EVOPART: AN OPEN, LIGHTWEIGHT, AND STRONG HYPERGRAPH PARTITIONER EVOLVED WITH LLM CODER

Anonymous authors
Paper under double-blind review

ABSTRACT

Hypergraph partitioning is a fundamental NP-hard problem in VLSI physical design, scientific computing and, increasingly, LLM infrastructure. Yet the field lacks an open partitioner that is at once lightweight, strong and fast: the most influential tool, hMETIS, is closed-source, and the leading open framework, KaHyPar, spans some 27,000 lines of C++, so recent work typically wraps hMETIS as a black box or replaces a single sub-step of KaHyPar. Coding agents such as AlphaEvolve and its open-source counterpart OpenEvolve offer a way forward, having already found better programs for problems including matrix multiplication. Building on OpenEvolve, we design a harness for evolving a whole solver rather than a single heuristic: it places programs in the MAP-Elites grid by behavior (running time and relative strength across k) rather than text, deduplicates programs by partition fingerprinting, and feeds the measured per-instance cut and wall time back into the prompt. Starting from a round-robin initial program whose cut is $97\times$ KaHyPar's, a single 200-iteration run yields EvoPart: a lightweight and strong partitioner of 1,186 lines of single-threaded C++ in one file, $22.5\times$ less code than KaHyPar. On the full ISPD98 and Titan23 benchmarks, its cut is 5.5% lower than KaHyPar's in 70% of KaHyPar's wall time and 5.1% lower than hMETIS's at equal wall time, with 30 of the 40 instances unseen during evolution. Two of its components are, to our knowledge, absent from existing partitioners: racing two coarsenings with opposite net-size priorities inside the restarts, and re-partitioning the final blocks pair by pair. We release the partitioner and the harness at https://anonymous.4open.science/r/EvoPart-4F6C.

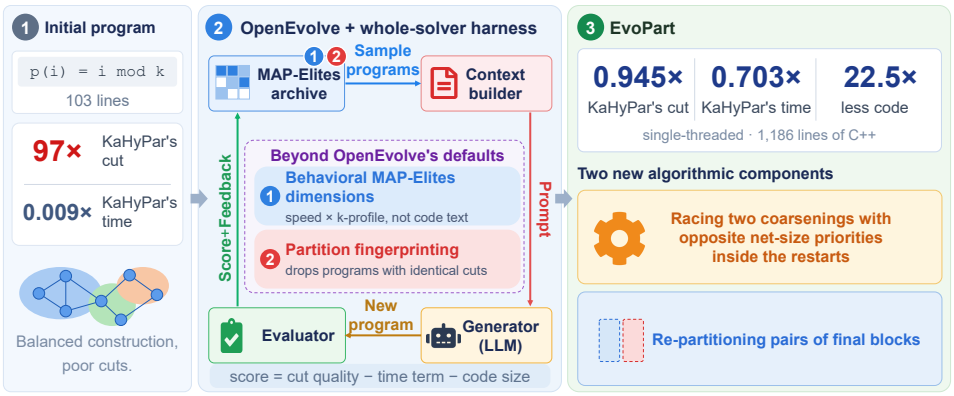

Figure 1: From a round-robin initial program to a small open partitioner. *Left:* the starting program assigns vertex i to part $i \bmod k$ and cuts $97\times$ as many nets as KaHyPar. *Middle:* LLM code evolution with OpenEvolve (Sharma, 2025) under a harness tailored to evolving a whole solver; ①–② mark where it departs from OpenEvolve's defaults. *Right:* the evolved partitioner, EvoPart, compared with KaHyPar over the full ISPD98 and Titan23 suites, and its two new algorithmic components (section 4).

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

1 INTRODUCTION

Hypergraph partitioning is a fundamental NP-hard problem: cut a system into capacity-bounded parts while keeping the communication between them as small as possible. It has a long history in VLSI physical design and scientific computing, and it now recurs throughout machine-learning infrastructure. Placing the experts of a mixture-of-experts layer across devices is itself such a problem: MoETuner (Go & Mahajan, 2025) assigns experts to GPUs under load-balance constraints while minimizing the token traffic routed between them. Distributed GNN training systems such as DistDGL (Zheng et al., 2020) partition the input graph with METIS (Karypis & Kumar, 1998) so that few edges cross machines, and Demirci et al. (2022) further show that partitioning with the hypergraph partitioners PaToH (Çatalyürek & Aykanat, 2011) and KaHyPar (Schlag et al., 2023) yields faster distributed GCN training. Progress on the underlying problem therefore compounds across domains.

Despite its importance, the field lacks a good open foundation. The most influential partitioner, hMETIS (Karypis et al., 1997), is closed source, and several follow-up works (Bustany et al., 2022; 2023) embed it as a black-box subroutine. The main open-source alternative, KaHyPar (Schlag et al., 2023), has grown through years of incremental extensions into a large codebase that combines many algorithms in each phase; architecture-level changes are therefore hard, and works built on it typically only modify a single stage of the pipeline (e.g., SHyPar (Sajadinia et al., 2025), IMPart (Zhu et al., 2026b)). Derived problems such as multi-FPGA system partitioning, which imposes additional physical constraints, are therefore usually addressed with multilevel frameworks implemented from scratch (Zhu et al., 2026a), which lack the engineering refinements of mature partitioners such as hMETIS and KaHyPar.

We therefore aim for an algorithm that is *better* than the leading hypergraph partitioners, not merely competitive with them, while remaining fully open source and small enough for researchers to read and extend easily. KaHyPar's partitioning library and the algorithm libraries it depends on comprise 26,628 lines of C++ (excluding comments and blank lines, here and throughout); our evolved partitioner, EvoPart, is only 1,186 lines in one file, 22.5× smaller. It is also stronger: on the union of the ISPD98 (Alpert, 1998) and Titan23 (Murray et al., 2013) benchmark suites, it achieves a 5.5% lower cut than KaHyPar in 70% of KaHyPar's wall time (Table 1). hMETIS, KaHyPar and EvoPart all run single-threaded. EvoPart thus offers the community a new open starting point — and not merely a compact re-implementation of known techniques: two of its components have, to our knowledge, no counterpart in existing partitioners (section 4):

Table 1: Code size, cut and wall time of EvoPart over the full ISPD98 and Titan23 suites, normalized to KaHyPar (= 1.000); lower is better.

	KaHyPar	EvoPart
C++ lines	26,628	**1,186**
Cut	1.000	**0.945**
Wall time	1.000	**0.703**

1. **Racing two coarsenings with opposite net-size priorities inside the restarts**, so that all pins of a large net, which the heavy-edge rating barely sees, can be merged into one cluster whenever that yields a better bisection.
2. **Re-partitioning pairs of final blocks**, which can rearrange a split that the bisection tree committed to before its children existed, and thereby fine-tune the partition locally, one pair of blocks at a time.

The algorithm is *evolved* by an LLM coding agent built on OpenEvolve (Sharma, 2025), an open-source implementation of the AlphaEvolve (Novikov et al., 2025) paradigm. The starting point is not a partitioner at all: an initial program of about a hundred lines that reads the hypergraph and assigns vertex i to part $i \bmod k$, so that every data structure and every phase of the final program is the model's own. To evolve a whole solver rather than a single heuristic, we build a harness around OpenEvolve that departs from its defaults in two places, behavioral MAP-Elites dimensions and partition fingerprinting, and feeds the measured cut and wall time back into the prompt (section 3).

Our contribution is therefore two-fold:

- **An open, lightweight and strong hypergraph partitioner: EvoPart.** A single-threaded, self-contained partitioner of 1,186 lines in a single file whose cut is 5.5% lower than KaHyPar's and 5.1% lower than hMETIS's over the full ISPD98 and Titan23 suites, in 70% of KaHyPar's wall

time and the same wall time as hMETIS, released together with the evolution harness. Two of its components are, to our knowledge, new to hypergraph partitioning: racing two coarsenings with opposite net-size priorities inside the restarts, and re-partitioning pairs of final blocks.

- **A harness for evolving a whole solver against state-of-the-art tools**, instantiated for hypergraph partitioning. It organizes the MAP-Elites grid by behavior rather than text, with speed and k-profile descriptors, and removes behavioral duplicates from each island with partition fingerprints; these are the two places where it departs from OpenEvolve's defaults. It feeds each program's measured cut and wall time on every training cell, together with its own printed diagnostics, back into the prompt, and ranks programs by a score that balances cut quality against running time and code length. From a round-robin initial program, a single 200-iteration run reaches a partitioner that surpasses hMETIS and KaHyPar in cut.

2 PRELIMINARIES

2.1 HYPERGRAPH PARTITIONING

A hypergraph $H = (V, E)$ generalizes a graph: each hyperedge (*net*) $e \in E$ is an arbitrary subset of V, whose vertices are its *pins*. A k-way partition $\Pi = \{V_1, \ldots, V_k\}$ assigns each vertex to exactly one block. The goal of hypergraph partitioning is to minimize the *cut-net* metric, i.e., the number of nets whose pins span more than one block, subject to a balance constraint with imbalance factor ε:

$$|V_p| \;\leq\; (1+\varepsilon)\left\lceil \frac{|V|}{k} \right\rceil \qquad \text{for every block } p. \tag{1}$$

The problem is NP-hard, and every leading partitioner — hMETIS (Karypis et al., 1997), PaToH (Çatalyürek & Aykanat, 2011) and KaHyPar (Schlag et al., 2023) — follows the *multilevel* scheme of three phases: *coarsening*, which builds a hierarchy of smaller hypergraphs by repeatedly merging tightly connected vertices; *initial partitioning*, which partitions the coarsest hypergraph directly; and *uncoarsening and refinement*, which projects the partition back level by level and improves it at each level with a local search such as FM (Fiduccia & Mattheyses, 1988). A k-way partition is obtained either by *recursive bisection* or by a *direct k-way* method. Since coarsening and initial partitioning are randomized, a partitioner may perform several *restarts*, i.e., independent multilevel runs with different random seeds, and keep the best; a finished partition can also be improved further by *V-cycles*, which repeat coarsening and refinement without merging vertices of different blocks. We use this vocabulary in section 4 to describe the evolved program.

2.2 LLM CODE EVOLUTION

LLM code evolution opens a new route for combinatorial problems such as hypergraph partitioning: it replaces the hand-designed mutation operator of genetic programming with a language model. FunSearch (Romera-Paredes et al., 2024) pairs an LLM with a scorer and a programs database, evolving a short Python priority function inside a fixed skeleton, and produced new cap-set constructions and online bin-packing heuristics. Subsequent work refines the search. EoH (Liu et al., 2024) evolves a natural-language *thought* alongside the code of each heuristic; ReEvo (Ye et al., 2024) adds LLM-written reflections on earlier generations; HSEvo (Dat et al., 2025) tracks population diversity explicitly and balances it against convergence; MCTS-AHD (Zheng et al., 2025) replaces the population with a Monte Carlo search tree; and RefineEvo (Wu et al., 2026) schedules the evolutionary operators with a planner and keeps an experience pool of positive and negative lessons. All of them, however, evolve the same kind of object: one short function, such as a priority or scoring rule, while the rest of the solver stays fixed. LLaMEA (Van Stein & Bäck, 2024) and PartEvo (Hu & Zhang, 2026) go further and let the LLM write a complete metaheuristic, with every step producing the whole program rather than a patch.

We also start from almost nothing — an initial program that assigns vertices round-robin — but the partitioners that evolution produces grow to about a thousand lines of C++, and a candidate must compile and run 90 times before it has a score. A program of that size is built up by many localized edits rather than rewritten in one piece at every step. AlphaEvolve (Novikov et al., 2025) is designed for this regime: unlike FunSearch, it evolves entire code files in any language, applies edits as SEARCH/REPLACE diffs, keeps candidates in an island-structured MAP-Elites

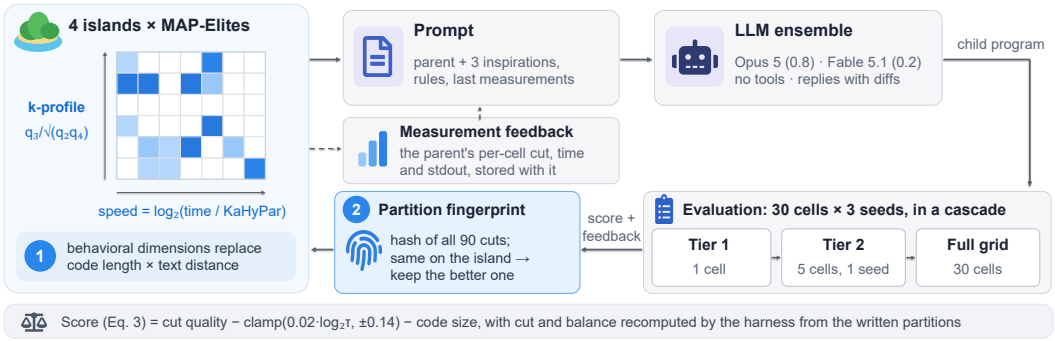

Figure 2: One iteration of the evolution loop. Blue marks where we depart from OpenEvolve's defaults: ① behavioral MAP-Elites dimensions and ② partition fingerprinting in the population. The measurements of every evaluation are stored with the program and enter the prompt whenever that program is chosen as the parent.

archive (Mouret & Clune, 2015), and tolerates evaluations that take hours. This design from Google DeepMind has been influential, and several open-source systems in the same family follow it (Lange et al., 2026; Assumpção et al., 2025). Since AlphaEvolve itself is closed source, we build on OpenEvolve (Sharma, 2025), an open-source reimplementation of its design.

3 Evolution Setup

We evolve the partitioner in a single run of OpenEvolve, starting from an initial program that assigns vertices round-robin. Figure 2 shows one iteration: a prompt is built around a parent drawn from the population, an LLM answers with diffs, the child is evaluated on a training grid, and its score and its two behavioral descriptors, speed and k-profile (① below), decide whether and where it enters the population, while its measurements are fed back to the prompt that edits it next. We describe the parts of the loop in this order; where we depart from OpenEvolve's defaults is marked in blue in the figure and numbered ①–② below.

- **Evolved partitioner.** What the LLM edits is one C++ file, `partition.cpp`, that is the whole partitioner: parser, data structures, algorithm and output. The starting file reads the `.hgr`, assigns vertex i to part $i \bmod k$ and writes the result — balanced, but cutting nearly every net ($97\times$ KaHyPar's cut). The model edits it with SEARCH/REPLACE diffs against the text, and the evaluator recomputes cut and balance from the partition file, so the program cannot redefine the objective.
- **Prompt.** Each prompt holds four parts: a fixed system message that states the task, the training grid, the scoring rules of equation 3 and the rules (one file, single-threaded), but contains no code; the parent program with a short description of its algorithm, which the lineage maintains through the same diffs; the parent's measurements (see *measurement feedback* below); and three inspirations, two of the best programs and one random program of the parent's island, each with its score and description. The model has no tools and cannot see the partitioners installed on our machine; a reply without a diff is discarded.
- **Population.** Candidates are kept in an island-based MAP-Elites structure (Mouret & Clune, 2015) — four islands, 150 programs, migration every 25 generations — from which the prompt sampler draws the parent and the inspirations for each iteration. We change two things about how a program enters and stays in it.
  - ① **Behavioral MAP-Elites dimensions.** OpenEvolve places a program in the MAP-Elites grid by two textual features, *complexity* (the character count of the source) and *diversity* (a text distance to about twenty reference programs), so programs that differ only in comments, dead code or renamed variables, exactly what LLM edits keep producing, occupy different cells and are all kept. We replace both with *behavioral* descriptors computed by the evaluator:

$$\text{speed} \;=\; \text{clamp}\big(\log_2 \tau,\, [-2, 5]\big), \qquad k\text{-profile} \;=\; \frac{q_3}{\sqrt{q_2\, q_4}} \;\text{clamped to}\; [0.5, 2], \tag{2}$$

where τ is the slowdown relative to KaHyPar and q_k is the geometric mean of the cut-quality ratios c^{ref}/c over the cells with that k; each dimension has ten bins. The log scale makes $2\times$ faster and $2\times$ slower symmetric steps, and the k-profile, centered at 1, compares a program's skill at a non-power-of-two k with its skill at powers of two, which tends to separate recursive bisection from direct k-way strategies.

  - ② **Partition fingerprinting.** Programs that produce the same partitions, for instance one carrying dead code or computing the same result at a different cost, pass OpenEvolve's duplicate check, which compares source text. Each fully evaluated candidate therefore carries a fingerprint, a hash of its 90 cut values on the training grid; equal fingerprints almost certainly mean the same algorithm. When a child's fingerprint matches a program on its island, only the one with the higher score stays, so a duplicate survives only if it improves on the resident, e.g. by running faster at the same cuts.

- **LLM ensemble.** Each iteration draws its model from an ensemble of two: Claude Opus 5 with weight 0.8 and Claude Fable 5.1 with weight 0.2. The run lasts 200 iterations with four candidates in flight at a time. The model sees only the prompt and answers only with diffs; the harness applies them to obtain the child program and compiles it.
- **Evaluation.** A candidate is run single-threaded on a fixed training grid and scored only on the partition files it writes, one part id per vertex, **never on what it reports about itself**.
  - **Training grid.** The grid must be small enough to keep evaluation fast, yet representative enough that improvements found on it generalize. It has ten instances, each at $k \in \{2, 3, 4\}$ with three seeds, i.e. 30 cells and 90 runs per candidate. Five come from ISPD98 (`ibm01`, `ibm05`, `ibm09`, `ibm13`, `ibm17`), evenly spaced from 13k to 185k vertices to cover a range of scales. Five come from Titan23, a suite with many large hyperedges (`neuron`, `cholesky_mc`, `minres`, `openCV`, `gsm_switch`), all of moderate size to keep evaluation fast and drawn from several application domains to represent the suite. The other 13 ISPD98 and 17 Titan23 instances never feed back into the evolution and are marked as held out in the appendix tables.
  - **Cross-tier reuse in the cascade.** As in OpenEvolve, a candidate passes three tiers, a single small cell, a five-cell grid with one seed, and the full grid, and moves on only if its score exceeds a threshold; every $(\text{instance}, k, \text{seed})$ result is cached, so a later tier runs only the cells not yet measured.
  - **Score.** Let $c^{\text{ref}}_{i,k}$ and $t^{\text{ref}}_{i,k}$ be the cut and wall time of KaHyPar (Schlag et al., 2023) on cell (i, k), medians over three seeds, and let $c_{i,k}$ and $t_{i,k}$ be the candidate's medians on the same cell. With τ the geometric mean of $t_{i,k}/t^{\text{ref}}_{i,k}$ over the N cells, the fitness is

$$\text{score} = \underbrace{\Big(\prod_{i,k} \frac{c^{\text{ref}}_{i,k}}{c_{i,k}}\Big)^{1/N}}_{\text{cut quality}} - \underbrace{\text{clamp}\big(0.02 \log_2 \tau,\ \pm 0.14\big)}_{\text{time term}} - \underbrace{\min(10^{-8}\text{chars},\ 0.003)}_{\text{code size}}. \quad (3)$$

    Cut quality is the main objective; running time and code size enter as small additive terms. Cut and time are dimensionless ratios to KaHyPar, time on a log scale, and the weight of 0.02 makes a $2\times$ slowdown cost as much as 2% of cut quality. A cell scores zero, which zeroes the whole geometric mean, if any seed fails to compile, crashes, exceeds the time limit, or writes an unreadable partition or one that violates equation 1.

- **Measurement feedback.** Two kinds of feedback reach the prompt that edits a program next. The first is written by the evaluator: the cut and wall time of every cell against the reference, plus a per-k summary of both. The second is written by the program itself: whatever a candidate prints during evaluation (via `std::cout` or `printf`), such as a per-phase timing breakdown or the strategy an internal portfolio selected, is passed on as debugging notes for its own descendants, so the evolution can develop its own diagnostics. Appendix D gives an example of each.

4 Algorithm Analysis

The evolved partitioner was written from an initial program that assigned vertices round-robin; every data structure and every phase below is the model's. This section describes what the final program does and what in it is new.

The program is a multilevel recursive-bisection partitioner of 1,186 lines in one file, with the hypergraph in CSR arrays. The rest of the program follows established practice in multilevel hypergraph partitioners (Karypis et al., 1997; Schlag et al., 2023; Heuer et al., 2019; Çatalyürek & Aykanat, 2011); the two components marked ❶ and ❷ are, to our knowledge, not part of existing hypergraph partitioners.

- **Coarsening.** Vertices are merged one at a time into the neighbouring cluster with the highest heavy-edge rating $\sum_e w(e)/(|e|-1)$, until about 200 vertices remain; a cluster may weigh at most $1.5\times$ the average weight of a coarsest vertex, and coarse nets that have become identical are merged.
  - *Rating by net-size class.* Nets are divided into three size classes for rating: up to 16 pins, 17–128 pins, and 129–400 pins; nets with more than 400 pins are ignored. A vertex scans the classes from smallest to largest and proceeds to a larger class only if rating budget remains, so small nets, which most clearly identify a partner, are always counted.
  - *Racing two coarsenings with opposite net-size priorities inside the restarts* ❶. When nets with more than 60 pins hold at least 12% of all pins, the restarts of every bisection alternate between two coarsenings with opposite priorities and keep the better bisection. Even restarts use the heavy-edge clustering above, which favours small nets. Odd restarts, at every level, first contract whole nets of 8 to 512 pins, largest first, each into one cluster if none of its pins is clustered yet and the cluster fits the weight cap, until half of the vertices are used; the heavy-edge clustering then handles the remaining vertices.
- **Initial partitioning.** On the coarsest graph, 3 to 30 partitions are grown greedily from random seed vertices, each polished by FM; the best is kept.
- **Refinement.**
  - *Two-way FM.* The program runs FM (Fiduccia & Mattheyses, 1988) for up to 6 passes per level, with candidate moves restricted to boundary vertices and rollback to the best prefix; nets of more than 10^4 pins are left out of the gain updates.
  - *Whole-net compound moves.* A cut net with many pins on both sides is a plateau for single-vertex FM; the program moves all pins of one side across at once when the cut decreases and the balance holds.
  - *Flow refinement.* As in KaHyPar-MF (Heuer et al., 2019), the program takes a small region around the cut at the finest level, builds a flow network on it, and finds a minimum cut with Dinic's algorithm. If the new cut violates the balance constraint, the region is halved and the step is repeated. Flow steps alternate with FM until the cut stops improving.
- **Restarts and V-cycles.** Each bisection is run from scratch 3 to 24 times, stopping once 12 (small graphs) or 5 (large graphs) consecutive restarts fail to improve or the budget is spent. The best bisection is then refined in three steps: flow refinement, up to 8 V-cycles (stopped after two consecutive failures), and flow refinement again.
- k **parts.**
  - *Recursive bisection.* A block that still has to be split reserves 30% of its balance slack for the bisections below it. For $k \geq 3$ the whole recursion is run twice with different seeds and the result with the lower k-way cut is kept.
  - *Re-partitioning pairs of final blocks* ❷. After the k blocks are formed, each pair of blocks is merged and bisected again, and the new bisection replaces the pair whenever it lowers the cut. One pass over all pairs is a *pair sweep*; up to three sweeps are run, as long as each still lowers the cut.
- **Instance-adaptive budget.** Before partitioning, the program estimates the expected cost of the instance from its number of pins, k, and the share of pins in nets with more than 60 pins, sets a wall-clock budget from that estimate, and decides how many restarts, V-cycles and pair sweeps each bisection gets.

The two new components are worth carrying into hand-written partitioners, because each answers a weakness of the multilevel scheme that current tools leave unaddressed. Racing two coarsenings with opposite net-size priorities ❶ targets large nets, which the heavy-edge rating barely sees and single-vertex FM cannot repair once they are split; rather than deciding per instance which nets to merge first, the restarts try both priorities and keep the better bisection. Re-partitioning pairs of

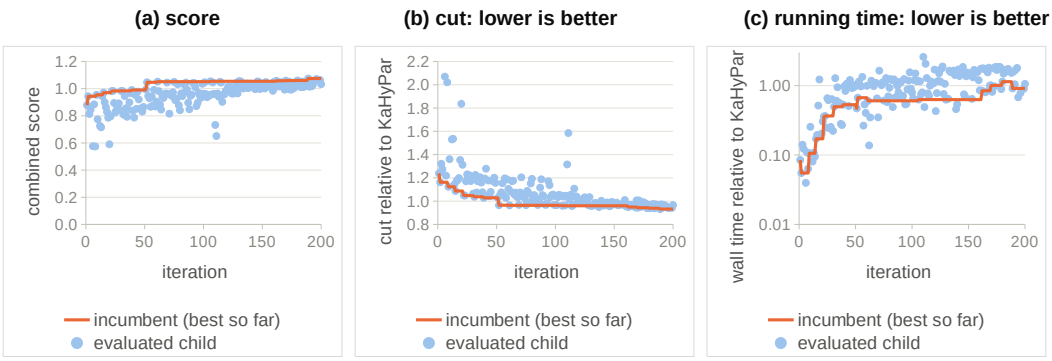

Figure 3: The evolution run. Each dot is one evaluated child, the line is the incumbent: (a) score, (b) cut relative to KaHyPar (the geometric mean of our cut over KaHyPar's on the training grid), (c) wall time relative to KaHyPar (log scale). Children that failed to compile or produced an infeasible partition score zero and are omitted.

Table 2: ISPD98 (18 instances, $k \in \{2, 3, 4\}$, 3 seeds): cut size and wall time normalized to KaHyPar ($= 1.00$), geometric means of per-instance ratios; lower is better. *Avg* is the mean over the three seeds, *Best* the best of the three, *Time* the mean wall time. The best of the three systems in each column is in bold.

System	$k = 2$			$k = 3$			$k = 4$		
	Avg	**Best**	**Time**	**Avg**	**Best**	**Time**	**Avg**	**Best**	**Time**
KaHyPar	1.000	1.000	1.00	1.000	**1.000**	**1.00**	1.000	1.000	**1.00**
hMETIS	0.994	1.007	2.33	1.005	1.020	1.95	0.993	1.005	1.85
EvoPart	**0.992**	**0.993**	**0.88**	**0.999**	1.008	2.93	**0.986**	**0.998**	2.94

Table 3: Titan23 (22 instances, $k \in \{2, 3, 4\}$, 3 seeds), in the format of Table 2; KaHyPar $= 1.00$, lower is better throughout.

System	$k = 2$			$k = 3$			$k = 4$		
	Avg	**Best**	**Time**	**Avg**	**Best**	**Time**	**Avg**	**Best**	**Time**
KaHyPar	1.000	1.000	1.00	1.000	1.000	1.00	1.000	1.000	1.00
hMETIS	1.005	1.013	0.38	0.967	1.007	**0.27**	1.015	1.023	**0.22**
EvoPart	**0.945**	**0.968**	**0.13**	**0.866**	**0.909**	0.48	**0.915**	**0.929**	0.45

final blocks ❷ can rearrange a split that the bisection tree committed to before its children existed, and thereby fine-tune the partition locally, one pair of blocks at a time. We see this as a promising direction, as it gives recursive bisection an extra way to revisit its early decisions.

5 Experiments

5.1 Setup

We evaluate on the two standard benchmarks, ISPD98 (Alpert, 1998) (18 instances) and Titan23 (Murray et al., 2013) (22 instances), whose instance sizes are listed in Appendix A, with $k \in \{2, 3, 4\}$, imbalance $\varepsilon = 0.02$, and three seeds per (instance, k) cell — 360 partitioner runs per system. We report the average and the best of the three seeds, and the average wall time. The baselines are hMETIS (Karypis et al., 1997) 2.0pre1, its latest release, and KaHyPar (Schlag et al., 2023) with its `cut_kKaHyPar_sea20.ini` configuration; both are run single-threaded, the same as ours.

5.2 Comparison with hMETIS and KaHyPar

The evolution run. The evolution ran for 200 iterations. Figure 3 shows the score, the cut and the wall time of every evaluated child against the iteration, together with the incumbent (the best program so far). The first child already turned the initial program into a working multilevel partitioner,

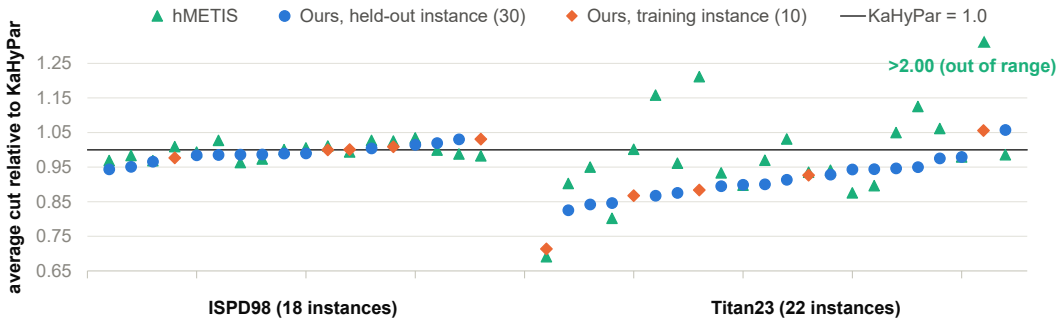

Figure 4: Average cut of EvoPart and hMETIS relative to KaHyPar on each of the 40 instances (geometric mean over $k \in \{2, 3, 4\}$, average of three seeds), sorted within each suite by EvoPart's ratio; values below 1 are better than KaHyPar. hMETIS's 2.00× on `gsm_switch` is off the scale.



Figure 5: Average cut over three seeds at $k \in \{5, 6, 7, 8\}$ relative to KaHyPar, on (a) ISPD98 and (b) Titan23. The evolution only ever evaluated $k \in \{2, 3, 4\}$.

with a cut of 1.24× KaHyPar's in 8% of KaHyPar's time. The highest score of the run was reached at iteration 189, with a cut of 0.93× KaHyPar's and a wall time of 0.91× KaHyPar's *on the training grid*. We take this program as EvoPart; it is selected on training-grid measurements only and is a single file of 1,186 lines of C++.

ISPD98. On ISPD98 (Table 2) the three systems are close: ours has the lowest average cut at every k (0.992, 0.999 and 0.986× KaHyPar's), by margins of under 1% at k=2 and k=3 and of 1.4% at k=4, and its best-of-three cut matches KaHyPar's. It is faster than both tools at k=2 (0.88× KaHyPar's time against 2.33× for hMETIS) and slower at k=3 and 4 (2.9× KaHyPar, 1.5–1.6× hMETIS), where the program spends its budget on two seeded runs of the whole recursion followed by re-partitioning the final blocks pair by pair (section 4).

Titan23. The picture on Titan23 (Table 3) is where the evolved program separates from the baselines. Its cut is 0.908× KaHyPar's over the suite and 0.866× at k=3, in 0.30× KaHyPar's wall time; hMETIS is as fast as ours on this suite (0.28×) but matches KaHyPar's cut only within 0.5%, so against hMETIS ours is 8.8% better in cut at the same time. The advantage on Titan23 comes largely from two of the evolved components (section 4): the race between the two coarsenings inside the restarts, which triggers only where large nets hold a real share of the pins, and the pair sweeps; the ablations of section 5.3 show that the race gains both cut and time on this suite and that the sweeps gain cut.

Whole benchmark. Over the whole benchmark the evolved partitioner's cut is 5.5% lower than KaHyPar's and 5.1% lower than hMETIS's, in 70% of KaHyPar's wall time and the same wall time as hMETIS. Per instance, it beats KaHyPar on 31 of the 40 instances and hMETIS on 27 (Figure 4). On the 30 instances never seen during evolution, its cut is 5.4% lower than KaHyPar's, nearly the same as on the whole benchmark.

Generalization to larger k. To test whether EvoPart generalizes beyond the values of k seen during evolution (2, 3 and 4), we also run the three systems at $k \in \{5, 6, 7, 8\}$ on the full benchmark, again

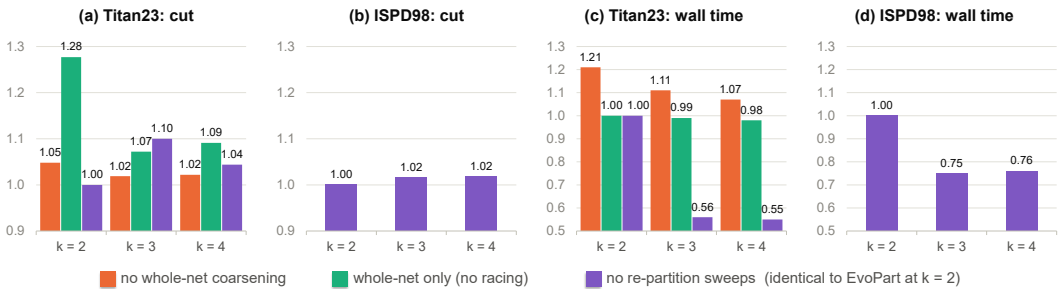

Figure 6: Component ablation, relative to EvoPart (= 1.00): average cut on (a) Titan23 and (b) ISPD98, wall time on (c) Titan23 and (d) ISPD98. The coarsening variants are measured on the 21 Titan23 instances that trigger whole-net coarsening.

with three seeds (Figure 5; best-of-three cuts in Appendix C). The picture of Tables 2 and 3 carries over unchanged. On ISPD98 EvoPart's average cut stays within 1% of KaHyPar's at every k, lower at $k = 5, 6, 8$ and slightly higher at $k = 7$, and below hMETIS's throughout. On Titan23 it is 9–11% below KaHyPar's and 6–11% below hMETIS's, whereas hMETIS and KaHyPar are within 4% of each other. EvoPart thus generalizes well across k.

5.3 Ablations

Harness. To test the behavioral MAP-Elites dimensions, we rerun the evolution three times with OpenEvolve's default textual descriptors, complexity and diversity, keeping everything else unchanged, and select from each run the program with the highest score, as for EvoPart. Table 4 evaluates the four programs on the full benchmark. All three textual runs fall short of EvoPart on both suites: their average cut is 1–3% higher on ISPD98, where none of them beats KaHyPar, and 3–6% higher on Titan23, at wall times of 0.65–1.15× KaHyPar's against EvoPart's 0.70×. Although EvoPart comes from a single run, it leads all three textual runs on both ISPD98 and Titan23.

Table 4: Harness ablation: the best-score program of each run, relative to KaHyPar as in Table 2; lower is better. Textual: OpenEvolve's default descriptors.

Run	ISPD98		Titan23		Time
	Avg	Best	Avg	Best	
Textual 1	1.002	1.010	0.935	0.951	0.81
Textual 2	1.023	1.025	0.959	0.970	**0.65**
Textual 3	1.003	1.013	0.948	0.965	1.15
EvoPart	**0.992**	**1.000**	**0.908**	**0.935**	0.70

Components. Figure 6 switches off each new component with a one-line change: *no whole-net coarsening* makes every restart use only the pairwise heavy-edge clustering with rating $\sum_e w(e)/(|e|-1)$, *whole-net coarsening only* makes every restart first contract large nets whole, and *no re-partition sweeps* removes the re-bisection of pairs of final blocks, which acts only at $k \geq 3$. Racing the two coarsenings gives a lower cut than either alone, which raises it by 3% (heavy-edge clustering only) or 14% (whole-net coarsening only), and is no slower than either. It acts only on Titan23, and leaves ISPD98 unchanged. The re-partition sweeps trade time for cut: at $k \in \{3, 4\}$ they multiply the running time by 1.8 on Titan23 and 1.3 on ISPD98, and lower the cut by 4–9% and 1.5–1.9%, respectively. Under the score of equation 3, this is a profitable trade of time for cut quality.

6 Conclusion

We have used an LLM coding agent built on OpenEvolve, with a harness tailored to evolving a whole solver, to evolve EvoPart, an open, lightweight and strong hypergraph partitioner. Its cut is better than or on par with that of hMETIS and KaHyPar at $k \in \{2, 3, 4\}$, the values it was evolved on, and also at the unseen $k \in \{5, \ldots, 8\}$, in about 70% of KaHyPar's wall time and with 22.5× less code. Two of its components, racing two coarsenings with opposite net-size priorities inside the restarts and re-partitioning pairs of final blocks, are to our knowledge new and worth carrying into hand-written partitioners. We release EvoPart and the harness as a new starting point for open hypergraph partitioning.

AI USE STATEMENT

In this work, we used generative AI tools to assist with translation and method implementation. We have not used generative AI tools to develop theoretical models or conceptual frameworks, to propose or refine hypotheses, to design or provide feedback on our research methodology or experiments, to clean and reformat datasets, or to interpret results; the remaining required disclosure tasks—generating synthetic data sets, formulating mathematical claims, providing critical ingredients for proving mathematical claims, assisting in the writing of proofs, and supporting qualitative and thematic data analysis—are not applicable to this work. Additionally, we used generative AI tools to identify relevant literature, to edit the manuscript for readability, and to format tables. We have reviewed all AI-assisted work: all AI-assisted code was read, tested, and verified for correctness by the authors, and every reported number was produced by running the final code on the benchmarks described in the paper. We note that LLM-generated code is also an intrinsic component of our proposed method rather than an authoring aid (our framework performs evolutionary search directly over program code, with an LLM generating candidate programs); this is described in detail in the main text. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

To ensure the reproducibility of our findings, we release both our evolutionary search framework and the final evolved hypergraph partitioner as source code at the following anonymous URL: https://anonymous.4open.science/r/EvoPart-4F6C. We believe this allows our reported results to be verified and provides a useful resource for future research on hypergraph partitioning algorithms.

REFERENCES

Charles J Alpert. The ispd98 circuit benchmark suite. In *Proceedings of the 1998 international symposium on Physical design*, pp. 80–85, 1998.

Henrique Assumpção, Diego Ferreira, Leandro Campos, and Fabricio Murai. Codeevolve: an open source evolutionary coding agent for algorithmic discovery and optimization. *arXiv preprint arXiv:2510.14150*, 2025.

Ismail Bustany, Andrew B Kahng, Ioannis Koutis, Bodhisatta Pramanik, and Zhiang Wang. Specpart: A supervised spectral framework for hypergraph partitioning solution improvement. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, pp. 1–9, 2022.

Ismail Bustany, Andrew B Kahng, Ioannis Koutis, Bodhisatta Pramanik, and Zhiang Wang. K-specpart: Supervised embedding algorithms and cut overlay for improved hypergraph partitioning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 43(4): 1232–1245, 2023.

Ümit V Çatalyürek and Cevdet Aykanat. Patoh (partitioning tool for hypergraphs)., 2011.

Pham Vu Tuan Dat, Long Doan, and Huynh Thi Thanh Binh. Hsevo: Elevating automatic heuristic design with diversity-driven harmony search and genetic algorithm using llms. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pp. 26931–26938, 2025.

Gunduz Vehbi Demirci, Aparajita Haldar, and Hakan Ferhatosmanoglu. Scalable graph convolutional network training on distributed-memory systems. *arXiv preprint arXiv:2212.05009*, 2022.

Charles M Fiduccia and Robert M Mattheyses. A linear-time heuristic for improving network partitions. In *Papers on Twenty-five years of electronic design automation*, pp. 241–247. 1988.

Seokjin Go and Divya Mahajan. Moetuner: Optimized mixture of expert serving with balanced expert placement and token routing. *arXiv preprint arXiv:2502.06643*, 2025.

Tobias Heuer, Peter Sanders, and Sebastian Schlag. Network flow-based refinement for multilevel hypergraph partitioning. *Journal of Experimental Algorithmics (JEA)*, 24:1–36, 2019.

Qinglong Hu and Qingfu Zhang. Partition to evolve: Niching-enhanced evolution with llms for automated algorithm discovery. *Advances in Neural Information Processing Systems*, 38:154771–154817, 2026.

George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392, 1998.

George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. Multilevel hypergraph partitioning: Application in vlsi domain. In *Proceedings of the 34th annual Design Automation Conference*, pp. 526–529, 1997.

Robert Lange, Yuki Imajuku, and Edoardo Cetin. Shinkaevolve: Towards open-ended and sample-efficient program evolution. In *International Conference on Learning Representations*, volume 2026, pp. 74026–74078, 2026.

Fei Liu, Xialiang Tong, Mingxuan Yuan, Xi Lin, Fu Luo, Zhenkun Wang, Zhichao Lu, and Qingfu Zhang. Evolution of heuristics: Towards efficient automatic algorithm design using large language model. *arXiv preprint arXiv:2401.02051*, 2024.

Jean-Baptiste Mouret and Jeff Clune. Illuminating search spaces by mapping elites. *arXiv preprint arXiv:1504.04909*, 2015.

Kevin E Murray, Scott Whitty, Suya Liu, Jason Luu, and Vaughn Betz. Titan: Enabling large and complex benchmarks in academic cad. In *2013 23rd International Conference on Field programmable Logic and Applications*, pp. 1–8. IEEE, 2013.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco JR Ruiz, Abbas Mehrabian, et al. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M Pawan Kumar, Emilien Dupont, Francisco JR Ruiz, Jordan S Ellenberg, Pengming Wang, Omar Fawzi, et al. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024.

Hamed Sajadinia, Ali Aghdaei, and Zhuo Feng. Shypar: A spectral coarsening approach to hypergraph partitioning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 45(2):673–685, 2025.

Sebastian Schlag, Tobias Heuer, Lars Gottesbüren, Yaroslav Akhremtsev, Christian Schulz, and Peter Sanders. High-quality hypergraph partitioning. *ACM Journal of Experimental Algorithmics*, 27:1–39, 2023.

Asankhaya Sharma. Openevolve: an open-source evolutionary coding agent, 2025. URL https://github.com/algorithmicsuperintelligence/openevolve.

Niki Van Stein and Thomas Bäck. Llamea: A large language model evolutionary algorithm for automatically generating metaheuristics. *IEEE Transactions on Evolutionary Computation*, 29 (2):331–345, 2024.

Yang Wu, Junran Pan, Yifan Zhang, Ning Xu, Fanshuo Zeng, and Jian Cheng. Refineevo: Planning-guided heuristic evolution with bidirectional experience. *arXiv preprint arXiv:2607.11358*, 2026.

Haoran Ye, Jiarui Wang, Zhiguang Cao, Federico Berto, Chuanbo Hua, Haeyeon Kim, Jinkyoo Park, and Guojie Song. Reevo: Large language models as hyper-heuristics with reflective evolution. *Advances in neural information processing systems*, 37:43571–43608, 2024.

Da Zheng, Chao Ma, Minjie Wang, Jinjing Zhou, Qidong Su, Xiang Song, Quan Gan, Zheng Zhang, and George Karypis. Distdgl: Distributed graph neural network training for billion-scale graphs. In *2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, pp. 36–44. IEEE, 2020.

Zhi Zheng, Zhuoliang Xie, Zhenkun Wang, and Bryan Hooi. Monte carlo tree search for comprehensive exploration in llm-based automatic heuristic design. *arXiv preprint arXiv:2501.08603*, 2025.

Yugao Zhu, Wenji Fang, Yao Lu, Shang Liu, Yanzhen Zhu, Qijun Zhang, and Zhiyao Xie. Mfspart: A generalized partitioning framework for multi-fpga systems and its ensemble-based extension. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2026a.

Yugao Zhu, Zhicheng Guo, Shang Liu, Mengming Li, Jing Wang, and Zhiyao Xie. Impart: Integration of memetic operations into multi-level framework for large-k-way hypergraph partitioning. *arXiv preprint arXiv:2606.18117*, 2026b.

A Benchmark instances

Table 5 lists the size of every instance in the two suites, as read from the `.hgr` files we partition: ISPD98 (Alpert, 1998) spans 13k to 211k vertices and Titan23 (Murray et al., 2013) 92k to 1.09M.

Table 5: The benchmark instances: number of vertices, nets and pins of every `.hgr` file used, each suite ordered by size. Instances marked † appear in the training grid.

ISPD98				Titan23			
Instance	**Vertices**	**Nets**	**Pins**	**Instance**	**Vertices**	**Nets**	**Pins**
ibm01†	12,752	14,111	50,566	sparcT1_core	91,976	92,827	503,922
ibm02	19,601	19,584	81,199	neuron†	92,290	125,305	453,181
ibm03	23,136	27,401	93,573	stereo_vision	94,050	127,085	414,259
ibm04	27,507	31,970	105,859	des90	111,221	139,557	771,009
ibm05†	29,347	28,446	126,308	SLAM_spheric	113,115	142,408	637,880
ibm06	32,498	34,826	128,182	cholesky_mc†	113,250	144,948	622,678
ibm07	45,926	48,117	175,639	segmentation	138,295	179,051	801,427
ibm08	51,309	50,513	204,890	bitonic_mesh	192,064	235,328	1,364,768
ibm09†	53,395	60,902	222,088	dart	202,354	223,301	1,133,096
ibm10	69,429	75,196	297,567	openCV†	217,453	284,108	1,273,473
ibm11	70,558	81,454	280,786	stap_qrd	240,240	290,123	1,260,458
ibm12	71,076	77,240	317,760	minres†	261,359	320,540	1,309,022
ibm13†	84,199	99,666	357,075	cholesky_bdti	266,422	342,688	1,410,151
ibm14	147,605	152,772	546,816	denoise	275,638	356,848	1,610,717
ibm15	161,570	186,608	715,823	sparcT2_core	300,109	302,663	1,615,849
ibm16	183,484	190,048	778,823	gsm_switch†	493,260	507,821	2,712,782
ibm17†	185,495	189,581	860,036	mes_noc	547,544	577,664	3,187,553
ibm18	210,613	201,920	819,697	LU230	574,372	669,477	2,996,579
				LU_Network	635,456	726,999	3,331,887
				sparcT1_chip2	820,886	821,274	4,031,921
				directrf	931,275	1,374,742	4,812,506
				bitcoin_miner	1,089,284	1,448,151	4,557,197

B Full per-instance results

This appendix gives the raw per-instance measurements behind Tables 2 and 3. Every system is run single-threaded on the same machine, with $k \in \{2, 3, 4\}$, $\varepsilon = 0.02$ and seeds 1–3. Column headings abbreviate the three systems: **hM** is hMETIS (Karypis et al., 1997) 2.0pre1, **KaHy** is KaHyPar (Schlag et al., 2023) with its `cut_kKaHyPar_sea20.ini` configuration, and **EvoPart** is the evolved partitioner of section 5.2. In every cell the lowest of the three values is set in bold. Instances are listed in the order of Table 5, i.e. by size within each suite.

Instances marked † belong to the training grid, five in ISPD98 and five in Titan23. The remaining 13 ISPD98 and 17 Titan23 instances never produced feedback for the evolution, so the unmarked rows can be read as a pure holdout.

Cuts are cut-net counts (lower is better). Tables 6 and 9 report the mean over the three seeds, Tables 7 and 10 the best of the three, and Tables 8 and 11 the mean wall time in seconds.

Table 6: ISPD98 — average cut over three seeds.

Instance	$k = 2$			$k = 3$			$k = 4$		
	hM	**KaHy**	**EvoPart**	**hM**	**KaHy**	**EvoPart**	**hM**	**KaHy**	**EvoPart**
ibm01†	206	**203**	224	367	364	**362**	**509**	549	550
ibm02	362	**351**	353	369	369	**365**	699	792	**669**
ibm03	**962**	986	**962**	1527	**1508**	1640	1740	1723	**1721**
ibm04	599	605	**594**	1251	1230	**1229**	1686	1667	**1651**
ibm05†	1731	**1728**	1732	2735	**2691**	2692	3065	3017	**3008**
ibm06	**1003**	1008	1004	1312	1309	**1301**	1537	1687	**1535**
ibm07	986	988	**967**	1951	1873	**1808**	2289	**2193**	2221
ibm08	1159	1145	**1141**	**2015**	2099	2029	2436	2414	**2388**
ibm09†	**628**	637	634	1530	**1442**	1446	1729	**1678**	1724
ibm10	1345	1369	**1308**	2032	2051	**1904**	2539	2594	**2515**
ibm11	**1073**	1193	1119	**1833**	1848	1904	2559	2553	**2535**
ibm12	2102	**2066**	2071	3033	**2935**	2936	4104	**3977**	4016
ibm13†	864	**850**	860	1449	**1397**	1411	**1916**	2054	2015
ibm14	1923	1956	**1857**	**2840**	2846	2841	3440	**3365**	3441
ibm15	**2840**	2841	2936	**4269**	4466	4596	4984	**4935**	5079
ibm16	**2062**	2074	2149	3242	3520	**3187**	4328	**4291**	4397
ibm17†	2433	**2390**	2427	4519	**4426**	4440	6050	6115	**5597**
ibm18	1989	2032	**1832**	2848	**2742**	3015	3348	**3075**	3240

Table 7: ISPD98 — best cut of three seeds.

Instance	$k = 2$			$k = 3$			$k = 4$		
	hM	**KaHy**	**EvoPart**	**hM**	**KaHy**	**EvoPart**	**hM**	**KaHy**	**EvoPart**
ibm01†	206	**203**	**203**	364	**359**	361	**506**	545	532
ibm02	357	**350**	351	364	**362**	365	650	694	**637**
ibm03	960	960	**957**	1516	**1469**	1604	1733	1715	**1713**
ibm04	598	596	**593**	1248	1225	**1208**	1675	1664	**1632**
ibm05†	1728	1727	**1725**	2726	**2649**	2651	3040	3008	**2989**
ibm06	990	986	**983**	1299	**1298**	1300	1525	**1490**	1493
ibm07	976	970	**946**	1930	1865	**1776**	2279	**2168**	2199
ibm08	1150	**1140**	**1140**	2010	2088	**1987**	2427	2395	**2362**
ibm09†	627	**625**	634	1516	1439	**1402**	1722	**1676**	1707
ibm10	1326	1369	**1295**	1984	1929	**1891**	**2473**	2506	2491
ibm11	1068	**1066**	1070	1809	**1788**	1834	2548	2521	**2487**
ibm12	2078	**1966**	2071	2971	**2814**	2924	4022	**3925**	3965
ibm13†	853	**848**	**848**	1415	**1346**	1360	**1882**	1998	1989
ibm14	1920	1950	**1850**	2823	**2765**	2777	3397	**3238**	3421
ibm15	2836	2822	**2743**	**4088**	4363	4406	4949	**4859**	5056
ibm16	**2020**	2025	2060	3223	3224	**3132**	4273	**4225**	4321
ibm17†	2421	**2341**	2357	4400	**4266**	4403	5953	6102	**5588**
ibm18	1958	1916	**1825**	2721	**2502**	2761	3272	**3026**	3219

Table 8: ISPD98 — average wall time over three seeds, in seconds, single-threaded.

Instance	$k = 2$			$k = 3$			$k = 4$		
	hM	**KaHy**	**EvoPart**	**hM**	**KaHy**	**EvoPart**	**hM**	**KaHy**	**EvoPart**
ibm01†	1.18	**0.66**	1.05	1.14	**0.58**	3.10	2.07	**1.10**	3.31
ibm02	2.57	**0.88**	1.18	3.37	**1.45**	4.48	3.28	**1.58**	5.64
ibm03	2.22	**0.96**	1.19	2.92	**1.69**	4.59	**2.57**	2.67	5.68
ibm04	2.83	**1.00**	1.17	3.03	**2.07**	6.34	3.96	**1.71**	5.90
ibm05†	5.16	**2.07**	2.07	6.19	**2.49**	6.12	7.31	**3.44**	9.02
ibm06	3.83	1.72	**1.52**	3.88	**1.84**	7.27	4.53	**2.62**	7.66
ibm07	3.62	2.76	**2.20**	5.33	**4.00**	8.90	5.16	**2.93**	10.18
ibm08	5.59	2.28	**2.08**	6.26	**4.71**	10.88	7.27	**3.89**	11.08
ibm09†	4.43	2.08	**1.80**	4.32	**2.89**	8.95	5.84	**2.87**	11.19
ibm10	7.46	**2.52**	3.02	8.43	**3.45**	11.10	9.05	**5.64**	16.21
ibm11	5.64	4.26	**2.39**	5.45	**3.10**	10.91	7.54	**4.25**	15.46
ibm12	9.11	**2.93**	3.33	9.25	**5.01**	13.29	10.60	**5.59**	20.09
ibm13†	7.26	2.99	**2.82**	8.67	**3.75**	13.29	9.62	**5.22**	15.62
ibm14	15.98	6.29	**4.94**	15.16	**7.49**	25.13	18.67	**13.83**	26.74
ibm15	21.32	8.80	**4.95**	19.48	**11.60**	20.58	23.55	**10.73**	30.36
ibm16	21.56	10.84	**4.93**	25.09	**10.11**	27.11	23.64	**13.87**	31.90
ibm17†	29.49	10.05	**5.74**	32.18	**12.10**	34.60	34.67	**15.39**	37.45
ibm18	25.49	8.41	**5.43**	27.86	**11.12**	28.02	30.39	**11.45**	37.87

Table 9: Titan23 — average cut over three seeds.

Instance	$k = 2$			$k = 3$			$k = 4$		
	hM	**KaHy**	**EvoPart**	**hM**	**KaHy**	**EvoPart**	**hM**	**KaHy**	**EvoPart**
sparcT1_core	1034	1305	**988**	2353	2156	**1993**	2737	2760	**2356**
neuron†	361	279	**263**	524	473	**402**	603	487	**419**
stereo_vision	178	**169**	175	**350**	418	354	467	494	**452**
des90	**380**	466	500	562	649	**537**	789	829	**785**
SLAM_spheric	**1061**	**1061**	**1061**	3073	3258	**2990**	**3480**	3491	3576
cholesky_mc†	357	392	**310**	1046	987	**827**	1148	1103	**1086**
segmentation	164	182	**143**	543	576	**520**	620	592	**559**
bitonic_mesh	598	610	**587**	930	1148	**928**	**1181**	1297	1210
dart	828	850	**799**	1318	1409	**1180**	1490	1489	**1381**
openCV†	**474**	743	579	585	891	**580**	674	856	**613**
stap_qrd	421	463	**407**	**538**	601	554	838	838	**742**
minres†	**246**	249	250	395	398	**387**	467	559	**457**
cholesky_bdti	1236	1311	**1163**	1997	2390	**1755**	2037	2181	**1885**
denoise	554	493	**457**	860	856	**822**	1130	1164	**998**
sparcT2_core	1213	**1189**	1229	3131	2647	**2225**	3591	3039	**3004**
gsm_switch†	4959	**1732**	**1732**	5246	**2720**	3365	5465	3753	**3570**
mes_noc	723	**701**	909	**1202**	1403	1220	1566	**1443**	1514
LU230	**3533**	3597	3599	**4781**	6148	4995	**5882**	6235	6462
LU_Network	**654**	703	679	1178	1026	**1009**	1885	1738	**1554**
sparcT1_chip2	1218	1120	**1012**	1952	1688	**1404**	2337	1891	**1644**
directrf	**586**	664	665	749	1079	**729**	**1127**	1339	1200
bitcoin_miner	1580	1570	**1539**	2666	2395	**2322**	3320	3108	**3036**

Table 10: Titan23 — best cut of three seeds.

Instance	$k = 2$			$k = 3$			$k = 4$		
	hM	**KaHy**	**EvoPart**	**hM**	**KaHy**	**EvoPart**	**hM**	**KaHy**	**EvoPart**
sparcT1_core	989	1221	**984**	2296	1965	**1822**	2676	2676	**2304**
neuron†	315	245	**243**	505	457	**396**	570	464	**416**
stereo_vision	175	**169**	171	**343**	394	353	466	484	**440**
des90	**380**	435	485	**514**	575	534	758	778	**752**
SLAM_spheric	**1061**	**1061**	**1061**	**2905**	3111	2981	3441	**3353**	3475
cholesky_mc†	317	371	**306**	964	950	**823**	1106	**1080**	1082
segmentation	144	144	**136**	524	540	**478**	572	556	**540**
bitonic_mesh	586	594	**585**	928	1022	**925**	1153	1164	**1094**
dart	820	**784**	798	1273	1336	**1166**	1466	1419	**1361**
openCV†	**463**	689	544	568	798	**546**	652	814	**586**
stap_qrd	399	456	**383**	**534**	545	539	825	816	**716**
minres†	243	**238**	239	382	379	**373**	450	533	**449**
cholesky_bdti	1176	**1163**	**1163**	1913	2344	**1728**	1947	2068	**1876**
denoise	529	460	**456**	828	820	**797**	1019	1145	**908**
sparcT2_core	1189	**1187**	1196	2887	2526	**2187**	3329	2985	**2967**
gsm_switch†	4944	**1691**	1715	5225	**2683**	3247	5422	**3510**	3536
mes_noc	**698**	**698**	840	1179	1264	**1154**	1547	**1377**	1424
LU230	3493	3580	**3399**	**4751**	5854	4770	**5696**	5864	6317
LU_Network	641	**612**	652	1109	**932**	961	1777	1667	**1526**
sparcT1_chip2	1000	1083	**953**	1843	1582	**1382**	2107	1726	**1613**
directrf	**573**	663	662	737	780	**728**	**1083**	1254	1120
bitcoin_miner	**1514**	1568	1521	2636	2314	**2310**	3204	3007	**2949**

Table 11: Titan23 — average wall time over three seeds, in seconds, single-threaded.

Instance	$k = 2$			$k = 3$			$k = 4$		
	hM	**KaHy**	**EvoPart**	**hM**	**KaHy**	**EvoPart**	**hM**	**KaHy**	**EvoPart**
sparcT1_core	10.43	137.87	**7.31**	**11.14**	195.57	30.00	**12.55**	266.49	35.83
neuron†	5.46	112.50	**1.97**	**6.14**	155.29	10.96	**6.95**	222.52	13.03
stereo_vision	5.48	2.99	**2.26**	5.47	**4.82**	12.33	5.77	**5.21**	17.28
des90	14.44	131.51	**4.79**	**12.68**	185.67	28.20	**13.02**	221.25	34.81
SLAM_spheric	13.40	26.00	**3.63**	18.50	43.15	**17.77**	**18.51**	72.82	21.30
cholesky_mc†	8.72	8.02	**3.06**	**11.28**	13.31	23.24	**13.02**	14.54	26.51
segmentation	12.72	14.71	**4.67**	**12.17**	18.24	25.03	**12.78**	24.15	27.73
bitonic_mesh	21.27	76.09	**7.14**	**23.48**	110.71	37.60	**25.85**	145.40	53.77
dart	14.67	89.85	**7.31**	**16.74**	137.99	48.40	**18.12**	178.85	54.54
openCV†	12.27	76.23	**7.96**	**13.89**	113.67	40.69	**14.95**	149.28	47.21
stap_qrd	22.90	17.41	**5.66**	**22.77**	22.82	28.87	**21.46**	28.77	49.18
minres†	18.45	19.31	**8.44**	**20.52**	25.87	36.16	**23.57**	32.42	46.30
cholesky_bdti	24.98	41.52	**7.23**	**26.49**	81.75	49.56	**34.14**	83.28	50.97
denoise	30.46	40.70	**12.89**	**32.06**	53.82	47.06	**29.43**	71.05	54.63
sparcT2_core	40.10	32.59	**13.47**	**48.39**	67.07	85.71	**49.16**	81.95	110.63
gsm_switch†	33.45	308.52	**17.78**	**36.27**	442.90	99.95	**42.41**	697.39	123.19
mes_noc	61.20	132.57	**26.79**	**63.65**	184.74	136.87	**64.45**	228.82	149.08
LU230	60.64	854.53	**23.91**	**61.03**	1092.66	126.99	**65.42**	1670.78	132.59
LU_Network	73.14	160.06	**13.24**	**71.74**	211.14	110.93	**90.15**	378.60	142.01
sparcT1_chip2	104.43	308.14	**29.60**	**97.46**	480.64	181.58	**126.37**	1363.57	211.19
directrf	158.40	130.96	**28.34**	148.15	132.11	**130.28**	**160.91**	175.19	182.61
bitcoin_miner	268.89	627.63	**24.36**	237.56	996.26	**148.93**	**180.11**	641.71	197.17

C FULL PER-INSTANCE RESULTS AT $k \in \{5, 6, 7, 8\}$

Figure 7 repeats Figure 5 for the best cut of the three seeds. The ordering of the systems is the same as for the average cut: on ISPD98 the three are within 2% of one another, and on Titan23 EvoPart is 8–10% below KaHyPar and 7–11% below hMETIS.



Figure 7: Best cut of three seeds at $k \in \{5, 6, 7, 8\}$, geometric means of per-instance ratios normalized to KaHyPar ($= 1.00$; lower is better), on (a) ISPD98 and (b) Titan23.

D HOW MEASUREMENTS SHAPED THE DESIGN

This appendix gives two examples of how the measurement feedback of the evolution loop shapes a child, one for each kind of feedback: the measurements the evaluator reports, such as the per-k summary, and the program's own stdout.

- **Iteration 132: per-k summary.** The summary showed k=3 as the weakest column, with `gsm_switch` the worst cell at a cut quality of 0.71. The model noticed that for k=3 the top-level split is 1:2 and that the side holding a single block was reserving balance slack it would never use, and stopped reserving it.
- **Iteration 189: the program's own stdout.** The timing line printed by the program showed coarsening taking 61% of the wall time and 144 restarts over six bisections, i.e. every bisection used the full cap of 24. The model added a rule that stops restarts after several consecutive failures. From the history of earlier programs it also saw that its parent lacked the flow refinement and the repeated pair sweeps of a better earlier program, a difference that "cost 2.5% of cut quality and bought only 0.016 of time term — a clearly losing trade at the stated exchange rate," and added both.