2D Partitioning

Aydin Buluc and Kamesh Madduri

Graph Partitioning for Scalable Distributed Graph Computations

Aydın Buluç

ABuluc@lbl.gov

Kamesh Madduri

madduri@cse.psu.edu

10th DIMACS Implementation Challenge, Graph Partitioning and Graph Clustering
February 13-14, 2012
Atlanta, GA

Overview of our study

- We assess the impact of graph partitioning for computations on ‘low diameter’ graphs
- Does minimizing edge cut lead to lower execution time?
- We choose parallel Breadth-First Search as a representative distributed graph computation
- Performance analysis on DIMACS Challenge instances

Key Observations for Parallel BFS

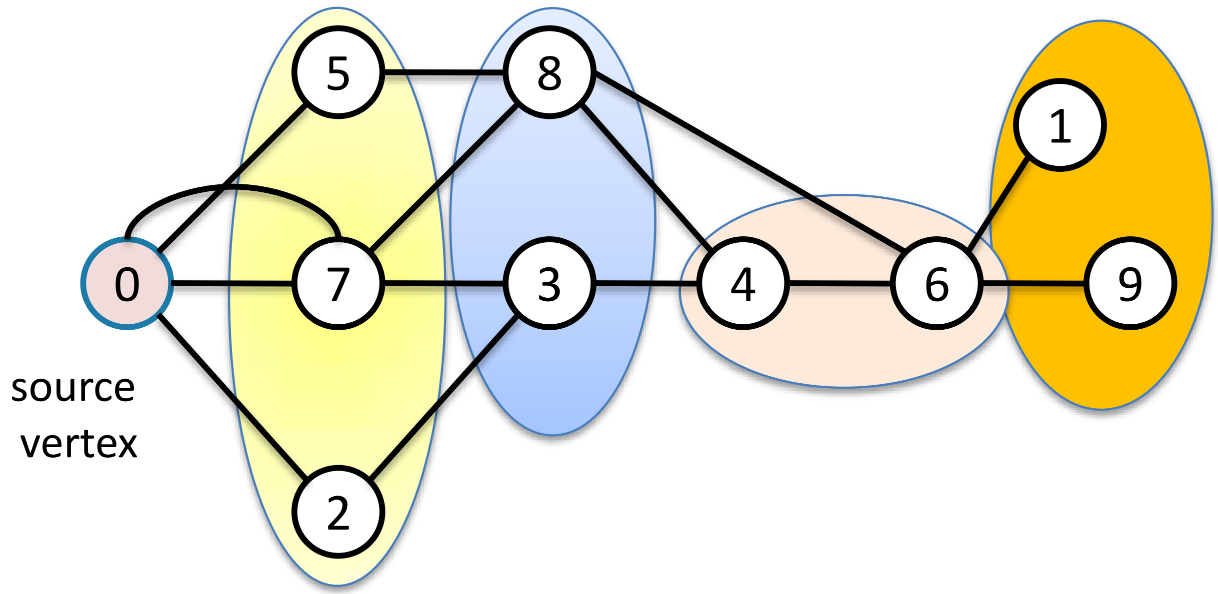
- Well-balanced vertex and edge partitions do not guarantee load-balanced execution, particularly for real-world graphs
  - Range of relative speedups (8.8-50X, 256-way parallel concurrency) for low-diameter DIMACS graph instances.
- Graph partitioning methods reduce overall edge cut and communication volume, but lead to increased computational load imbalance
- Inter-node communication time is not the dominant cost in our tuned bulk-synchronous parallel BFS implementation

Talk Outline

- Level-synchronous parallel BFS on distributed-memory systems
  - Analysis of communication costs
- Machine-independent counts for inter-node communication cost
- Parallel BFS performance results for several large-scale DIMACS graph instances

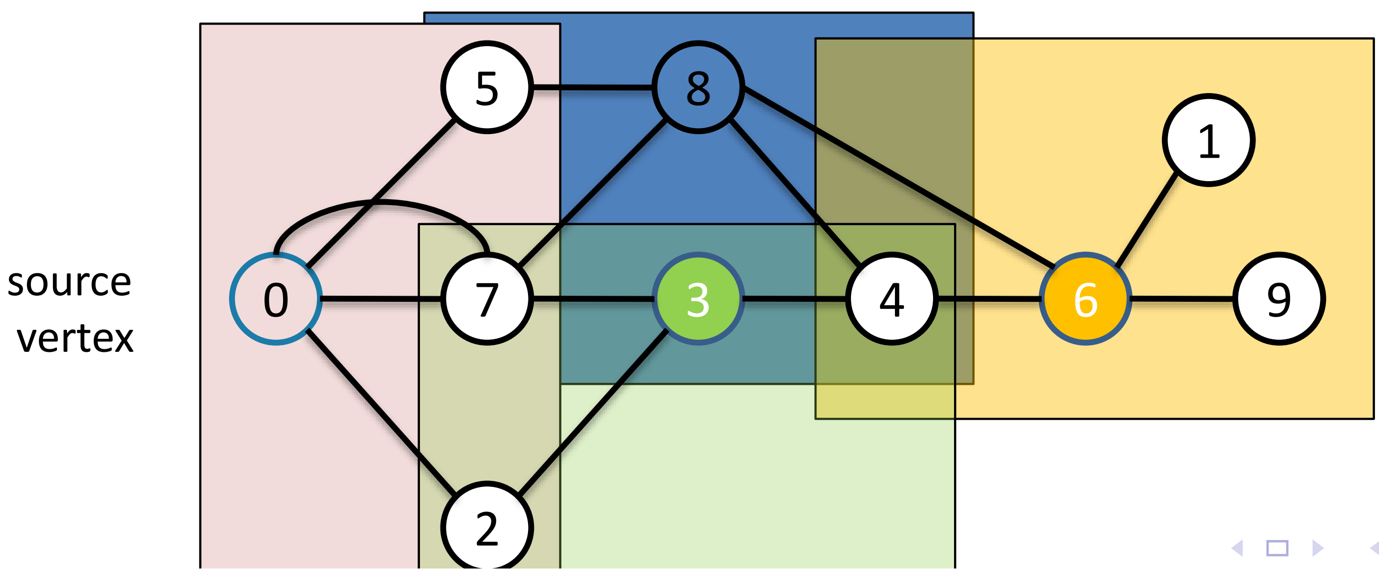
Parallel BFS strategies

1. Expand current frontier (**level-synchronous** approach, suited for **low diameter** graphs)



- O(D) parallel steps
- Adjacencies of all vertices in current frontier are visited in parallel

2. Stitch multiple concurrent traversals (Ullman-Yannakakis, for **high-diameter** graphs)

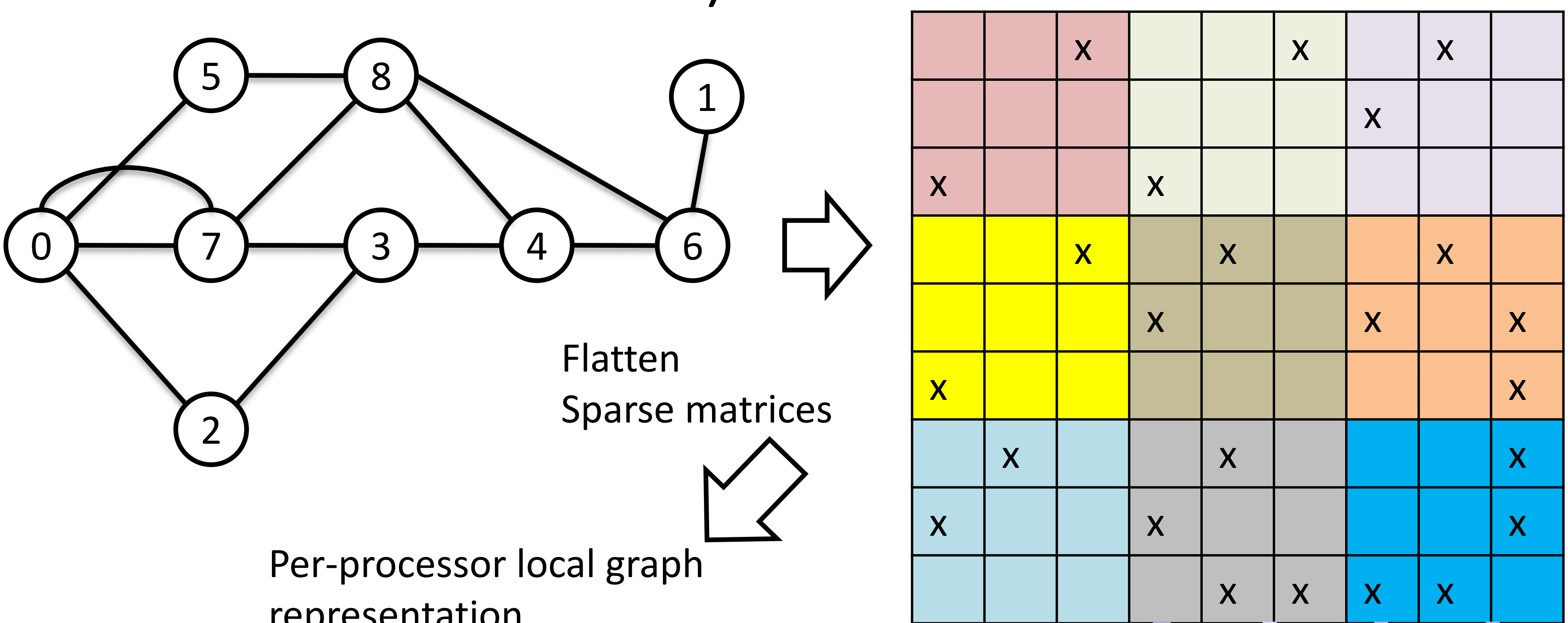


- path-limited searches from “super vertices”
- APSP between “super vertices”

“2D” graph distribution

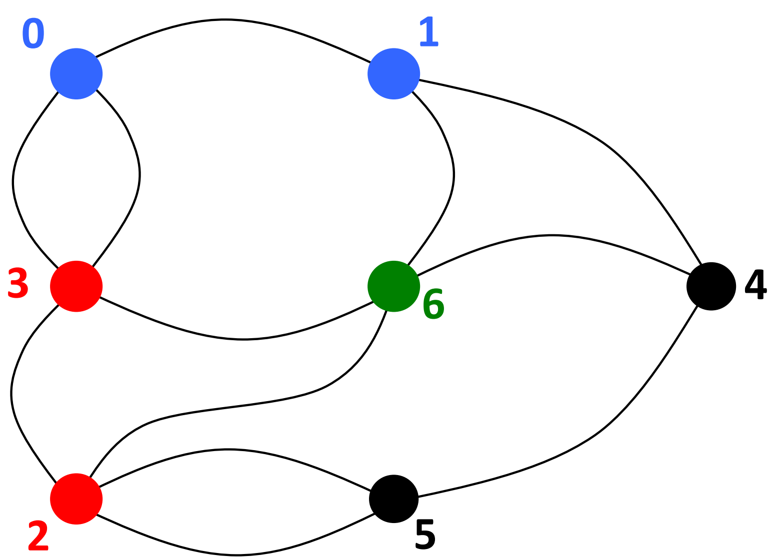
- Consider a logical 2D processor grid ($p_r * p_c = p$) and the dense matrix representation of the graph
- Assign each processor a sub-matrix (i.e, the edges within the sub-matrix)

9 vertices, 9 processors, 3x3 processor grid



BFS with a 1D-partitioned graph

Consider an undirected graph with **n** vertices and **m** edges



Each processor 'owns' **n/p** vertices and stores their adjacencies (~ **2m/p** per processor, assuming balanced partitions).

[0,1] [0,3] [0,3] [1,0] [1,4] [1,6]

[2,3] [2,5] [2,5] [2,6] [3,0] [3,0] [3,2] [3,6]

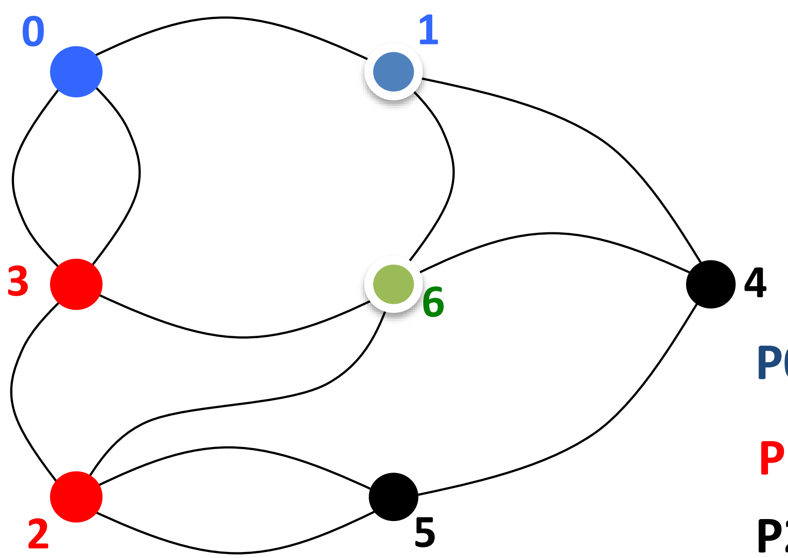
[4,1] [4,5] [4,6] [5,2] [5,2] [5,4]

[6,1] [6,2] [6,3] [6,4]

Steps:

1. Local discovery: Explore adjacencies of vertices in current frontier.
2. Fold: All-to-all exchange of adjacencies.
3. Local update: Update distances/parents for unvisited vertices.

BFS with a 1D-partitioned graph



Current frontier: vertices 1 (partition **Blue**) and 6 (partition **Green**)

1. Local discovery:

P0 **[1,0]** **[1,4]** **[1,6]**

P1 No work

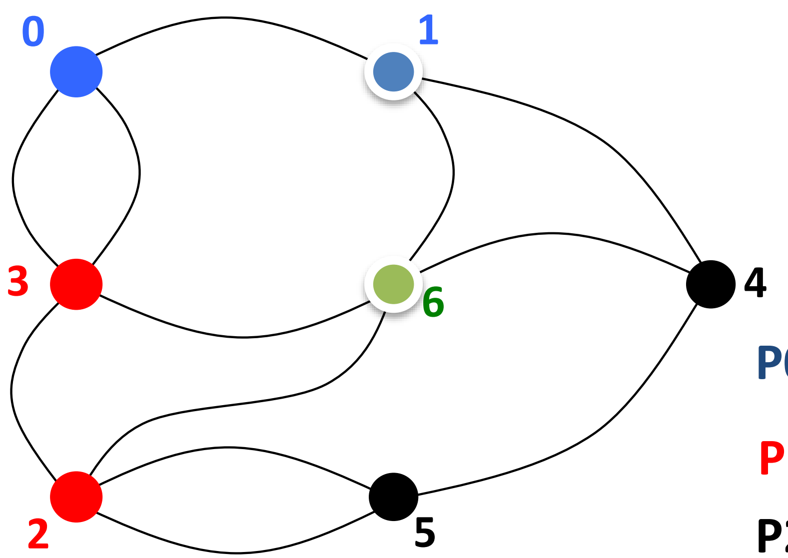
P2 No work

P3 **[6,1]** **[6,2]** **[6, 3]** **[6,4]**

Steps:

1. Local discovery: Explore adjacencies of vertices in current frontier.
2. Fold: All-to-all exchange of adjacencies.
3. Local update: Update distances/parents for unvisited vertices.

BFS with a 1D-partitioned graph



Current frontier: vertices 1 (partition **Blue**) and 6 (partition **Green**)

2. All-to-all exchange:

P0 [1,0] [1,4] [1,6]

P1 No work

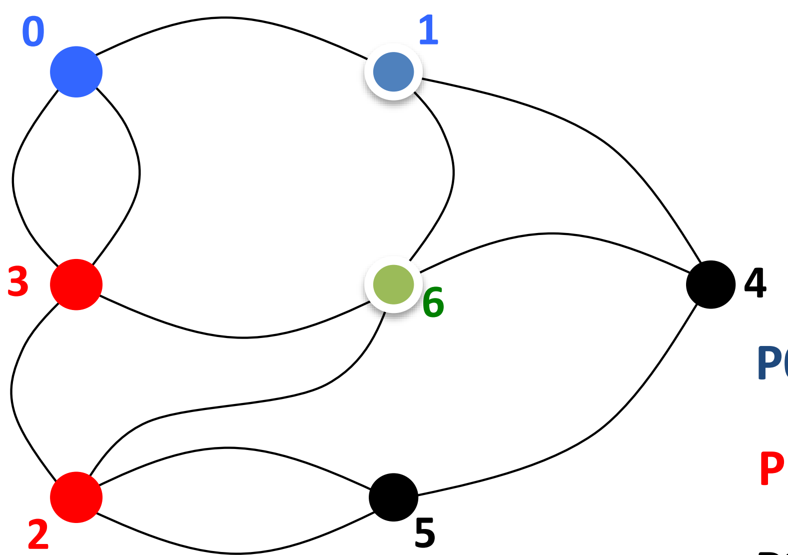
P2 No work

P3 [6,1] [6,2] [6, 3] [6,4]

Steps:

1. Local discovery: Explore adjacencies of vertices in current frontier.
2. Fold: All-to-all exchange of adjacencies.
3. Local update: Update distances/parents for unvisited vertices.

BFS with a 1D-partitioned graph



Current frontier: vertices 1 (partition **Blue**) and 6 (partition **Green**)

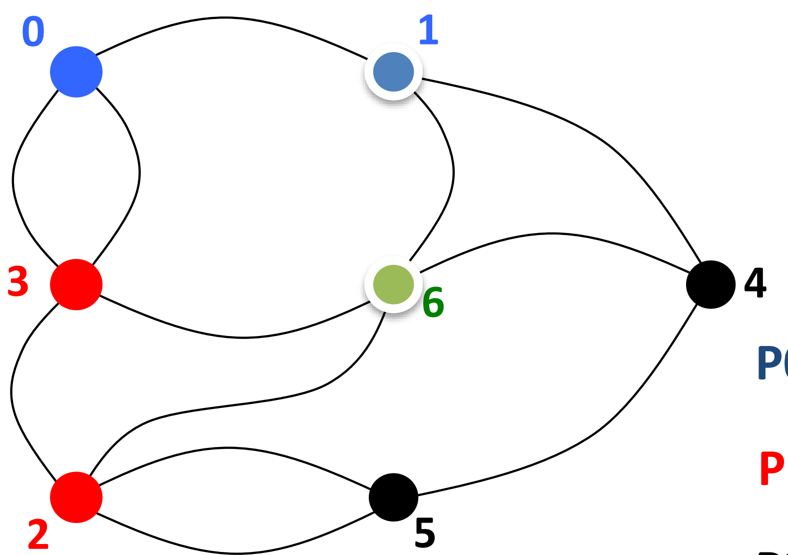
2. All-to-all exchange:

P0	[1,0]	[6,1]
P1	[6,2]	[6, 3]
P2	[6,4]	[1,4]
P3	[1,6]	

Steps:

1. Local discovery: Explore adjacencies of vertices in current frontier.
2. Fold: All-to-all exchange of adjacencies.
3. Local update: Update distances/parents for unvisited vertices.

BFS with a 1D-partitioned graph



Current frontier: vertices 1 (partition **Blue**) and 6 (partition **Green**)

3. Local update:

Frontier for next iteration

			Frontier for next iteration
P0	[1,0]	[6,1]	0
P1	[6,2]	[6, 3]	2, 3
P2	[6,4]	[1,4]	4
P3	[1,6]		

Steps:

1. Local discovery: Explore adjacencies of vertices in current frontier.
2. Fold: All-to-all exchange of adjacencies.
3. Local update: Update distances/parents for unvisited vertices.

Modeling parallel execution time

- Time dominated by local memory references and inter-node communication
- Assuming perfectly balanced computation and communication, we have

Local memory references:

$$\beta_L \frac{m}{p} + \alpha_{L,n/p} \frac{n+m}{p}$$

Inverse local RAM bandwidth

Local latency on working set |n/p|

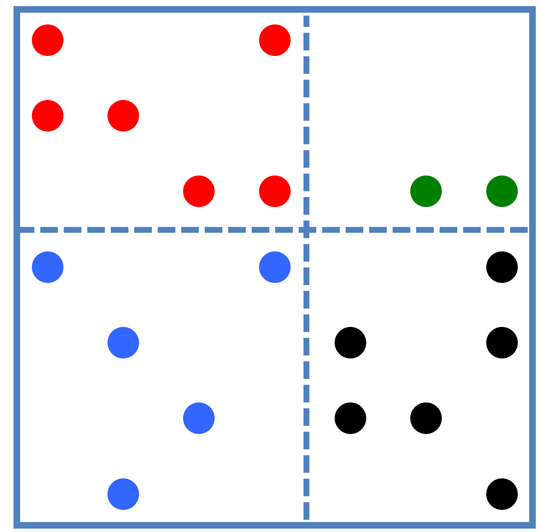
Inter-node communication:

$$\beta_{N,a2a}(p) \frac{edgecut}{p} + \alpha_N p$$

All-to-all remote bandwidth with p participating processors

BFS with a 2D-partitioned graph

- Avoid expensive *p*-way All-to-all communication step
- Each process collectively 'owns' n/p_r vertices
- Additional 'Allgather' communication step for processes in a row



Local memory references:

$$\beta_L \frac{m}{p} + \alpha_{L,n/p_c} \frac{n}{p} + \alpha_{L,n/p_r} \frac{m}{p}$$

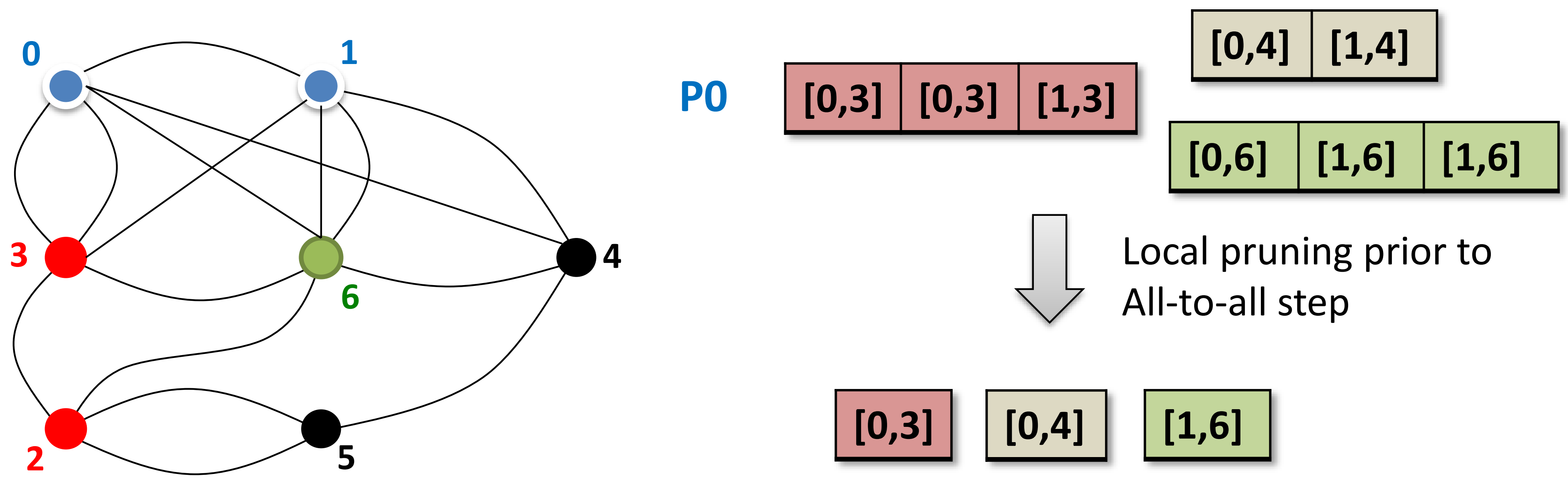
Inter-node communication:

$$\beta_{N,a2a}(p_r)\frac{edgecut}{p} + \alpha_N p_r +$$

$$\beta_{N,gather}(p_c)\left(1 - \frac{1}{p_r}\right)\frac{n}{p_c} + \alpha_N p_c$$

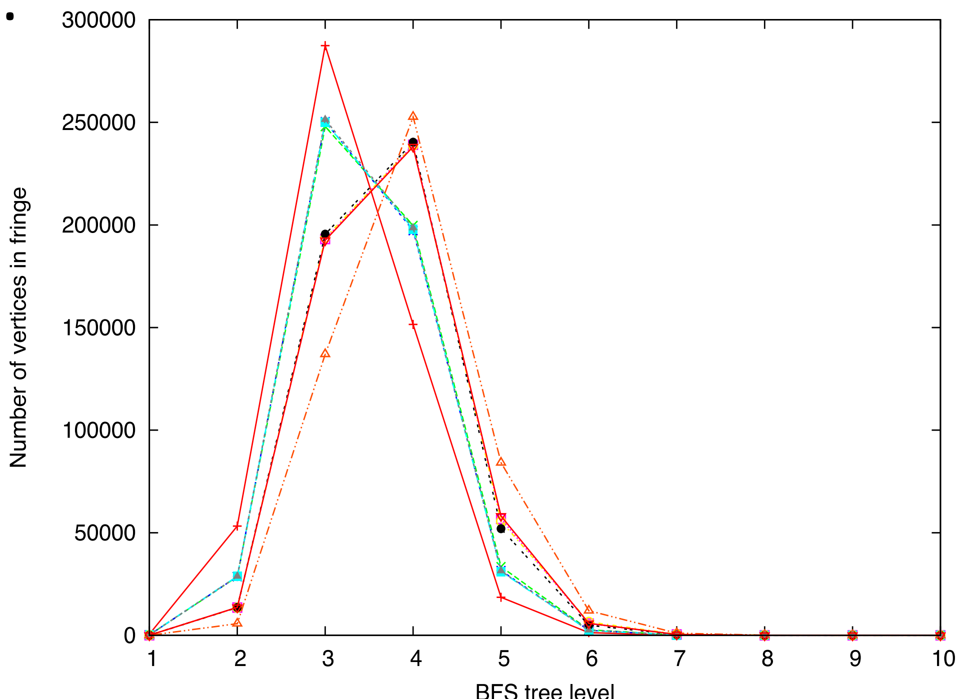
Temporal effects, communication-minimizing tuning prevent us from obtaining tighter bounds

- The volume of communication can be further reduced by maintaining state of non-local visited vertices



Predictable BFS execution time for synthetic small-world graphs

- Randomly permuting vertex IDs ensures load balance on R-MAT graphs (used in the Graph 500 benchmark).
- Our tuned parallel implementation for the NERSC Hopper system (Cray XE6) is ranked #2 on the current Graph 500 list.



Execution time is dominated by work performed in a few parallel phases

Buluc & Madduri, Parallel BFS on distributed memory systems, Proc. SC'11, 2011.

Modeling BFS execution time for real-world graphs

- Can we further reduce communication time utilizing existing partitioning methods?
- Does the model predict execution time for arbitrary low-diameter graphs?

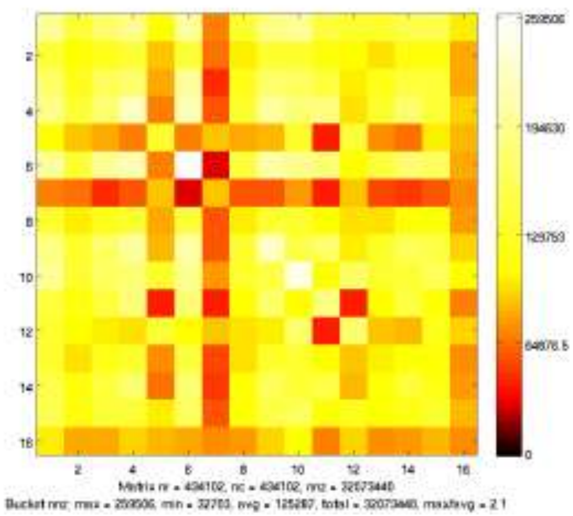
- We try out various partitioning and graph distribution schemes on the DIMACS Challenge graph instances
  - Natural ordering, Random, Metis, PaToH

Experimental Study

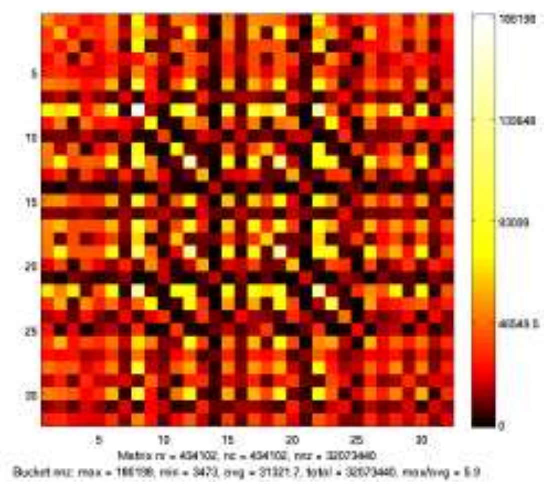
- The (weak) upper bound on aggregate data volume communication can be statically computed (based on partitioning of the graph)
- We determine runtime estimates of
  - Total aggregate communication volume
  - Sum of max. communication volume during each BFS iteration
  - Intra-node computational work balance
  - Communication volume reduction with 2D partitioning
- We obtain and analyze execution times (at several different parallel concurrencies) on a Cray XE6 system (Hopper, NERSC)

Orderings for the CoPapersCiteseer graph

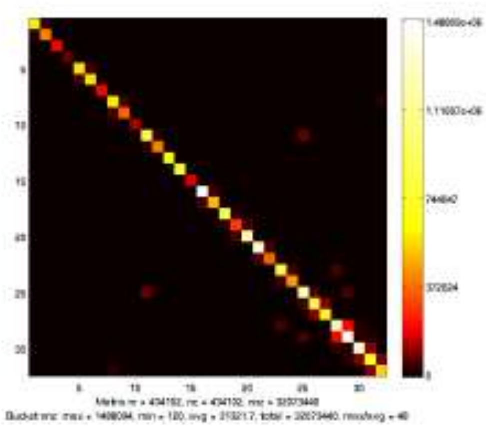
Natural



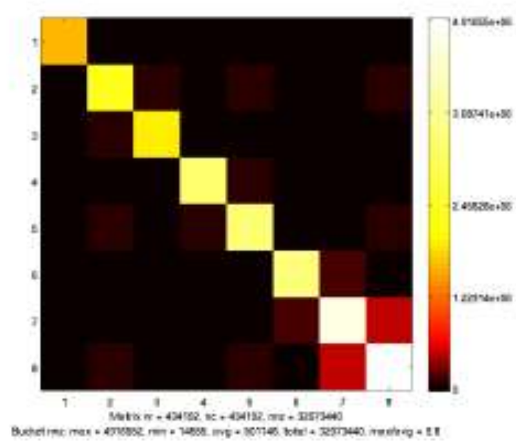
Random



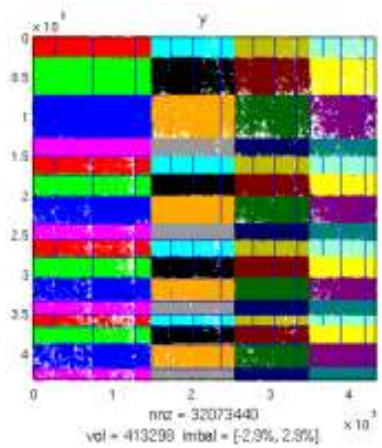
Metis



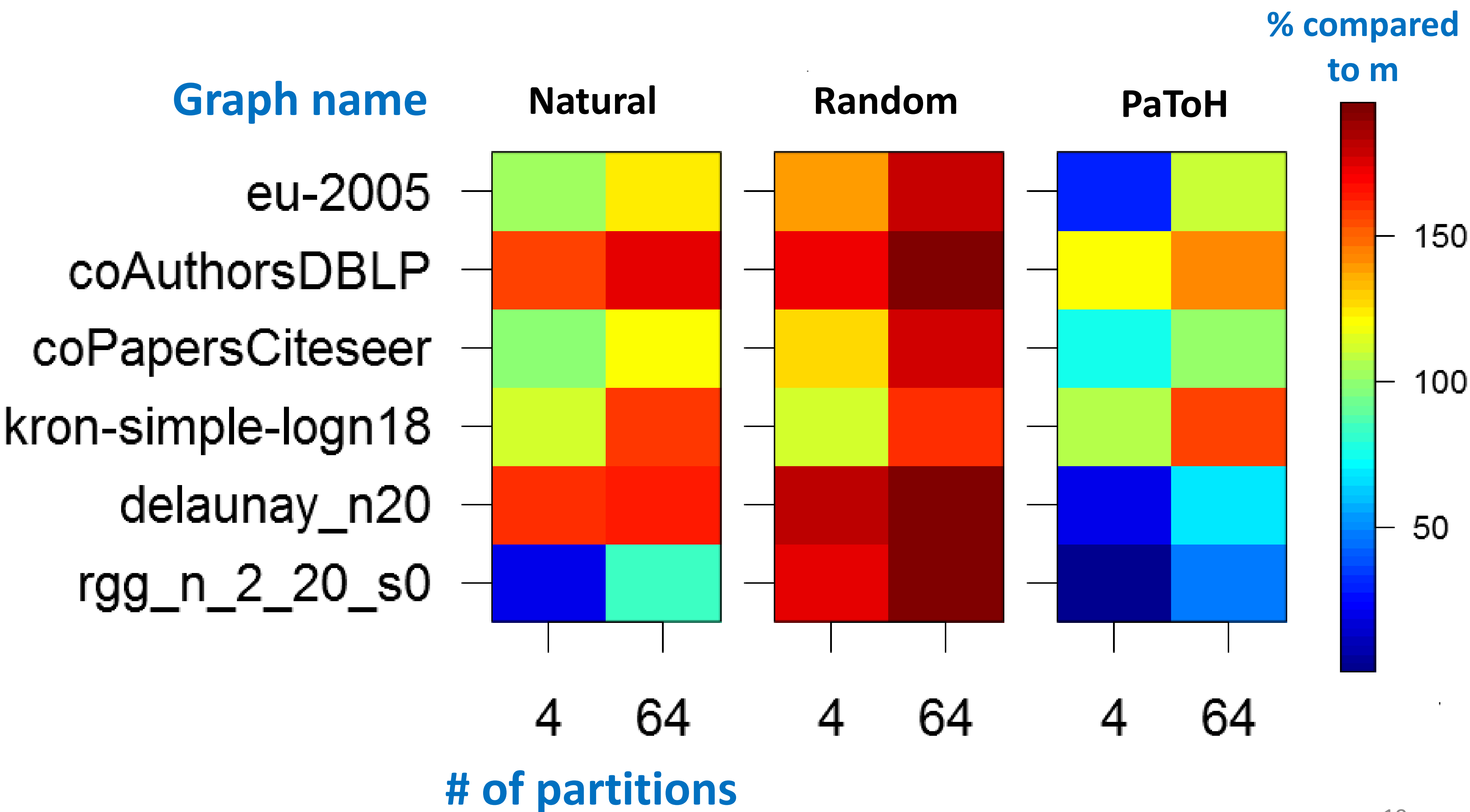
PaToH



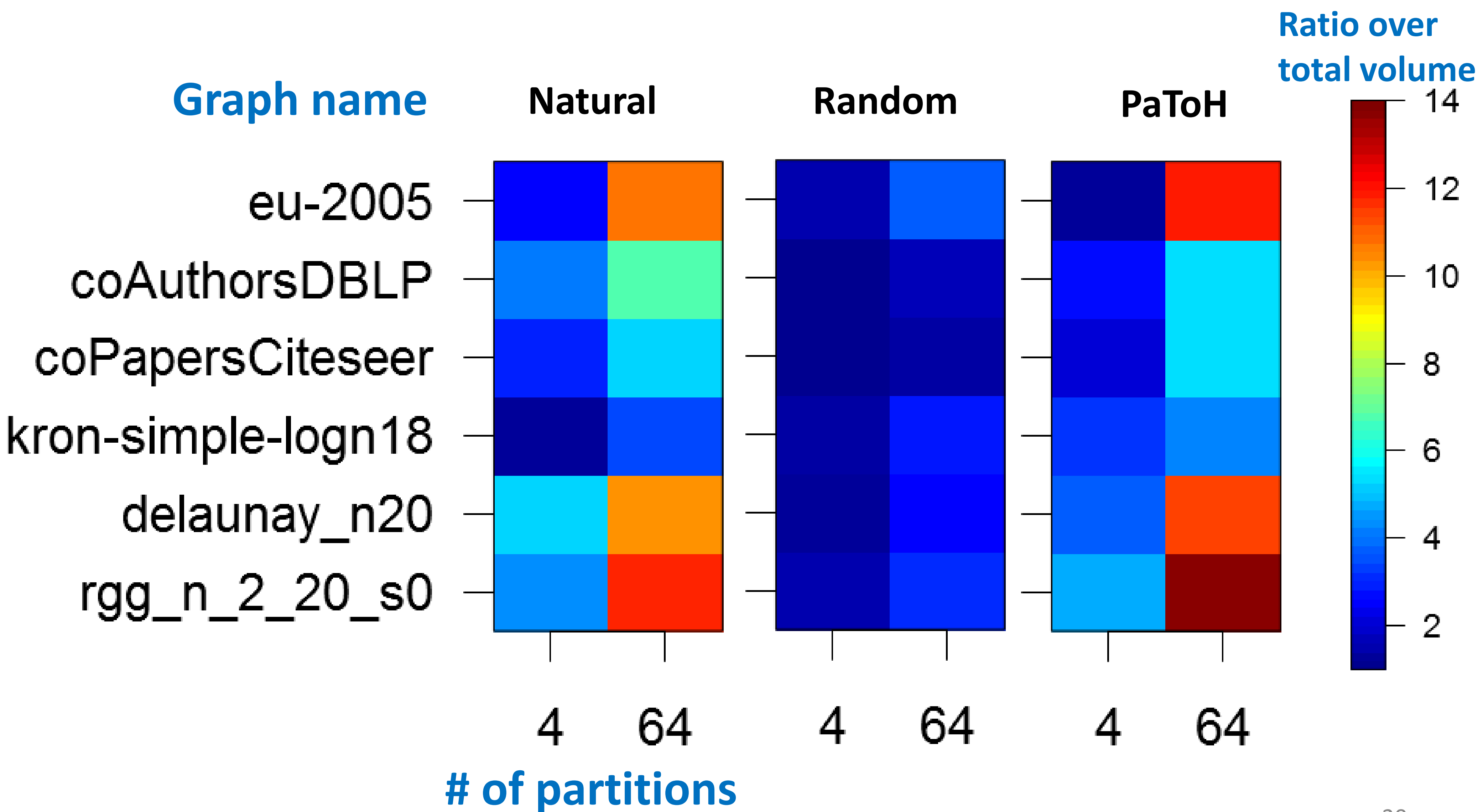
PaToH checkerboard



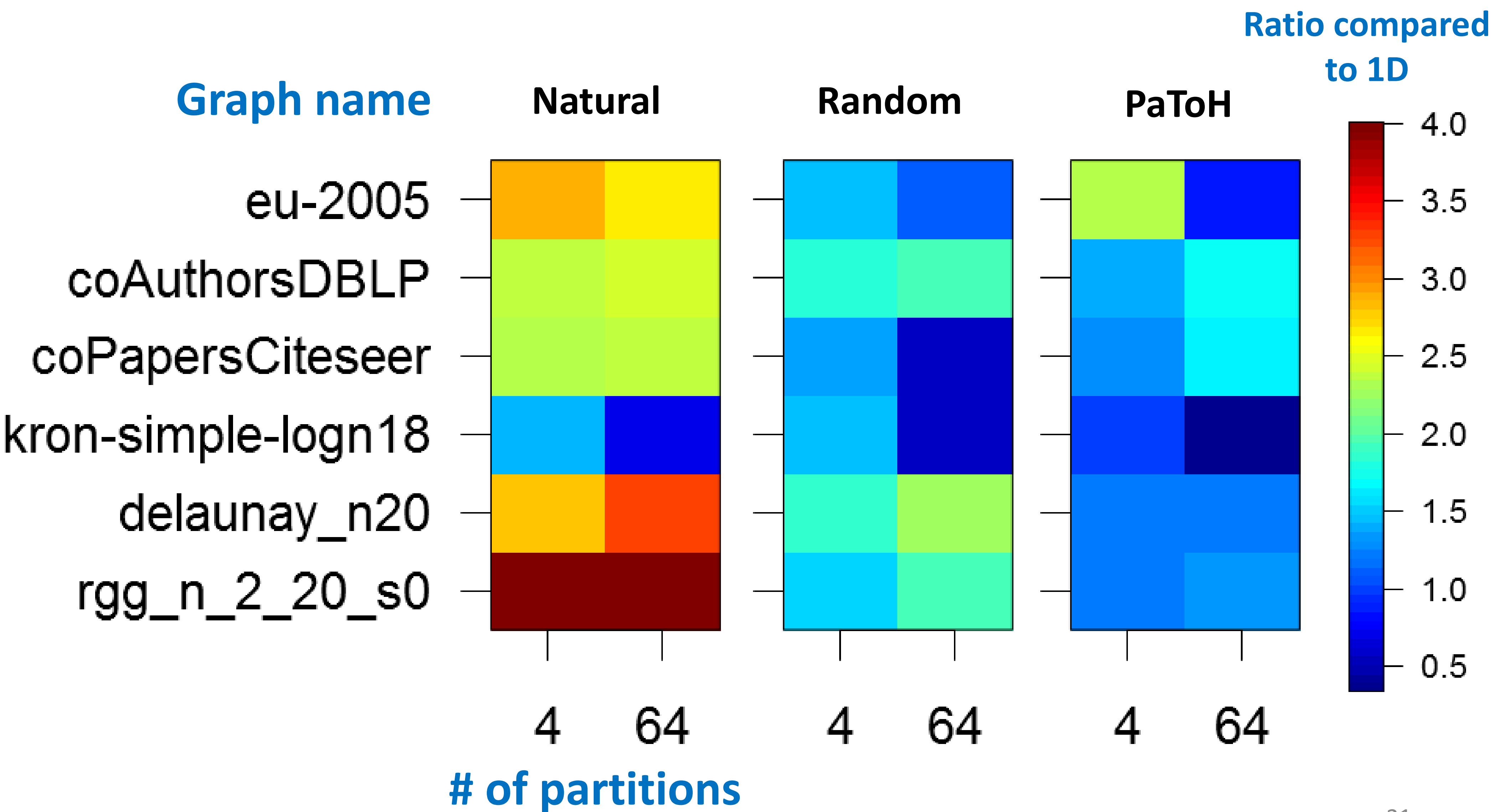
BFS All-to-all phase total communication volume normalized to # of edges (m)



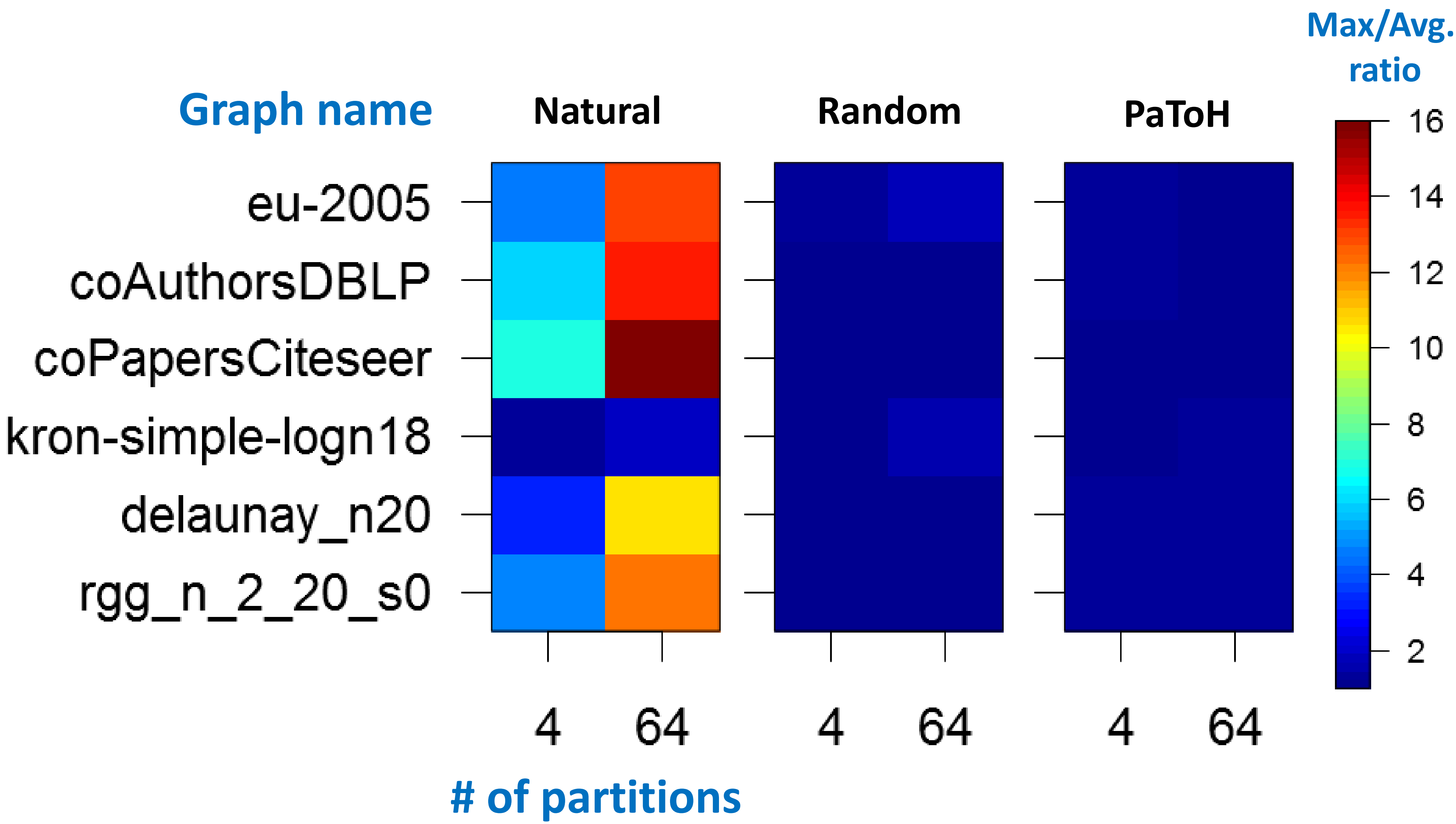
Ratio of max. communication volume across iterations to total communication volume



Reduction in total All-to-all communication volume with 2D partitioning



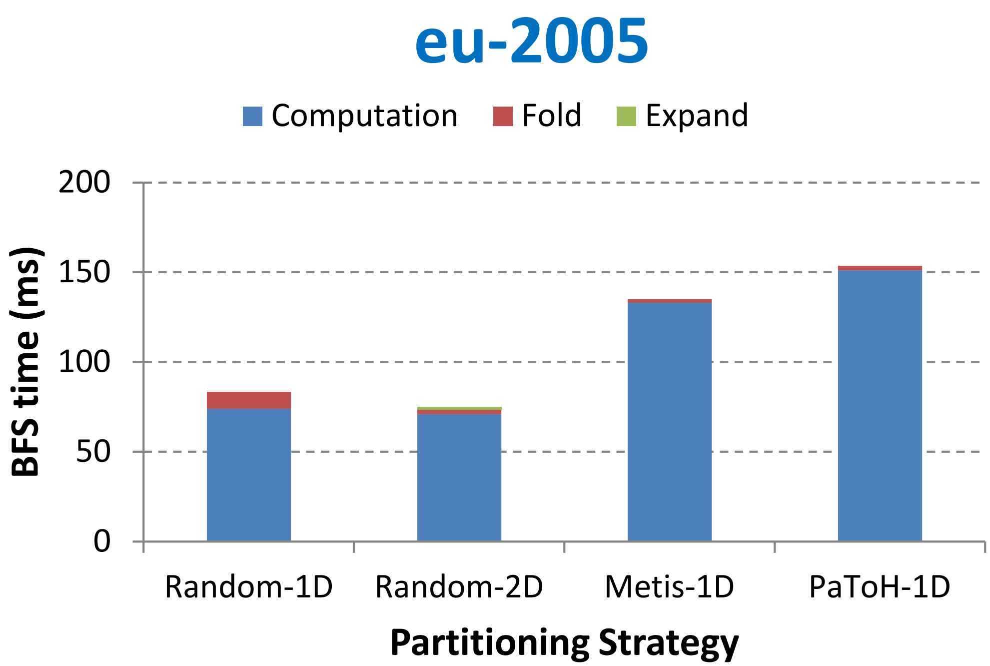
Edge count balance with 2D partitioning

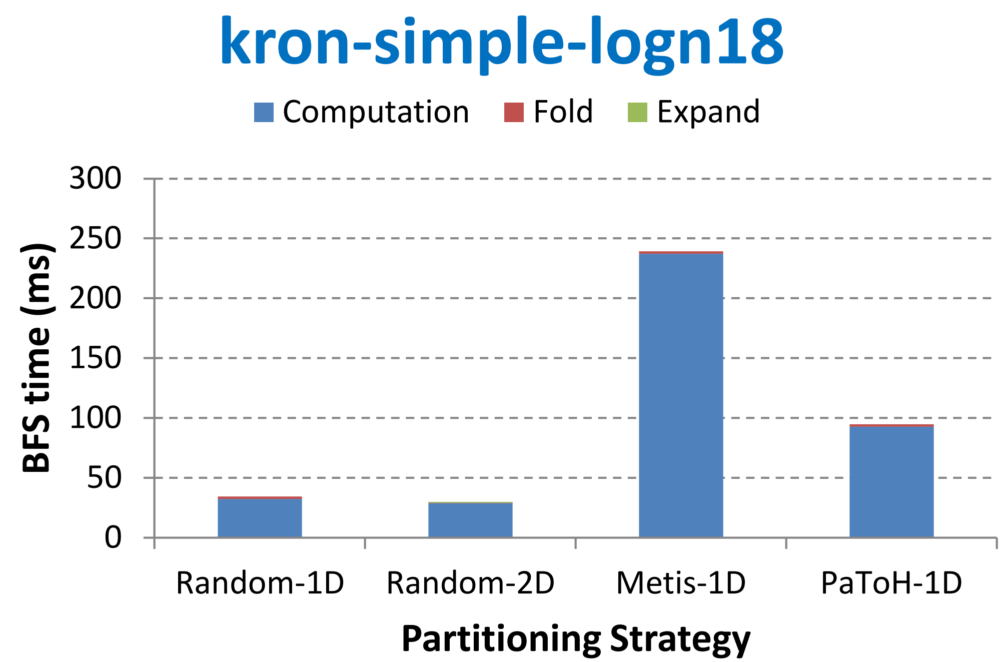


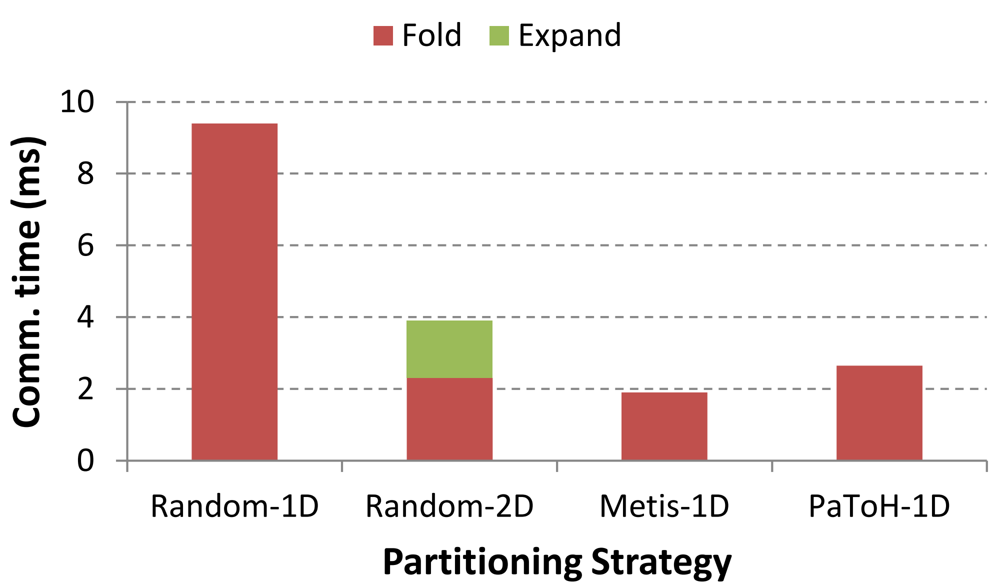
Parallel speedup on Hopper with 16-way partitioning

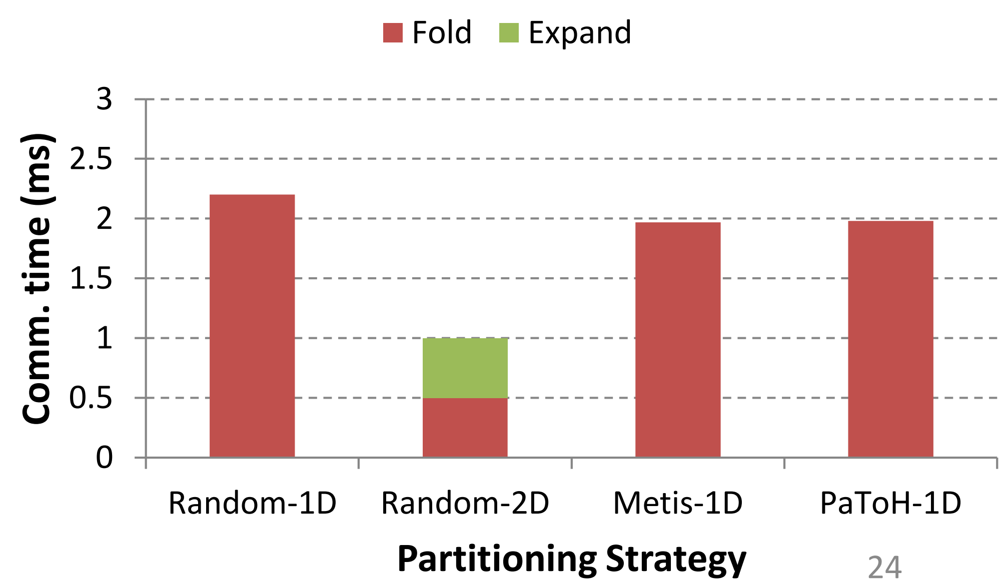
Graph	Perf Rate $p = 1 \times 1$	Relative Speedup $p = 16 \times 1$			Rel. Speedup over 1D $p = 4 \times 4$		
	N	N	R	M	N	R	M
coPapersCiteseer	24.9	5.6×	9.7×	8.0×	0.4×	**1.0×**	0.4×
eu-2005	23.5	6.1×	7.9×	5.0×	0.5×	**1.1×**	0.5×
kron-simple-logn18	24.5	12.6×	12.6×	1.8×	1.1×	**1.1×**	1.4×
er-fact1.5-scale20	14.1	11.2×	11.2×	11.5×	1.1×	**1.2×**	0.8×
road_central	7.2	3.5×	2.2×	3.5×	0.6×	0.9×	0.5×
hugebubbles-00020	7.1	3.8×	2.7×	3.9×	0.7×	0.9×	0.6×
rgg_n_2_20_s0	14.1	2.5×	3.4×	2.6×	0.6×	1.2×	0.6×
delaunay_n18	15.0	1.9×	1.6×	1.9×	0.9×	1.4×	0.7×

Execution time breakdown



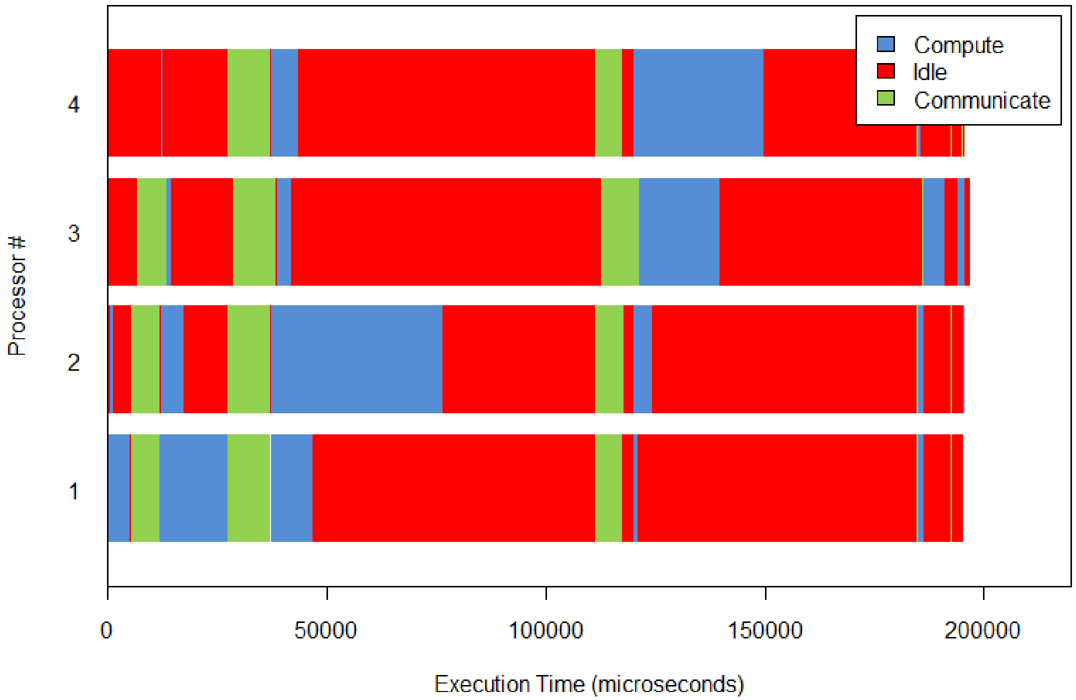




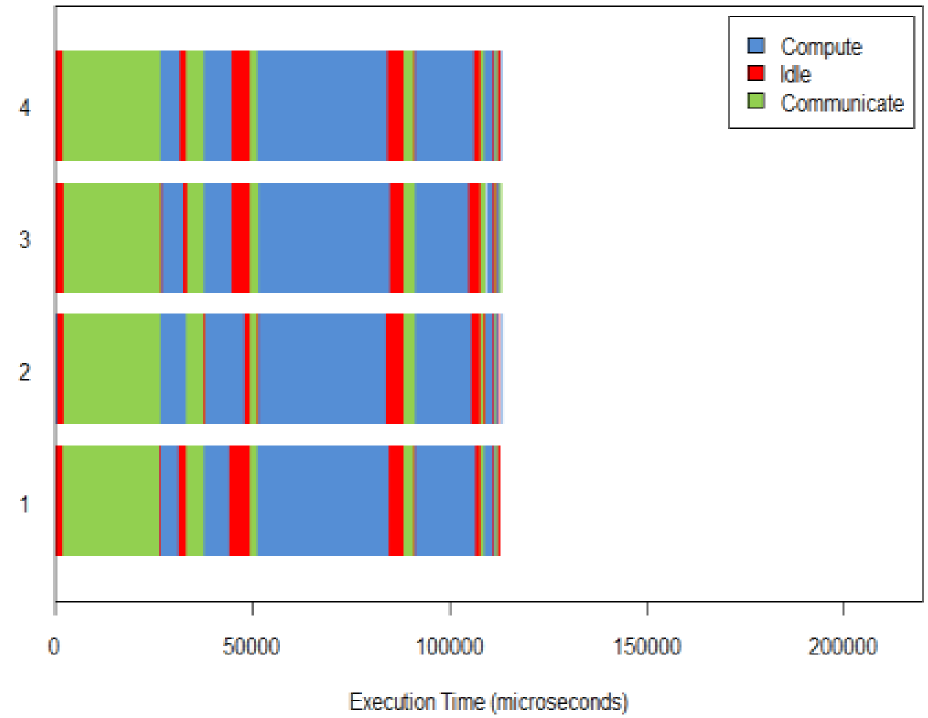


Imbalance in parallel execution

eu-2005, 16 processes*



PaToH



Random

*** Timeline of 4 processes shown in figures.**
PaToH-partitioned graph suffers from severe load imbalance in computational phases.

Conclusions

- Randomly permuting vertex identifiers improves computational and communication load balance, particularly at higher process concurrencies
- Partitioning methods reduce overall communication volume, but introduce significant load imbalance
- Substantially lower parallel speedup with real-world graphs compared to synthetic graphs (8.8X vs 50X at 256-way parallel concurrency)
  - Points to the need for dynamic load balancing