{base}}) \mid X \in C \setminus \{S_{\text{base}}\}\}$
// **2. Border Identification via Prioritization**
  for $v \in V$ **do**
    $\text{Votes}(v, j) \leftarrow \sum_{Y \in C'} \mathbb{I}(Y(v)=j)$
    $b \leftarrow S_{\text{base}}(v)$; $c \leftarrow \arg\max_{j \neq b} \text{Votes}(v, j)$
    if $\text{Votes}(v, c) \geq \text{Votes}(v, b)$ **then**
      $P[v] \leftarrow \text{Votes}(v, c) - \text{Votes}(v, b)$
    end if
  end for
  Propagate $P[\cdot]$ to 1-hop neighbors
  $V_{\text{border}} \leftarrow \text{Topmost}(\text{SortByPriority}(P),\ B_{\max})$
// **3. Localized ILP Formulation**
  $V_{\text{ILP}} \leftarrow V_{\text{border}}$
  for $i = 1$ to K **do** ▷ Create pseudo-vertices
    $w(p_i) \leftarrow \sum_{v \in (V_i^{\text{base}} \setminus V_{\text{border}})} w_v$; fix p_i to block i
    $V_{\text{ILP}} \leftarrow V_{\text{ILP}} \cup \{p_i\}$
  end for
  Build induced hypergraph $H_{\text{ILP}}(V_{\text{ILP}}, E_{\text{ILP}})$
// **4. Optimization and Patching**
  Minimize $\Phi_{\text{cut}}(H_{\text{ILP}})$ s.t. global balance (Equation (2))
  $S_{\text{ILP}} \leftarrow \text{SolveILP}(H_{\text{ILP}}, \text{time\_limit})$
  if solved **then**
    $S_{\text{C\&C}} \leftarrow \text{PatchSolution}(S_{\text{base}}, S_{\text{ILP}})$
    if $\Phi_{\text{cut}}(S_{\text{C\&C}}) < \Phi_{\text{cut}}(S_{\text{base}})$ **then**
      return $S_{\text{C\&C}}$
    end if
  end if
  return S_{base}
end procedure

score $P[v]$ is assigned from the vote counts in Algorithm 1: vertices where a competing block label is as strong as, or stronger than, the baseline label receive higher priority. We propagate this priority to one-hop neighbors so the selected region remains connected. A fixed-capacity window V_{border} with cap $B_{\max}$ is then selected; in practice, we set $B_{\max}$ by a small absolute size and by a small fraction of the instance size.

3. Localized ILP with Pseudo-Vertices. The core idea in C&C is to formulate a reduced ILP that operates only on V_{border} while still enforcing *global* balance. Let $V_i^{\text{base}} = \{v \in V : S_{\text{base}}(v) = i\}$ denote the baseline block-i vertices. We achieve the reduced ILP using *pseudo-vertices*.

- **Pseudo-vertices for global balance.** For each block i, we create a pseudo-vertex p_i. Its weight aggregates all vertices *outside* the border that are currently assigned to block i in S_{base}: $w(p_i) = \sum_{v \in V_i^{\text{base}} \setminus V_{\text{border}}} w_v$. We then fix p_i to block i.
- **Induced hypergraph H_{ILP}.** We construct H_{ILP} over $V_{\text{border}} \cup \{p_1, \ldots, p_K\}$. Hyperedges incident to V_{border} are included and rewritten so that their exterior adjacencies (connections) are to the appropriate pseudo-vertices.

4. Optimization and Patching. We minimize Φ_{cut} on H_{ILP} subject to the global balance constraints in Equation (2), which now incorporate the fixed weights $w(p_i)$. The ILP is solved under a strict local time budget. If the resulting solution $S_{\text{C\&C}}$ strictly improves the cut while satisfying all constraints, it is adopted; otherwise, KEP returns S_{base}. On the Titan23 testcases, this border averages only 0.032% of vertices, and C&C accepts an improvement in 96% of runs (fallback: 4%).

4 Experimental Setup and Results

In the following, Section 4.1 explains the benchmarks and reporting rules. Section 4.2 gives our main ϵ=2% results. Section 4.3 separates the contribution of KEP from the added benefit of the FastPart controller. Section 4.4 compares against recent methods, and Section 4.5 ends with a tighter-balance ϵ=0.1% stress test.

4.1 Benchmarks and Reporting Protocol

We evaluate on Titan23 (22 instances) and the ISPD98/IBM suite (18 instances) [7]. Unless stated otherwise, all reported results enforce two-sided balance with $\epsilon = 2\%$ and optimize the minimum hyperedge-cut objective Φ_{cut}. All runs use the same two-sided evaluator on a 112-thread / 220 GB AMD EPYC machine; FastPart, KEP and TritonPart [8] reruns use the full 112-thread resource. All paper collaterals are available in the artifact repository [2]. In the result tables, boldface identifies the best value in each comparison group.

Public targets. Leaderboard targets come from HypergraphPartitioning (commit `71636c3`, accessed April 10, 2026) [24]. Because those public rows use UBfactor-style balance constraints while our evaluation uses two-sided balance, we treat them as reference cuts rather than as exact-feasibility reruns and report our gap to them after evaluating our solutions with the stricter two-sided checker.

How we report FastPart and KEP. FastPart is reported from a single fixed-budget 180s run per instance (seed 0). The deeper KEP sweep groups 1000 independently-seeded runs into 50 sets of 20 and reports the rounded average best-of-20 cut. KEP and FastPart use the C++ OpenROAD/TritonPart lineage, with CPLEX primary and OR-Tools alternative; the TritonPart baseline is public src/`par` at commit `5e8f15c075` [1, 24]. We rerun SHyPar [34] and use published settings for HyperEF 2.0 [35] and EasyPart [41]. Thus, FastPart and KEP are not budget-matched. The artifact also includes best single-run KEP cuts of 428 on `openCV` at K=2 and 1832 on `bitcoin_miner` at K=4; the reported KEP rows characterize the expert itself.

4.2 Main results at two-sided $\epsilon = 2\%$

This subsection answers the basic question: how well do FastPart and KEP perform in the well-studied ϵ=2% setting? Under a stricter two-sided balance check than the public UBfactor-style references [24], FastPart matches or improves on 18/22, 20/22, and 22/22 Titan23 reference targets for K=2, 3, 4, respectively (Table 1), and improves all 18/18 IBM K=2 targets (Table 5), with average cut gaps of −0.5, −66.0, −246.1 and −45.4. Standalone KEP is also competitive, matching or improving on 16/22, 20/22, and 18/22 Titan23 reference targets for K=2, 3, 4, with average gaps of −0.4, −111.9 and −213.3 (Table 1). The largest gains remain concentrated on instances such as `gsm_switch` and `bitcoin_miner`, while the main

ID	Bench	VE	T_{234}	FP_{234}	KEP_{234}	hM_{234}	TP_{234}	SP_2
01	sparcT1_core	91976/92827	978/1889/2517	**974/1762/2174**	976/1776/2457	982/2188/2533	983/1787/2517	977
02	neuron	92290/125305	**243**/396/431	**243**/363/**405**	**243/362/405**	245/372/432	**243**/371/**405**	244
03	stereo_vision	94050/127085	181/331/407	**169/320/372**	171/322/409	171/333/440	176/325/413	**169**
04	des90	111221/139557	**371**/535/747	**371/504/656**	**371/504**/668	377/536/696	373/524/668	374
05	SLAM_spheric	113115/142408	**1061**/2720/3241	**1061**/2684/**3137**	**1061**/2685/3138	**1061**/2797/3371	**1061/2592**/3167	**1061**
06	cholesky_mc	113250/144948	285/864/984	**282/787/975**	**282/787**/977	**282**/886/983	**282**/793/978	**282**
07	segmentation	138295/179051	**107**/453/490	**107/447/479**	**107/447**/480	120/476/496	**107/447**/495	120
08	bitonic_mesh	192064/235328	**581/895**/1311	583/**895/1087**	**581/895**/1092	585/**895**/1304	582/898/1123	584
09	dart	202354/223301	828/1243/1401	**784/1083/1279**	829/1123/1354	837/1190/1430	834/1200/1413	805
10	openCV	217453/284108	435/525/522	491/508/515	**432**/492/534	435/502/526	442/**487/512**	434
11	stap_qrd	240240/290123	376/497/695	**370/462/675**	**370/462**/689	377/501/715	**370**/467/696	464
12	minres	261359/320540	215/**309**/407	**207/309/405**	**207/309/405**	**207/309**/407	**207**/311/**405**	**207**
13	cholesky_bdti	266422/342688	**1156/1652**/1865	**1156**/1694/**1844**	1164/1665/1890	**1156**/1769/1874	1166/1697/1889	**1156**
14	denoise	275638/356848	**416**/915/1001	**416**/755/873	446/720/**847**	497/953/1172	456/**670**/860	418
15	sparcT2_core	300109/302663	1198/2249/3558	**1183/2059/2851**	1194/2064/3008	1221/2827/3324	1214/2076/2948	1188
16	gsm_switch	493260/507821	1485/3694/4404	**1479/2345**/2793	1480/2418/**2751**	4235/4149/5169	1483/2446/2813	1833
17	mes_noc	547544/577664	634/1125/1346	**633**/1116/1306	634/**1105**/1309	635/1164/1315	651/1140/**1302**	**633**
18	LU230	574372/669477	**3273**/4548/6310	3390/5479/**5444**	3277/**4484**/5463	3334/4550/6325	3314/4498/5591	3363
19	LU_Network	635456/726999	525/786/1417	**524/784/1352**	526/787/1437	**524**/787/1496	526/786/1488	**524**
20	sparcT1_chip2	820886/821274	899/1404/1601	**874**/1342/**1531**	968/**1297**/1587	914/1453/1610	982/1298/1604	876
21	directrf	931275/1374742	574/762/1092	**490/711**/1086	493/713/**1030**	603/728/1104	527/720/1093	515
22	bitcoin_miner	1089284/1448151	**1489**/1917/2737	1512/1848/**1831**	**1489/1831**/1862	1514/1945/2605	1492/1879/1860	1562

Table 1: Titan23 ϵ=2% cuts. VE is $|V|/|E|$ of the benchmark. T, FP, KEP, hM, and TP give the K=2/3/4 cuts for leaderboard target, FastPart, KEP best-of-20, hMETIS best-of-20, and TritonPart best-of-20; SP_2 is SpecPart at K=2.

ID	$TP_{t\text{-CPLEX}}$	$KEP_{t\text{-CPLEX}}$	$TP_{\text{cut-ORT}}$	$KEP_{\text{cut-ORT}}$	ID	$TP_{t\text{-CPLEX}}$	$KEP_{t\text{-CPLEX}}$	$TP_{\text{cut-ORT}}$	$KEP_{\text{cut-ORT}}$
01	**80.7/515.1/9658.1**	92.8/592.4/11106.8	985/1891/**2604**	**976/1776**/2665	12	**87.8/140.5/975**	101/161.6/1121.2	209/311/407	**207/309/405**
02	**42.5/130.4/556.1**	48.9/150/639.5	246/375/**408**	**243/362**/409	13	**130.8/505.4/11865.3**	150.4/581.2/13645.1	1176/1694/**1908**	**1165/1675**/2056
03	**40.9/87.7/444.7**	47.0/100.9/511.4	179/331/**414**	**171/328**/418	14	**110.5/164.5/212.2**	127.1/189.2/244	459/739/869	**446/737/853**
04	**53/104.6/587.3**	60.9/120.3/675.4	373/528/740	**371/507/669**	15	**184.2/419.1/263**	211.8/482/302.4	1231/2161/**3151**	**1194/2078**/3682
05	**61.4/786.2/18458.9**	70.6/904.1/21227.7	1063/2756/**3169**	**1061/2685**/3209	16	**279.7/566.4/3243.6**	321.7/651.4/3730.1	1489/2513/3016	**1480/2418/2860**
06	**50.5/251.4/2803.4**	58.1/289.1/3223.9	284/802/982	**283/798/977**	17	**222/276.5/504.8**	255.3/318/580.5	663/1155/1380	**640/1111/1309**
07	**59.9/109.3/239.3**	68.9/125.7/275.2	109/452/**486**	**107/447**/493	18	**186.9/626.8/25677.9**	214.9/720.8/29529.6	3381/4637/5789	**3277/4484/5637**
08	**85.8/142.4/679.5**	98.7/163.8/781.4	583/897/1110	**581/895/1092**	19	**207.9/296.7/355.1**	239.1/341.2/408.4	527/790/1542	**526/787/1437**
09	**82.8/263/2635**	95.2/302.4/3030.2	835/1197/**1416**	**829/1123**/1651	20	**404.2/457.1/497**	464.8/525.7/571.5	1018/1399/1713	**968/1297/1594**
10	**89.6/167.4/519.8**	103/192.5/597.8	439/504/**541**	**432/496**/568	21	**272.3/342/431.7**	313.1/393.3/496.5	588/767/**1119**	**518/729**/1314
11	**75.9/129.5/422.1**	87.3/148.9/485.4	373/**475**/703	**370**/490/**689**	22	**310.6/826.4/26311.9**	357.2/950.4/30258.7	1492/**1886/1843**	**1489**/2060/2068

ID	KEP_5	TP_5
01	**2922**	2956
02	**500**	545
03	**428**	434
04	**804**	866
05	**3594**	3625
06	**1392**	1409
07	**645**	665
08	**1384**	1422
09	1226	**1221**
10	**576**	580

ID	KEP+KaHyPar	HyperEF 2.0
01	**974**	**974**
02	**243**	**243**
03	171	**169**
04	**371**	383
05	**1061**	**1061**
06	**282**	283
07	**107**	**107**
08	**581**	585
09	**784**	**784**

Table 2: Titan23 runtime and backend results. Main: K=2/3/4 tuples of CPLEX time (s) and OR-Tools cut for TritonPart (TP) and KEP. Bottom: available CPLEX K=5 cuts (left) and KEP+KaHyPar versus HyperEF 2.0 K=2, ϵ=2% cuts (right).

misses are concentrated in LU230 at K=2, 3 and openCV at K=2. Note that the KEP column does not give a second 180s system result; it shows what FastPart's main solver can obtain with more (i.e., 20) tries. Tables 1, 4 and 5 together cover the full Titan23 and IBM suites.

4.3 KEP Alone and What FastPart Adds

We first use the standalone KEP rows to answer one question: is the main expert strong enough on its own to justify its central role inside FastPart? The answer is yes. Relative to the TritonPart best-of-20 results, KEP wins/ties/loses 15/7/0, 17/1/4 and 14/3/5 on Titan23 for K=2, 3, 4, respectively, for an overall mean relative cut improvement of approximately 1%; against hMETIS [26] the corresponding counts are 15/4/3, 19/3/0 and 20/0/2; and at K=2 it goes 11/3/8 against SpecPart [9]. The strongest per-instance gain over TritonPart in the main setting is 6.4% on directrf at K=2, with another 6.4% on dart at K=3.

Backend and runtime evidence. Table 2 reports Titan23 backend and runtime results. With the same OR-Tools backend, KEP achieves wins or ties versus TritonPart on 22/22, 20/22 and 11/22 Titan23 rows at K=2, 3, 4; further, KEP/OR-Tools achieves wins or ties versus the stronger TritonPart/CPLEX cuts on 21/22, 12/22 and 8/22 rows. Under CPLEX, TritonPart/KEP median times are 88.7/102.0s, 269.8/310.2s and 571.7/657.5s, an approximately 15% cost for the deeper KEP sweep.

All results are replicable with source code [2] including an optional no-KEP mode. No-KEP runs still use the scheduler and all other portfolio machinery, only KEP refinement is disabled.

Component and thread ablations. Table 3 reports component and thread ablations. Removing AL-FM raises mean cut by 0.8% in FastPart and 1.1% in KEP; removing C&C raises the K=4 FastPart mean by 1.6%. C&C's border averages 0.032% of vertices (range 0.01–0.05%), with an improvement adopted in 96% of runs and fallback in

ID	FP_{234}^{-AL}	KEP_{234}^{-AL}	$FP_{4}^{-C\&C}$	ID	FP_{234}^{-AL}	KEP_{234}^{-AL}	$FP_{4}^{-C\&C}$
01	984/1777/2192	989/1797/2485	2219	12	209/311/408	210/312/409	413
02	244/365/410	245/365/411	412	13	1164/1708/1865	1174/1685/1919	1866
03	171/321/374	173/324/413	378	14	419/758/881	450/724/858	882
04	374/508/662	376/510/676	664	15	1195/2074/2870	1211/2084/3036	2916
05	1067/2696/3171	1069/2702/3185	3172	16	1487/2364/2826	1492/2445/2796	2854
06	285/791/981	286/792/985	997	17	637/1120/1318	640/1111/1325	1320
07	108/451/484	108/452/487	489	18	3424/5522/5483	3322/4533/5517	5515
08	586/899/1099	586/900/1108	1097	19	527/790/1369	530/795/1461	1375
09	792/1089/1287	841/1132/1366	1296	20	880/1347/1544	977/1304/1605	1559
10	495/512/521	436/498/542	525	21	495/717/1094	500/721/1041	1109
11	372/464/682	373/464/699	688	22	1519/1862/1854	1498/1850/1893	1853

8-thread median time-to-target	K=2	K=3	K=4
FastPart / TritonPart (s)	45.3/118.1	269.7/487.8	188.6/1231.4
TritonPart/FastPart speedup	2.6×	1.8×	6.5×

Table 3: Titan23 component and eight-thread ablations at ϵ=2%. IDs follow Table 1; subscript 234 denotes K=2/3/4, superscripts identify the removed component. Times are matched median time-to-target (TritonPart cut value) seconds on an Apple M5.

ID	K2/2%	t(s)	K2/20%	t(s)	K4/5%	t(s)
01	**974**/**974**/**974**	**39.2**/157.9	**581**/631/583	24.4	**2184**/2754/3088/3542	36.5/38/**11**/3941
02	**243**/244/**243**	**30.6**/137.7	**199**/244/200	831.2	**413**/529/716/712	36.4/24/**5**/146
03	**169**/**169**/**169**	**26.4**/73	**90**/91/91	17.4	**315**/344/518/373	158.3/22/**6**/165
04	**371**/385/383	**38.8**/141.7	**345**/**345**/353	27	**681**/858/883/1027	37.8/32/**16**/148
05	**1061**/**1061**/**1061**	**29.2**/58.3	**1061**/**1061**/**1061**	23	**3193**/3719/3856/4248	42.3/40/**20**/7558
06	**282**/283/283	**35.9**/111.6	**281**/479/**281**	65.9	**965**/1147/1168/1555	39.8/39/**10**/592
07	**107**/**107**/**107**	**41.6**/75.2	**78**/**78**/**78**	28.5	**444**/464/589/537	35.5/28/**16**/146
08	**582**/611/585	**161.2**/197.8	**505**/506/506	3335	**1141**/1300/1291/1332	66.4/60/**31**/231
09	**784**/**784**/**784**	**59.5**/220.1	**539**/**539**/**539**	163.7	**1059**/1388/1641/1967	71.6/69/**19**/664
10	**451**/648/–	**177**/214.8	**443**/473/–	178	**504**/1288/1415/1347	167.3/68/**18**/225
11	**370**/456/–	**50**/278.6	**275**/**275**/–	29.8	**615**/1014/1150/993	67.8/101/**25**/190
12	**207**/239/–	**49.1**/256.4	**181**/191/–	172.5	**341**/556/610/543	74.6/91/**22**/260
13	**1156**/**1156**/–	**55.3**/306.9	**847**/848/–	161.1	**1783**/2149/2342/2276	79.9/98/**34**/4863
14	**416**/**416**/–	**61.5**/211.5	**220**/**220**/–	37	**931**/987/1154/1048	73.3/52/**42**/172
15	**1183**/**1183**/–	**43**/276.9	**890**/918/–	856.1	**2450**/2820/2857/3183	67.4/116/**64**/235
16	**1479**/1630/–	**135.3**/726.7	**1407**/**1407**/–	58.6	**2654**/4065/14201/4483	**46**/182/51/1514
17	**634**/**634**/–	**72.1**/599.4	**617**/**617**/–	863	**1262**/1546/1573/1785	**51.6**/152/87/302
18	**3465**/3488/–	**3473.4**/8666.5	**2859**/2923/–	851	**5627**/5678/6229/6010	67.7/161/**67**/10080
19	**524**/**524**/–	**76**/773.5	**524**/**524**/–	43.5	**1044**/1062/1754/1073	**45.3**/140/87/230
20	**873**/874/–	**103.9**/885.3	**655**/757/–	175.3	**1475**/1989/2288/2654	**82.7**/242/128/343
21	**630**/639/–	**179.8**/1302.2	**295**/**295**/–	53.8	**989**/1152/1164/1220	**61.6**/197/153/253
22	**1489**/1542/–	**144.4**/2195	**1281**/1282/–	68	**1579**/3869/4496/3984	**79.5**/365/221/1101

Table 4: Recent-method comparison on Titan23. In K2/2%, cuts are FastPart/SHyPar/HyperEF 2.0, and the adjacent time column gives FastPart/SHyPar runtimes. In K2/20%, cuts are FastPart/SHyPar/HyperEF 2.0, but only FastPart runtime is available. In K4/5%, both cuts and runtimes are FastPart/EasyPart/hMETIS/TritonPart. All runtimes are in seconds.

4%. All outputs pass the same two-sided checker. At eight threads, FastPart/TritonPart median times are 45.3/118.1s, 269.7/487.8s and 188.6/1231.4s, yielding 2.6×, 1.8× and 6.5× speedups at K=2, 3, 4.

FastPart answers a different question: what does the controller add under one fixed budget? On Titan23, FastPart reaches 18/22, 20/22 and 22/22 targets at K=2, 3, 4, versus KEP's 16/22, 20/22 and 18/22. Median FastPart/TritonPart times are 33.2/88.7s, 156.6/269.8s and 141.8/571.7s, or 3.4×, 3.1× and 8.6× speedups; ten K=4 cases exceed 10×. On IBM, the best cut appears within 10s on 7/18 cases, 20s on 11/18, and 60s on 14/18.

We note that IBM winners span multiple flows – recursive (8), spectral (5), parallel-refine (3), and direct (2) – showing that the portfolio draws on multiple experts. Because KEP is a deeper average-best-of-20 sweep, row-wise FastPart/KEP differences are not portfolio regressions; with one fixed run, the controller preserves the low-K quality envelope while broadening coverage and accelerating larger K.

4.4 Comparison to Recent Methods

We may also separate matched reruns from published-setting comparisons. In both contexts, FastPart has no loss cases. At K=2, ϵ=2%, it goes 23/17/0 against SHyPar over Titan23+IBM, and it ties or outperforms HyperEF 2.0 on all nine published Titan23 rows. In the other published settings, it remains strong: on IBM K=2, ϵ=10%, it goes 2/16/0 against both SHyPar and HyperEF 2.0; on Titan23 K=2, ϵ=20%, it goes 12/10/0 against SHyPar and 5/4/0 against HyperEF 2.0; and on Titan23 K=4, ϵ=5%, it goes 22/0/0 against EasyPart, hMETIS and TritonPart. Where matched runtime data are available, FastPart shows favorably against SHyPar, EasyPart and TritonPart, while hMETIS remains faster but weaker in cut. Tables 4 and 5 give full data underlying this assessment.

Bench	target/FP	K2/2%	t(s)	K2/10%	t(s)	Bench	target/FP	K2/2%	t(s)	K2/10%	t(s)
ibm01	203/**201**	**201**/**201**	156.2/**16.4**	**166**/**166**/**166**	5.4	ibm10	1283/**1254**	**1254**/**1254**	160.8/**37.9**	**1244**/**1244**/**1244**	157.2
ibm02	326/**324**	**324**/328	163.6/**17.3**	**262**/**262**/**262**	7.7	ibm11	1054/**1051**	**1051**/1062	**26.8**/53.1	**763**/**763**/**763**	21.5
ibm03	963/**952**	**952**/955	155.2/**19.7**	**950**/**950**/**950**	9.5	ibm12	2063/**1920**	**1920**/1987	162.9/**41**	**1841**/**1841**/**1841**	18.3
ibm04	592/**579**	**579**/580	167.6/**27.2**	**388**/**388**/**388**	6.8	ibm13	848/**831**	**831**/**831**	**33.9**/87.9	**655**/**655**/**655**	158.6
ibm05	1728/**1706**	**1706**/1707	23.7/**21.7**	**1645**/**1645**/**1645**	11.9	ibm14	1905/**1842**	**1842**/1849	**52.2**/117.7	**1516**/1534/1520	168.1
ibm06	977/**963**	**963**/978	158.9/**22**	**733**/**733**/**733**	10	ibm15	2800/**2728**	**2728**/**2728**	**60.1**/212.3	**2121**/2135/2127	3325.2
ibm07	952/**880**	**880**/894	164.4/**48**	**760**/**760**/**760**	15.3	ibm16	2121/**1836**	**1836**/1902	**182.5**/207.9	**1619**/**1619**/**1619**	40.8
ibm08	1143/**1140**	**1140**/**1140**	**17.4**/27.7	**1120**/**1120**/**1120**	156.6	ibm17	2343/**2284**	**2284**/2285	176.4/**137.9**	**1989**/**1989**/**1989**	42.2
ibm09	621/**620**	**620**/**620**	**17.4**/33.6	**519**/**519**/**519**	14.5	ibm18	1527/**1521**	**1521**/**1521**	**53.1**/127.7	**1520**/**1520**/**1520**	43.3

Table 5: Recent-method comparison on IBM. `target/FP` gives the leaderboard target and FastPart cut at K=2, ϵ=2%. K2/2% gives FastPart/SHyPar cuts and the corresponding FastPart/SHyPar runtimes. K2/10% gives FastPart/SHyPar/HyperEF 2.0 cuts, while the adjacent time column is FastPart-only because comparable baseline runtimes are not available. Runtimes are in seconds.

ID	TP	KEP	hM_{rep}	TP_t	KEP_t
01	1112/2539/3374	**1039**/**1771**/**2578**	1906/2842/3622	**66.4**/**168.9**/**414.6**	71/172.6/423.2
02	248/522/756	248/**392**/**429**	**245**/613/734	46.5/**116.9**/**219.8**	**44.2**/124.7/241.7
03	182/460/766	**178**/**363**/**471**	741/904/999	55.7/**137.4**/252.3	**55.3**/140.1/**241.8**
04	421/736/1462	**385**/**515**/**756**	572/617/1164	**78**/**195.4**/417.7	79/200.9/**415.6**
05	1134/3709/4588	**1061**/**2965**/**3286**	**1061**/4090/3558	**83.8**/**188.6**/378.2	90.4/200.8/**368.1**
06	333/1265/2050	**325**/**974**/**1108**	1309/1109/1949	**80.2**/**132.8**/273.2	85.2/137.2/**262.2**
07	194/660/1038	**167**/**494**/**567**	1533/2442/1859	**67.4**/**188.1**/385.9	73.3/203.5/**385.3**
08	592/1211/2142	**589**/**927**/**1130**	600/1268/2868	**93.4**/153/385.8	93.6/**144.5**/**368.4**
09	839/1662/2347	**804**/**1261**/**1462**	821/3590/3450	**85.3**/204/**307.4**	90.4/**199**/318.9
10	591/799/873	**583**/**739**/**753**	608/1493/1544	259.7/**704.2**/1774.6	**258.6**/766.4/**1766.5**
11	450/921/1250	**387**/**556**/**788**	447/1761/2262	**83.9**/**143.4**/**335.3**	84.6/150/352.8
12	271/562/766	**240**/**385**/**467**	1408/2311/1516	**91.7**/206.2/336.4	94.9/**199.4**/**329.3**
13	1246/2343/3300	**1225**/**1995**/**1966**	3090/2492/3585	259/**706.7**/**1474.3**	**246.3**/751.8/1499.8
14	467/938/1596	**463**/**809**/**1087**	1881/3242/3971	111.8/215.7/**437.4**	**110.9**/**214**/455.4
15	1248/2926/4994	**1214**/**2324**/**3103**	1241/5998/5133	**360**/1028/2054.3	380.4/**971**/**2000.8**
16	**1783**/4312/6731	1851/**2603**/**3084**	4847/8507/8516	**487.5**/1115/**2268**	524.5/**1103**/2325.8
17	827/2110/3398	**751**/**1260**/**1485**	7403/10487/7750	**470.5**/**1108**/**1670.9**	487/1155.7/1798.3
18	3637/6663/8212	**3616**/**5855**/**5709**	4136/11458/10818	498.7/1226.8/2877.9	**479.4**/**1171.8**/**2748.5**
19	730/1874/3018	**670**/**1169**/**1614**	5345/5112/6019	**527.2**/**1592.1**/2999.6	568.7/1663.5/**2880.3**
20	1094/2309/3747	**943**/**1430**/**1864**	5052/1939/12794	614/**1568.8**/**3457.4**	**595.2**/1569.5/3642.7
21	683/1082/2729	**614**/**862**/**1191**	6222/9862/6500	**563.7**/**1158.4**/**3119.1**	608.7/1232.4/3326.6
22	1491/2928/3217	**1489**/**2645**/**3092**	**1489**/15144/5542	**449.4**/1381.6/**2390.6**	473.6/**1354.6**/2513.7

Table 6: Tight-balance results for ϵ=0.1%. Each cell is the K=2/3/4 tuple. TP, KEP, and hM_{rep} are TritonPart, KEP, and repaired hMETIS; TP_t and KEP_t are runtimes in seconds.

4.5 Tight-balance stress test

At ϵ=0.1%, Table 6 compares KEP with TritonPart and repaired hMETIS (hMETIS does not natively support balance below 1%). For K=2, 3, 4, KEP goes 20/1/1, 22/0/0, 22/0/0 against TritonPart and 19/22, 22/22, 22/22 against hMETIS. Mean cut deltas are −33, −465, −1107 (5.2%, 24.9%, 38.9%); KEP is faster on 7, 8, 11 cases, with median TP/KEP times 102.6/102.9s, 211.0/208.8s, 427.6/439.3s. This stress test supports AL-FM under near-exact balance.

Across all matched rows, win/tie/loss totals are 113/69/0 across 182 recent-work comparisons for FastPart and, across 66 tight-balance rows each, 64/1/1 for KEP versus TritonPart and 63/2/1 versus repaired hMETIS. Every reported partition passes the same two-sided checker; the repair queue contributes only if an intermediate cut becomes feasible within budget. *We note that public baselines often use a laxer, upper-bound-only rule*; this inconsistency is avoided in [3].

5 Conclusion

FastPart combines one strong expert, KEP, with a small number of complementary solver flows under a single fixed budget. KEP provides most of the low-K quality through K-adaptive routing, AL-FM and localized C&C. FastPart then turns that strong expert into a more reliable system by improving coverage and time-to-target, especially at larger K, across Titan23 and IBM. In the recent-method comparisons we could evaluate, FastPart has no loss cases. Gains are largest when fixed budgets are tight and when one search bias alone misses the best structure, which is portfolio-friendly. FastPart is among the methods evaluated in the recent hypergraph partitioning leaderboard effort [3].

Acknowledgments. ABKGroup research is broadly supported by the Samsung AI Center.

AI Disclosure. ChatGPT Pro, GPT-5-class models and OpenAI Codex were used for writing, editing and code assistance.

References

[1] UCSD ABKGroup, TritonPart_OpenROAD, https://github.com/ABKGroup/TritonPart_OpenROAD, 2023.

[2] UCSD ABKGroup, FastPart artifact repository, https://github.com/ABKGroup/FastPart/tree/iccad26-camera-ready, 2026.

[3] UCSD ABKGroup, Hypergraph Partitioning Leaderboard, https://abkgroup.github.io/HypergraphPartitioning, 2026.

[4] T. Ajayi, V. A. Chhabria, M. Fogaça, S. Hashemi, A. Hosny, A. B. Kahng, M. Kim, J. Lee, U. Mallappa, M. Neseem, G. Pradipta, S. Reda, M. Saligane, S. S. Sapatnekar, C. Sechen, M. Shalan, W. Swartz, L. Wang, Z. Wang, M. Woo and B. Xu, "Toward an Open-Source Digital Flow: First Learnings from the OpenROAD Project", *Proc. ACM/IEEE Design Automation Conference*, 2019, pp. 1–4.

[5] Y. Akhremtsev, T. Heuer, P. Sanders and S. Schlag, "Engineering a Direct k-Way Hypergraph Partitioning Algorithm", *Proc. SIAM Symposium on Algorithm Engineering and Experiments*, 2017.

[6] C. J. Alpert and A. B. Kahng, "Recent Directions in Netlist Partitioning: A Survey", *Integration: The VLSI Journal* 19(1–2) (1995), pp. 1–81.

[7] C. J. Alpert, "The ISPD98 Circuit Benchmark Suite", *Proc. ACM/IEEE International Symposium on Physical Design*, 1998, pp. 80–85.

[8] I. Bustany, G. Gasparyan, A. B. Kahng, I. Koutis, B. Pramanik and Z. Wang, "An Open-Source Constraints-Driven General Partitioning Multi-Tool for VLSI Physical Design", *Proc. IEEE/ACM International Conference on Computer-Aided Design*, 2023, pp. 1–9.

[9] I. Bustany, A. B. Kahng, I. Koutis, B. Pramanik and Z. Wang, "SpecPart: A Supervised Spectral Framework for Hypergraph Partitioning Solution Improvement", *Proc. IEEE/ACM International Conference on Computer-Aided Design*, 2022.

[10] I. Bustany, A. B. Kahng, I. Koutis, B. Pramanik and Z. Wang, "K-SpecPart: Supervised Embedding Algorithms and Cut Overlay for Improved Hypergraph Partitioning", *arXiv preprint arXiv:2305.06167*, 2023.

[11] A. E. Caldwell, A. B. Kahng and I. L. Markov, "Improved Algorithms for Hypergraph Bipartitioning", *Proc. ACM/IEEE Asia and South Pacific Design Automation Conference*, 2000, pp. 661–666.

[12] U. V. Catalyürek and C. Aykanat, "PaToH: A Multilevel Hypergraph Partitioning Tool, Version 3.0", Bilkent University, Dept. of Computer Engineering, Ankara, 1999.

[13] Ü. V. Çatalyürek, K. Devine, M. Faraj, L. Gottesbüren, T. Heuer, H. Meyerhenke, P. Sanders, S. Schlag, C. Schulz, D. Seemaier and D. Wagner, "More Recent Advances in (Hyper)Graph Partitioning", *ACM Computing Surveys* 55(12) (2023), pp. 1–38.

[14] P. K. Chan, M. D. F. Schlag and J. Y. Zien, "Spectral k-Way Ratio-Cut Partitioning and Clustering", In *Proc. ACM/IEEE Design Automation Conference*, 1993, pp. 749–754.

[15] J. Cong and S. K. Lim, "Multiway Partitioning with Pairwise Movement", *Proc. IEEE/ACM International Conference on Computer-Aided Design*, 1998, pp. 512–516.

[16] S. Dutt and H. Theny, "Partitioning Around Roadblocks: Tackling Constraints with Intermediate Relaxations", *Proc. IEEE/ACM International Conference on Computer-Aided Design*, 1997, pp. 350–355.

[17] C. M. Fiduccia and R. M. Mattheyses, "A Linear-Time Heuristic for Improving Network Partitions", *Proc. ACM/IEEE Design Automation Conference*, 1982, pp. 175–181.

[18] C. P. Gomes and B. Selman, "Algorithm Portfolios", *Artificial Intelligence* 126(1–2) (2001), pp. 43–62.

[19] L. Gottesbüren, T. Heuer, N. Maas, P. Sanders and S. Schlag, "Scalable High-Quality Hypergraph Partitioning", *ACM Transactions on Algorithms* 20(1) (2024), pp. 1–54.

[20] L. Gottesbüren, T. Heuer and P. Sanders, "Parallel Flow-Based Hypergraph Partitioning", *arXiv preprint arXiv:2201.01556*, 2022.

[21] L. Hagen and A. B. Kahng, "New Spectral Methods for Ratio Cut Partitioning and Clustering", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 11(9) (1992), pp. 1074–1085.

[22] B. Hendrickson and R. Leland, "A Multilevel Algorithm for Partitioning Graphs", *Proc. ACM/IEEE Supercomputing Conference*, 1995.

[23] M. R. Hestenes, "Multiplier and Gradient Methods", *Journal of Optimization Theory and Applications* 4 (1969), pp. 303–320.

[24] TILOS AI Institute, HypergraphPartitioning Repository, https://github.com/TILOS-AI-Institute/HypergraphPartitioning, 2025.

[25] G. Karypis and V. Kumar, "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs", *SIAM Journal on Scientific Computing* 20(1) (1998), pp. 359–392.

[26] G. Karypis and V. Kumar, *hMETIS: A Hypergraph Partitioning Package, Version 1.5.3*, University of Minnesota, 1998.

[27] G. Karypis and V. Kumar, "Multilevel Algorithms for Multi-Constraint Graph Partitioning", *Proc. ACM/IEEE Supercomputing Conference*, 1998.

[28] G. Karypis and V. Kumar, "Multilevel k-Way Partitioning Scheme for Irregular Graphs", *Journal of Parallel and Distributed Computing* 48(1) (1998), pp. 96–129.

[29] B. W. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs", *Bell System Technical Journal* 49(2) (1970), pp. 291–307.

[30] D. LaSalle and G. Karypis, "A Parallel Hill-Climbing Refinement Algorithm for Graph Partitioning", *Proc. ACM International Conference on Parallel Processing*, 2016, pp. 236–241.

[31] J. Nocedal and S. J. Wright, *Numerical Optimization*, 2nd ed., Springer, 2006.

[32] A. Pothen, H. D. Simon and K.-P. Liou, "Partitioning Sparse Matrices with Eigenvectors of Graphs", *SIAM Journal on Matrix Analysis and Applications* 11(3) (1990), pp. 430–452.

[33] M. J. D. Powell, "A Method for Nonlinear Constraints in Minimization Problems", *Optimization*, Academic Press, 1969, pp. 283–298.

[34] H. Sajadinia, A. Aghdaei and Z. Feng, "SHyPar: A Spectral Coarsening Approach to Hypergraph Partitioning", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2025).

[35] H. Sajadinia and Z. Feng, "HyperEF 2.0: Spectral Hypergraph Coarsening via Krylov Subspace Expansion and Resistance-Based Local Clustering", *Proc. IEEE/ACM International Conference on Computer-Aided Design*, 2025.

[36] P. Sanders and C. Schulz, "Engineering Multilevel Graph Partitioning Algorithms", *Proc. European Symposium on Algorithms* (LNCS 6942), 2011, pp. 469–480.

[37] P. Sanders and C. Schulz, "Distributed Evolutionary Graph Partitioning", *Proc. SIAM Symposium on Algorithm Engineering and Experiments*, 2012, pp. 16–29.

[38] S. Schlag, T. Heuer, L. Gottesbüren, Y. Akhremtsev, C. Schulz and P. Sanders, "High-Quality Hypergraph Partitioning", *ACM Journal of Experimental Algorithmics* 27 (2022), pp. 1–39.

[39] H. D. Simon and S. Teng, "How Good is Recursive Bisection?", *SIAM Journal on Scientific Computing* 18(5) (1997), pp. 1436–1445.

[40] A. Strehl and J. Ghosh, "Cluster Ensembles—A Knowledge Reuse Framework for Combining Multiple Partitions", *Journal of Machine Learning Research* 3 (2002), pp. 583–617.

[41] S. Tong, H. Li, J. Xu, C. Pei, W. Yu, S. Liu and J. Shen, "EasyPart: An Effective and Comprehensive Hypergraph Partitioner for FPGA-Based Emulation", *Proc. IEEE/ACM International Conference on Computer-Aided Design*, 2024, pp. 1–9.

[42] S. Tong, C. Pei and W. Yu, "BlasPart: A Deterministic Parallel Partitioner for Balanced Large-Scale Hypergraph Partitioning", *Proc. ACM/IEEE Design Automation Conference*, 2025, pp. 1–7.

[43] J. Y. Zien, M. D. F. Schlag and P. K. Chan, "Multilevel Spectral Hypergraph Partitioning with Arbitrary Vertex Sizes", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 18(9) (1999), pp. 1389–1399.