# HYPERX: SCALABLE HYPERGRAPH PROCESSING

Jin Huang

November 15, 2015

The University of Melbourne

# OVERVIEW

# RESEARCH OUTLINE

# SCALABLE HYPERGRAPH PROCESSING

Any (high-order) relationships with more than 2 participants.

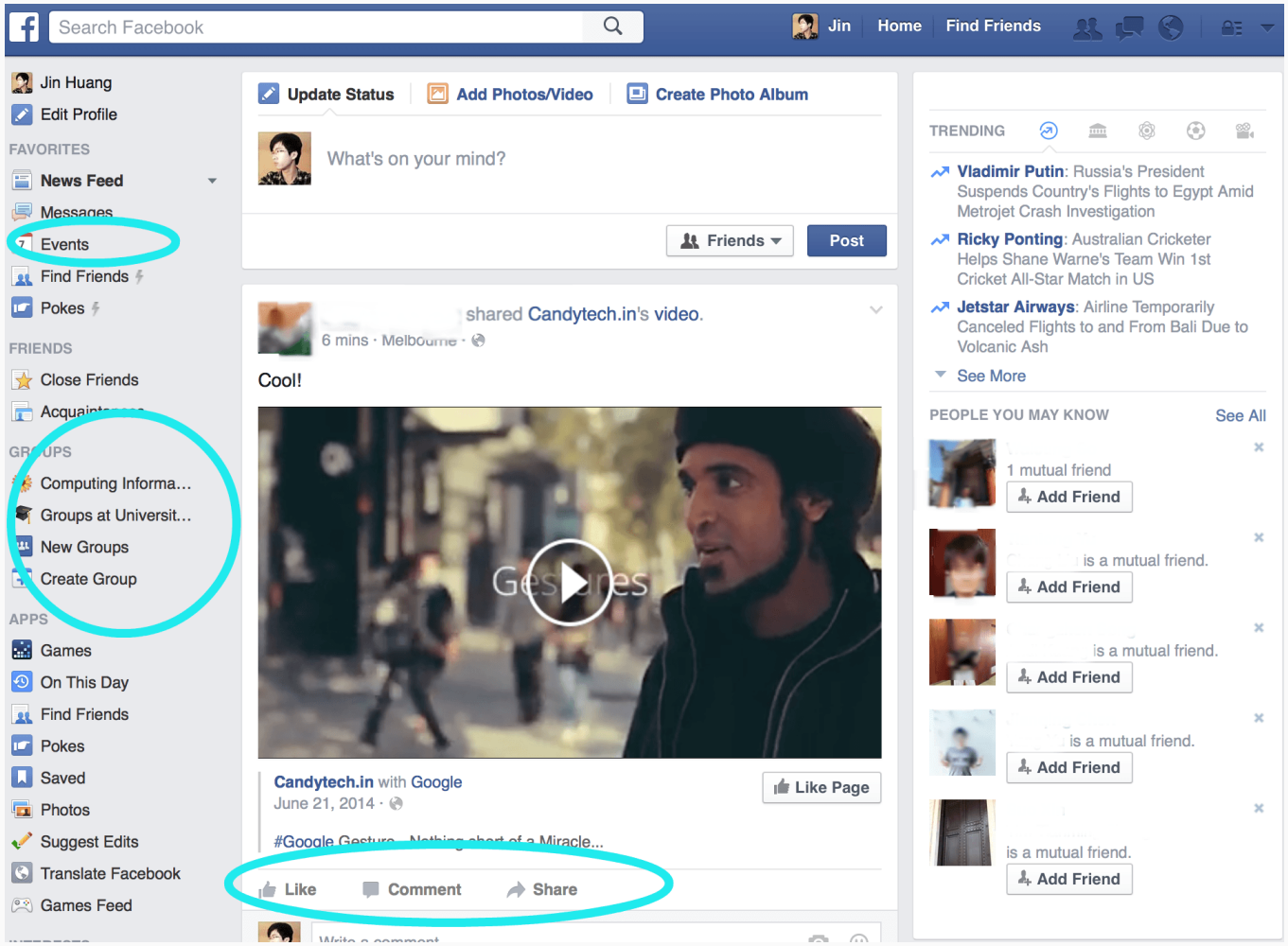


**Figure 1:** A few high-order relationships

# REPRESENTATIVE EXISTING HYPERGRAPH STUDIES

**Table 1:** Various hypergraph learning studies in literature

| Application | Study | Vertex | Hyperedge |
|---|---|---|---|
| Recommendation | [TMCCA'13] | Songs and users | Listening histories |
| Text retrieval | [SIGIR'08] | Documents | Semantic similarities |
| Image retrieval | [Pattern Recognition'13] | Images | Descriptor similarities |
| Multimedia | [Multimedia'08] | Videos | Hyperlinks |
| Bioinformatics | [ICDM'13] | Proteins | Interactions |
| Social mining | [AAAI'14] | Users | Communities |
| Machine learning | [Signal Processing'14] | Data Records | Labels |

Converting to a graph!

**Option I** a bipartite

**Option II** a clique

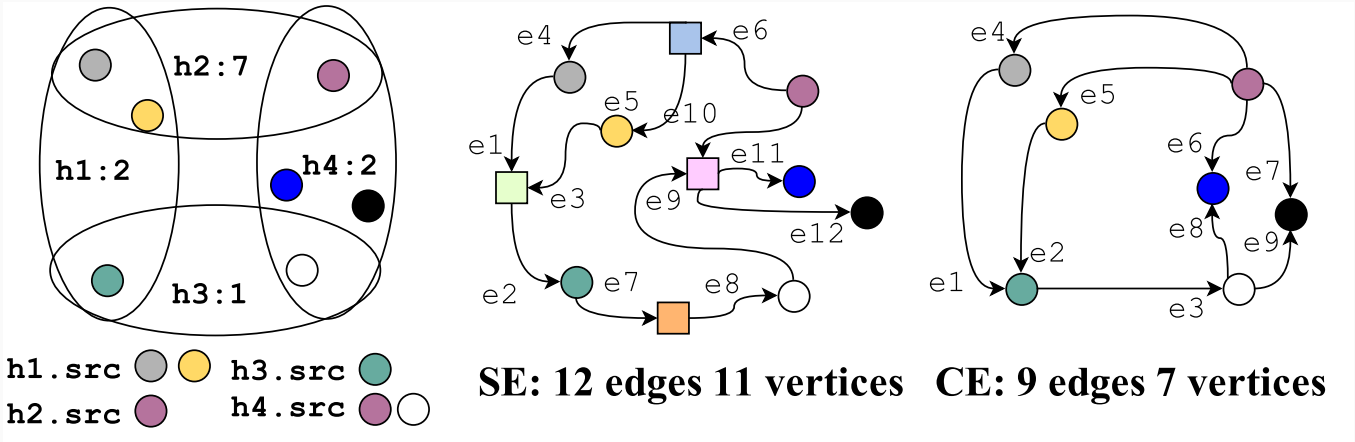


**Figure 2:** Graph conversion inflates the problem size

# CHALLENGES I

Scalable graph frameworks: GraphLab, Giraph, GraphX, etc.

- synchronous BSP (Pregel)
- vertex-centric style
- vertex replication and aggregation

Scalable graph frameworks: GraphLab, Giraph, GraphX, etc.

- synchronous BSP (Pregel)
- vertex-centric style
- vertex replication and aggregation

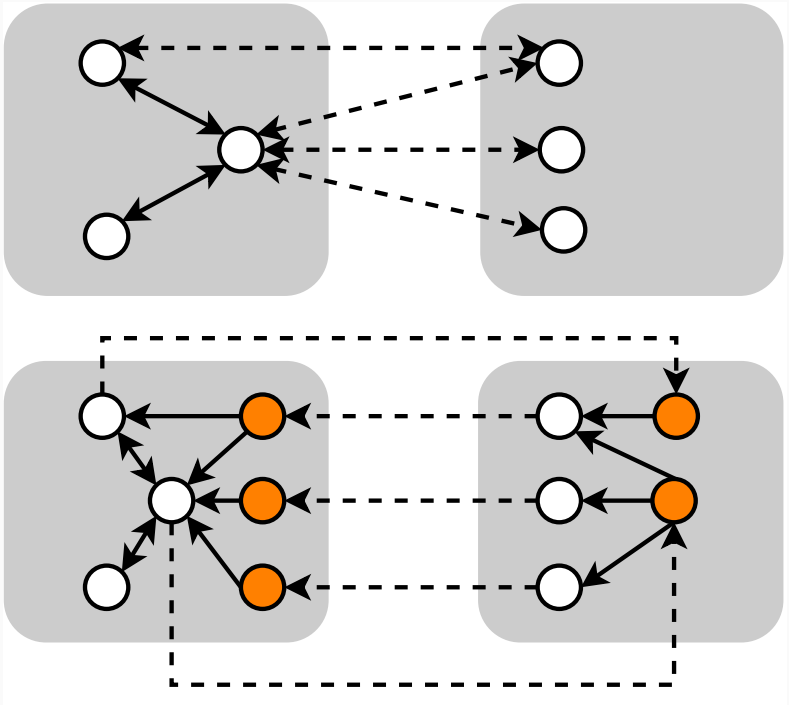

**Figure 3:** Vertex replicas to reduce network communication

# CHALLENGES I

Scalable graph frameworks: GraphLab, Giraph, GraphX, etc.

- synchronous BSP (Pregel)
- vertex-centric style
- vertex replication and aggregation

**Inflated Size** 2M *V* and 15M *H* -> 17M *V* and 1B *E*

**Excessive Replication** replicating both *V* and *H*

**Difficulty in Load Balance** two causes

1. *V* and *H* not active simultaneously
2. double overhead in each iteration

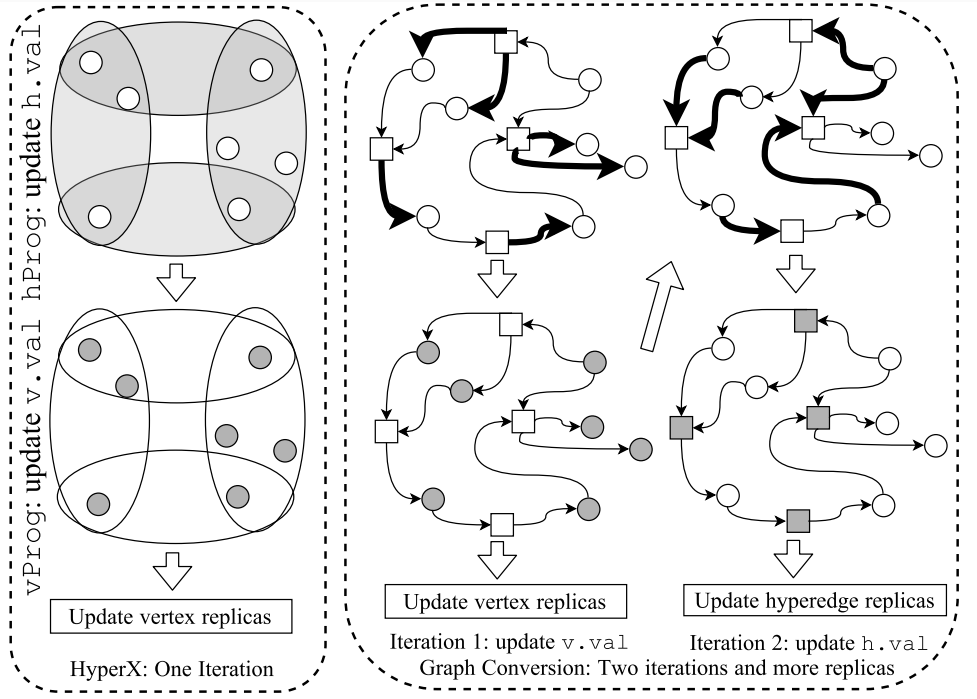


**Figure 3:** Two issues in balancing the loads

# IDEA

**To Support (API)** Random walks, label propagation, spectral

**Inflated Size (Representation)** a distributed hypergraph

**Excessive Replication (Representation)** replicate only $V$

**Difficulty in Lload Balance (Partitioning)** An optimization

- minimizes the communication cost
- minimizes the replication cost
- balances both $V$ and $H$ loads

# PROPOSED SOLUTION: HYPERX

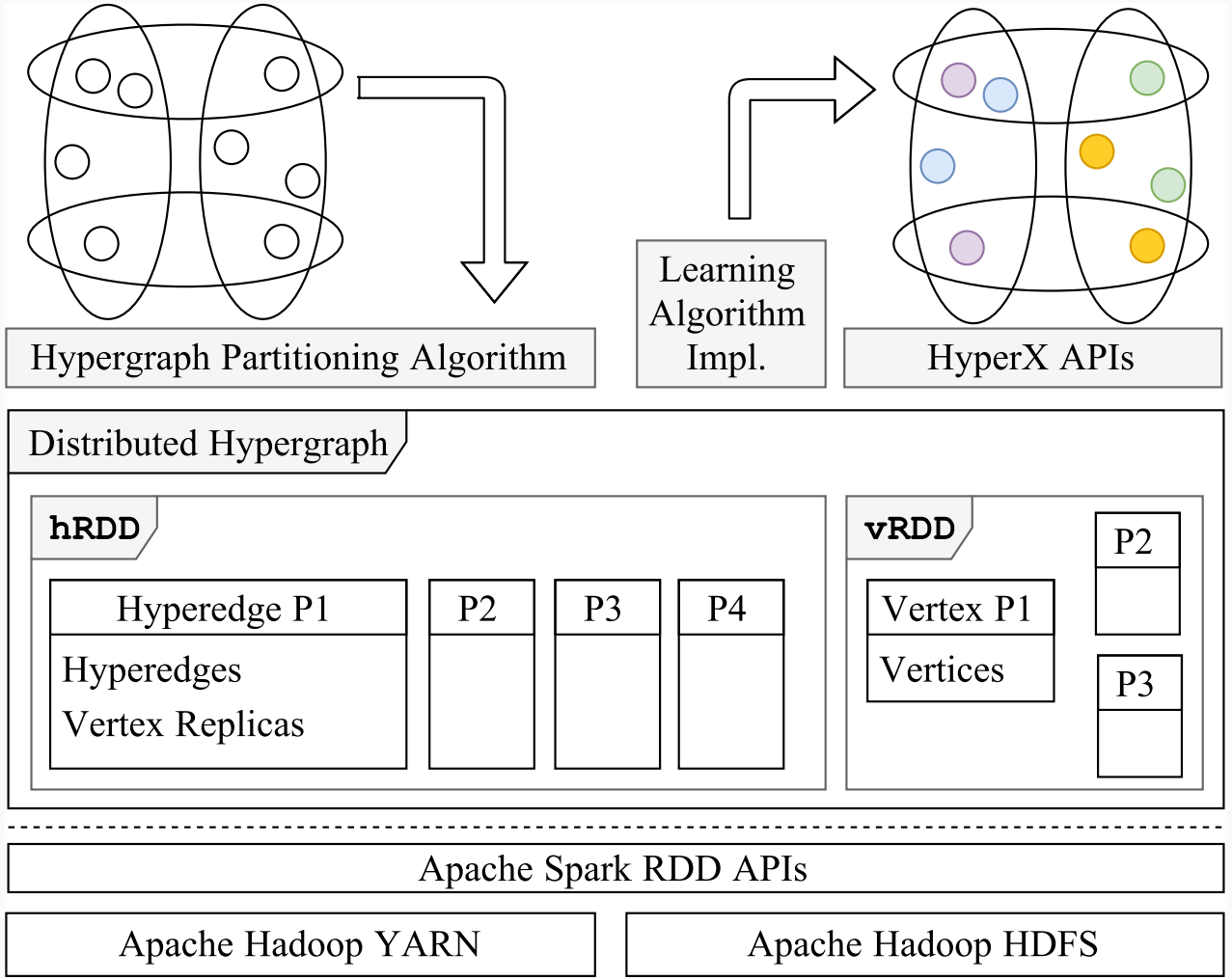


**Figure 4:** An overview of HyperX implemented over Spark

- Algorithms expressed as
  - *vProg* updates vertex values given incident hyperedges
  - *hProg* update hyperedge values given incident vertices

**Table 2:** HyperX Main APIs

| Name | Usage |
|---|---|
| *joinV* | *vProg* as distributed joins |
| *mrTuples* | *hProg* on hyperedges and reduce vertices |
| *mapV* | update vertices independently (locally) |
| *mapH* | update hyperedges independently (locally) |
| *subH* | restrict computation over a sub-hypergraph |
| *HyperPregel* | iteratively execute *mrTuple* and *joinV* |

**Algorithm 1:** HyperPregel

input : $\mathcal{G}$: `Hypergraph[V,H]`, `vProg`: `(Id,V)` $\Rightarrow$ `V`, `hProg`: `Tuple` $\Rightarrow$ `M`, `combine`: `(M,M)` $\Rightarrow$ `M`, *initial*: `M`

output: `RDD[(Id, V)]`

1 $\mathcal{G} \leftarrow \mathcal{G}$.**mapV**((*id*, *v*) $\Rightarrow$ `vProg`(*id*, *v*, *initial*))
2 *msg* $\leftarrow \mathcal{G}$.**mrTuples**(`hProg`, `combine`)
3 while $|msg| > 0$ do
4 $\quad \mathcal{G} \leftarrow \mathcal{G}$.**joinV** (*msg*) (`vProg`).**subH**(`v'`, `t'`)
5 $\quad$ *msg* $\leftarrow \mathcal{G}$.**mrTuples**(`hProg`, `combine`)
6 return $\mathcal{G}$.**vertices**

**Algorithm 2:** Random Walks (RW) with restart

**input** : $\mathcal{G}$, label vertex set $L$, restart probability $rp$
**output**: `RDD[(Id, Double)]`

1 `vProg(id,(v,d),msg)`= $((1 - rp) \times msg + rp \times v, d)$
2 `hProg(`$\mathcal{S}$`,`$\mathcal{D}$`,`$Sd$`,`$Dd$`,`$h$`)`= $\sum_{i \leq |\mathcal{S}|} \frac{\mathcal{S}_i}{Sd_i \times |\mathcal{D}|}$
3 `combine(a,b)`= $a + b$
4 $\mathcal{G} \leftarrow \mathcal{G}$.**joinV** ($\mathcal{G}$.**outDeg**, $(id, v, d) \Rightarrow d$)
5 $\mathcal{G} \leftarrow \mathcal{G}$.**mapV**($(id, v) \Rightarrow$ **if** $id \in L$ $(1.0, v)$ **else** $(0.0, v)$)
6 $\mathcal{G}$.**HyperPregel**($\mathcal{G}$, `vProg`, `hProg`, `combine`,0)

# DETAILS: REPRESENTATION

Built on Spark's RDD, how to represent a hypergraph?

- Vertices *vRDD*
- Hyperedges *hRDD*
  - Multiple vertices
  - $\times$ list or set
  - $\surd$ flattened ($vid, hid, isSrc$) in columnar arrays
  - saves 41% to 88% memory consumption

# DETAILS: REPRESENTATION

Built on Spark's RDD, how to represent a hypergraph?

- Vertices *vRDD*
- Hyperedges *hRDD*
- To do *mrTuples* locally, replicate vertices
  - One replica is adequate
  - Cost in distributed *vProg*
  - Cost in updating replicas
  - Cost in storing replicas
- How to partition *vRDD* and *hRDD* to minimize the cost?

# DETAILS: PARTITIONING INTRODUCTION

Different from *vertex-cut* or *edge-cut* in graph literature

- Cut both vertices and hyperedges simultaneously
- Minimizes the vertex replicas (with local aggregation)
- With separate load constaints on *vProg* and *hProg*

# DETAILS: PARTITIONING OBJECTIVE FORMULATION

$n$ vertices, $m$ hyperedges, $k$ workers, $a_h$ the *arity* of $h$

- number of replicas for vertex $u$

$$R(\mathbf{x_u}, \mathbf{y}) = \sum_{i=1}^{k} max((1 - x_{u,i} - \prod_{h \in N(u)} (1 - y_{h,i}), 0)$$

# DETAILS: PARTITIONING OBJECTIVE FORMULATION

$n$ vertices, $m$ hyperedges, $k$ workers, $a_h$ the *arity* of $h$

- number of replicas for vertex $u$

$$R(\mathbf{x_u}, \mathbf{y}) = \sum_{i=1}^{k} max((1 - x_{u,i} - \prod_{h \in N(u)} (1 - y_{h,i}), 0)$$

- to optimize

$$\textbf{minimize} \sum_{u \in \mathcal{V}} R(\mathbf{x_u}, \mathbf{y})$$

$$\textbf{subject to} \sum_{h \in H} y_{h,i} a_h \leq (1 + \alpha) \frac{\sum_{h \in \mathcal{H}} a_h}{k}, i \in \{1, 2, ..., k\}$$

$$\sum_{u \in \mathcal{V}} x_{u,i} R(\mathbf{x_u}, \mathbf{y}) \leq (1 + \beta) \frac{\sum_{u \in \mathcal{V}} R(\mathbf{x_u}, \mathbf{y})}{k}, i \in \{1, 2, ..., k\}$$

How hard?

- a special case where $\alpha = 0$ and $\beta = +\infty$

$$\textbf{minimize} \sum_{u \in \mathcal{V}} \sum_{i=1}^{k} (1 - \prod_{h \in N(u)} (1 - y_{h,i}))$$

$$\textbf{subject to} \sum_{h \in \mathcal{H}} y_{h,i} a_h \leq \frac{\sum_{h \in \mathcal{H}} a_h}{k}, i \in \{1, 2, ..., k\}$$

  - reduction from the strongly NP-Complete `3-Partition`
  - no polynomial solution with finite approximation factor
- in plain words, it is extremely hard!
- how about $\alpha > 0$?

Lable propagation partitioning (LPP)

- labels are partitions
- label both vertices and hyperedges
- iteratively update labels

Lable propagation partitioning (LPP)

- labels are partitions
- label both vertices and hyperedges
- iteratively update labels
- specifically,

$$L(h) = \arg\max_{i \in K} |\{v | v \in N(h) \wedge L(v) = i\}|$$

$$L(v) = \arg\max_{i \in K} (|\{h | h \in N(v) \wedge L(h) = i\}| \times e^{\frac{\bar{A}^2 - A_i^2}{\bar{A}^2}}),$$

where $A_i = \sum_{L(h)=i} a_h$.

# EXPERIMENTAL SETTINGS

- Metrics
  - data RDD size
  - data shuffuled
  - elapsed time
- Comparisons
  - HyperX (hx), Bipartite (star), Clique (clique)
  - random, greedy, aweto, hMetis, LPP
  - random walk (RW), label propagation (LP), spectural (SP)
- Environment
  - 8 node, 28 workers, network 600Mbps
  - Hadoop 2.4.0, YARN enabled, Spark 1.1.0
  - HyperX implemented in Scala

## Table 3: Datasets presented in the empirical study

| Dataset | $n$ | $m$ | $d_{min}$ | $d_{max}$ | $\bar{d}$ | $\sigma_d$ | $c_{vd}$ | $a_{min}$ | $a_{max}$ | $\bar{a}$ | $\sigma_a$ | $c_{va}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Medline Coauthor (Med) | 3.2m | 8m | 1 | 5913 | 10 | 36.91 | 3.69 | 2 | 744 | 4 | 2.15 | 0.54 |
| Orkut Communities (Ork) | 2.3m | 15m | 1 | 2958 | 46 | 80.23 | 1.74 | 2 | 9,120 | 71 | 70.81 | 1.00 |
| Friendster Communities (Fri) | 7.9m | 1.6m | 1 | 1700 | 5 | 5.14 | 1.03 | 2 | 9,299 | 81 | 81.39 | 1.00 |
| Synthetic (Zipfian $s = 2$) | 2m | 8m | 2 | 803 | 32 | 33.7 | 1.05 | 2 | 48,744 | 8 | 178.59 | 22.32 |
| | | 12m | 5 | 1,173 | 48 | 50.27 | 1.05 | 2 | 49,526 | 8 | 174.07 | 21.76 |
| | | 16m | 10 | 1,527 | 63 | 66.56 | 1.06 | 2 | 49,006 | 8 | 171.36 | 21.42 |
| | | 20m | 15 | 1,893 | 79 | 83.40 | 1.06 | 2 | 49,963 | 8 | 175.52 | 21.94 |
| | | 24m | 21 | 2,305 | 95 | 100.00 | 1.05 | 2 | 49,326 | 8 | 173.12 | 21.64 |
| | 4m | 16m | 1 | 1,102 | 32 | 36.04 | 1.13 | 2 | 49,843 | 8 | 173.12 | 21.64 |
| | 6m | | 1 | 940 | 21 | 25.04 | 1.19 | 2 | 49,728 | 8 | 179.55 | 22.44 |
| | 8m | | 1 | 799 | 16 | 19.42 | 1.21 | 2 | 49,526 | 8 | 173.84 | 21.73 |
| | 10m | | 1 | 716 | 13 | 15.79 | 1.21 | 2 | 49,932 | 8 | 173.84 | 21.73 |

# EVALUATING HYPERGRAPH REPRESENTATION: SPACE

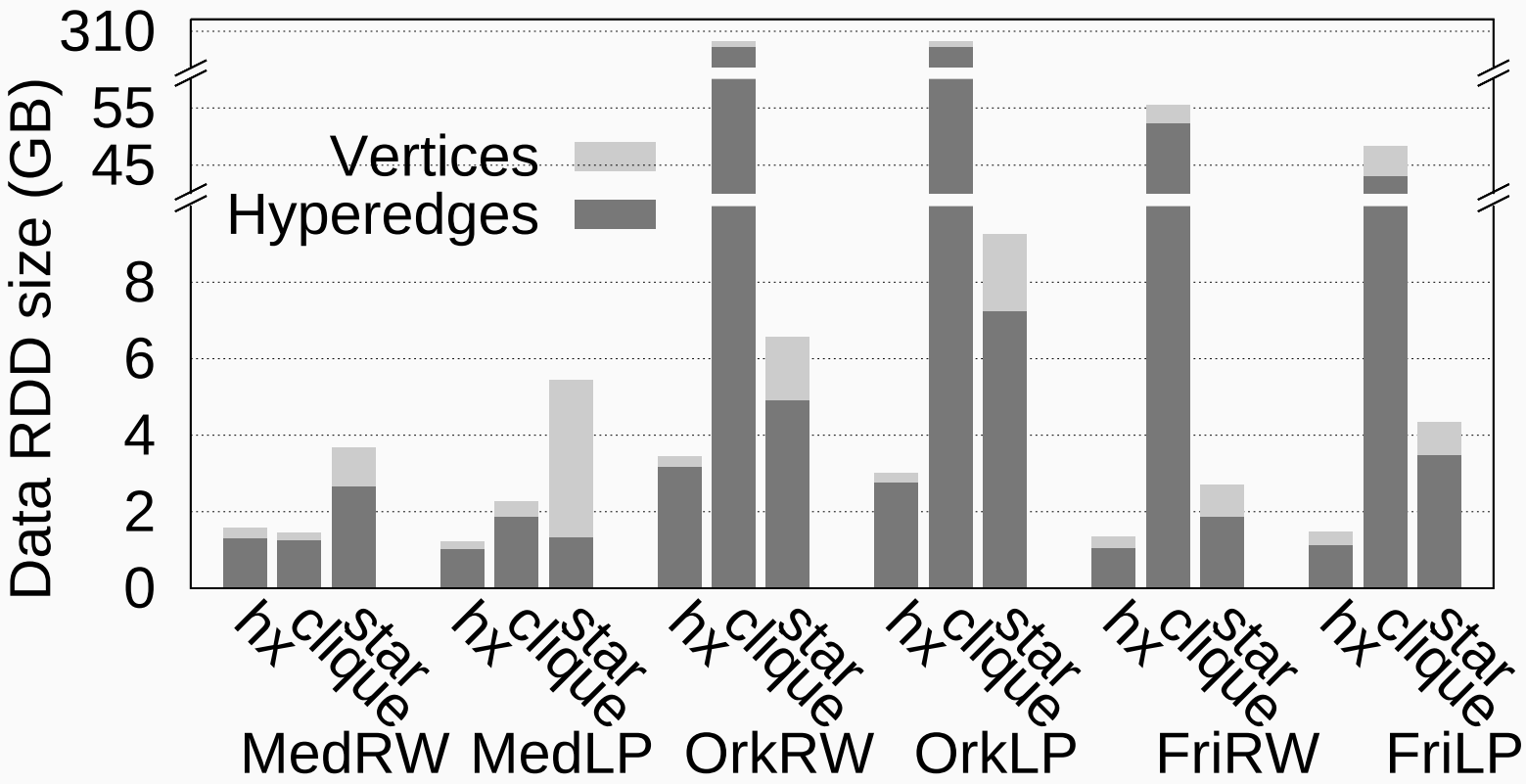


**Figure 5:** Memory Consumption of Data RDDs

HyperX consumes 44% to **77% less** memory than Bipartite.

# EVALUATING HYPERGRAPH REPRESENTATION: COMMUNICATION

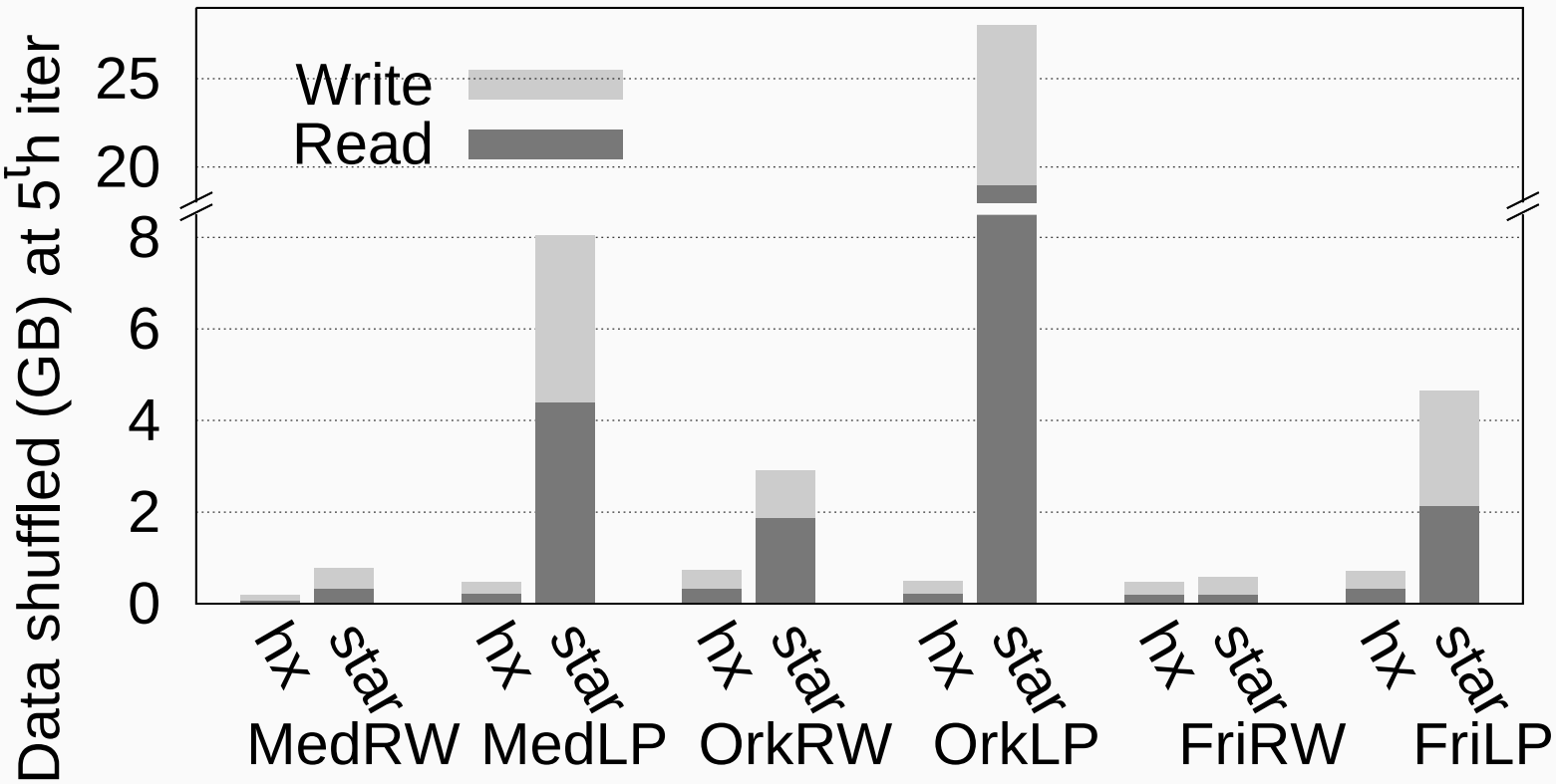


**Figure 6:** Data Shuffled on the Network

HyperX shuffles 19% to **98% fewer** data than Bipartite.

# EVALUATING HYPERGRAPH REPRESENTATION: TIME

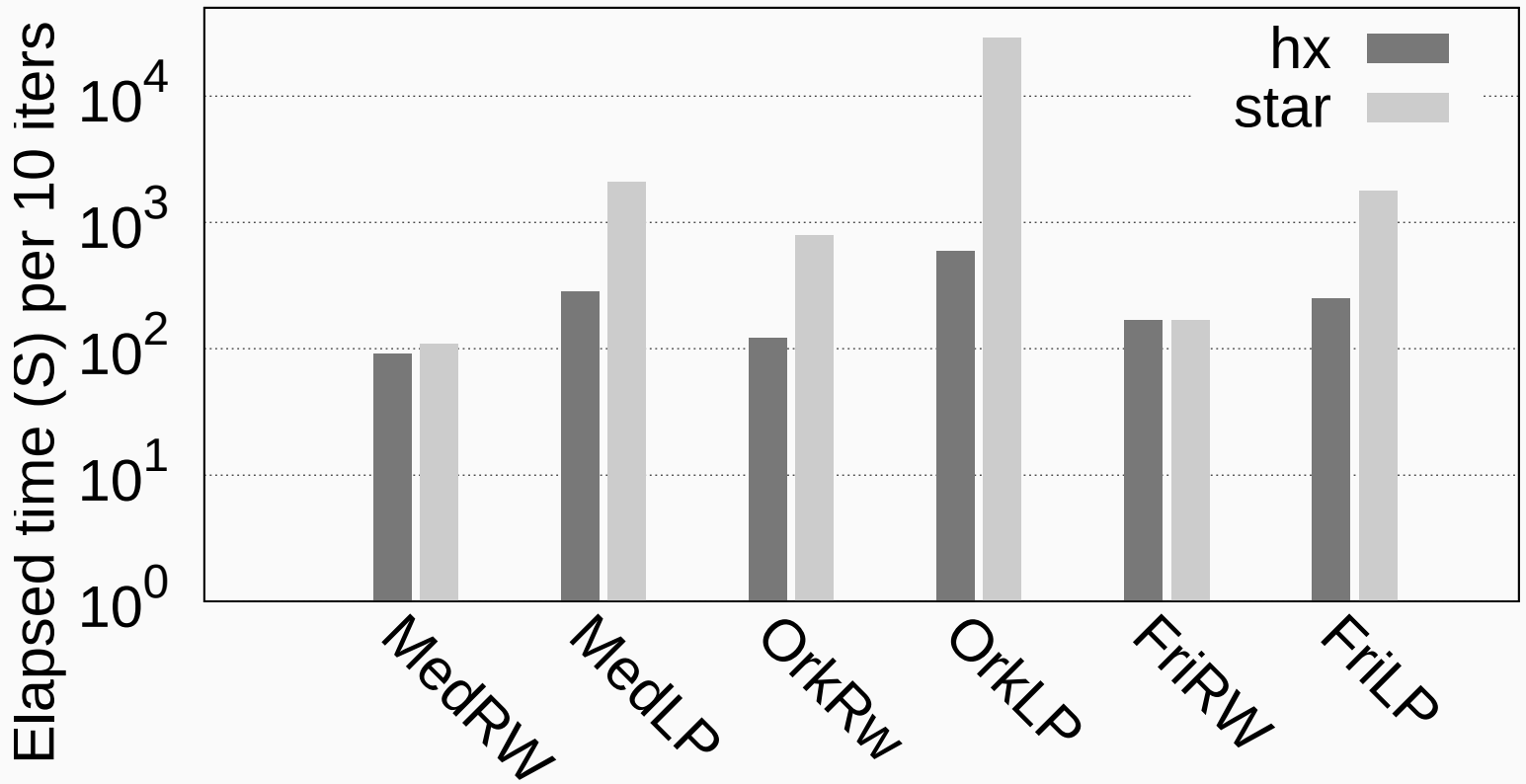


**Figure 7:** Elapsed Time

HyperX is up to **49.1 times** faster than Bipartite.

# EVALUATING PARTITIONING EFFECTIVENESS: REPLICA FACTOR

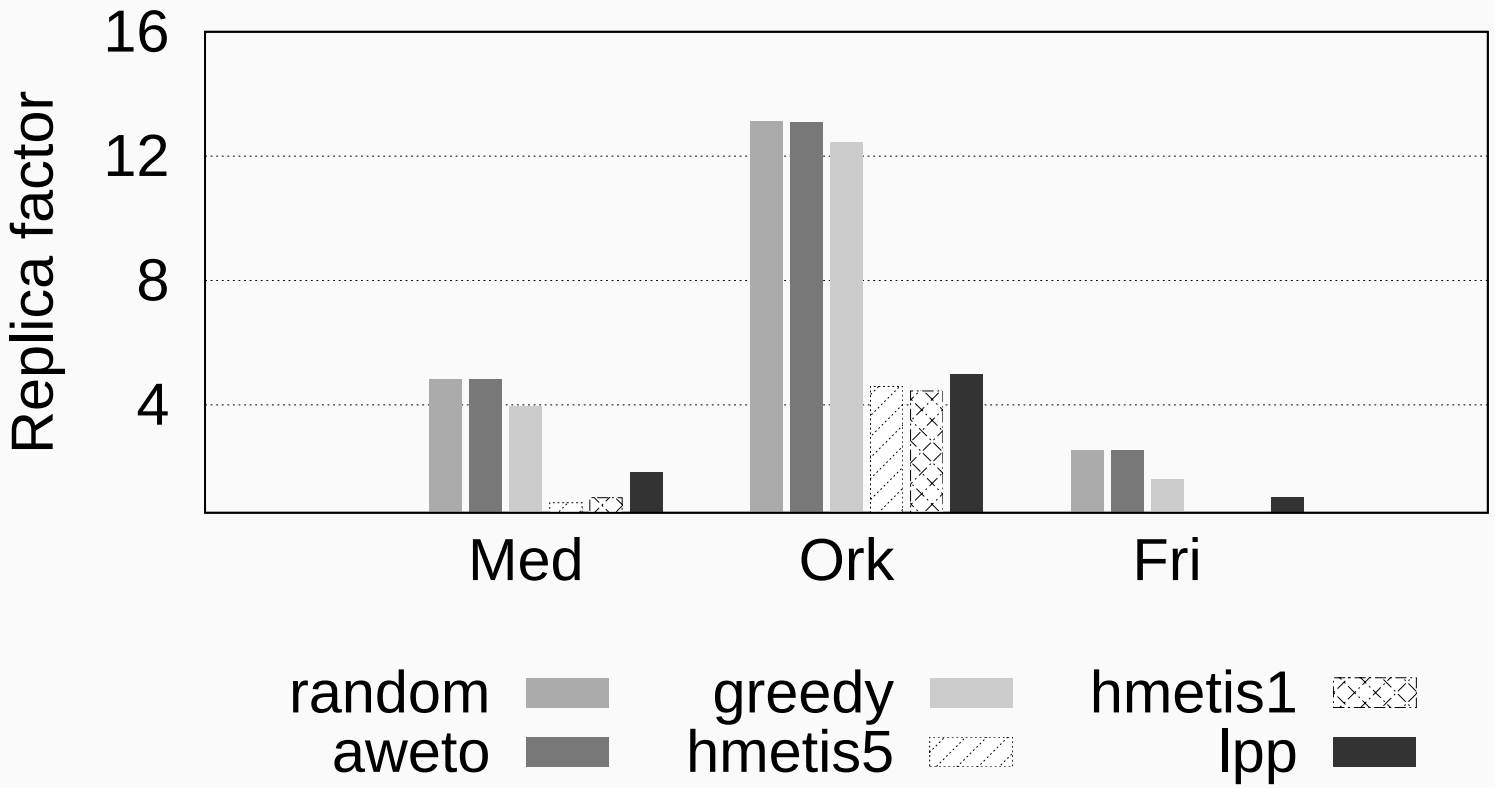


**Figure 8:** Different partitioning algorithms, replication factor

HyperX produces 1.1 to 1.9 times more replicas than hMetis.

# EVALUATING PARTITIONING EFFECTIVENESS: LOAD BALANCE

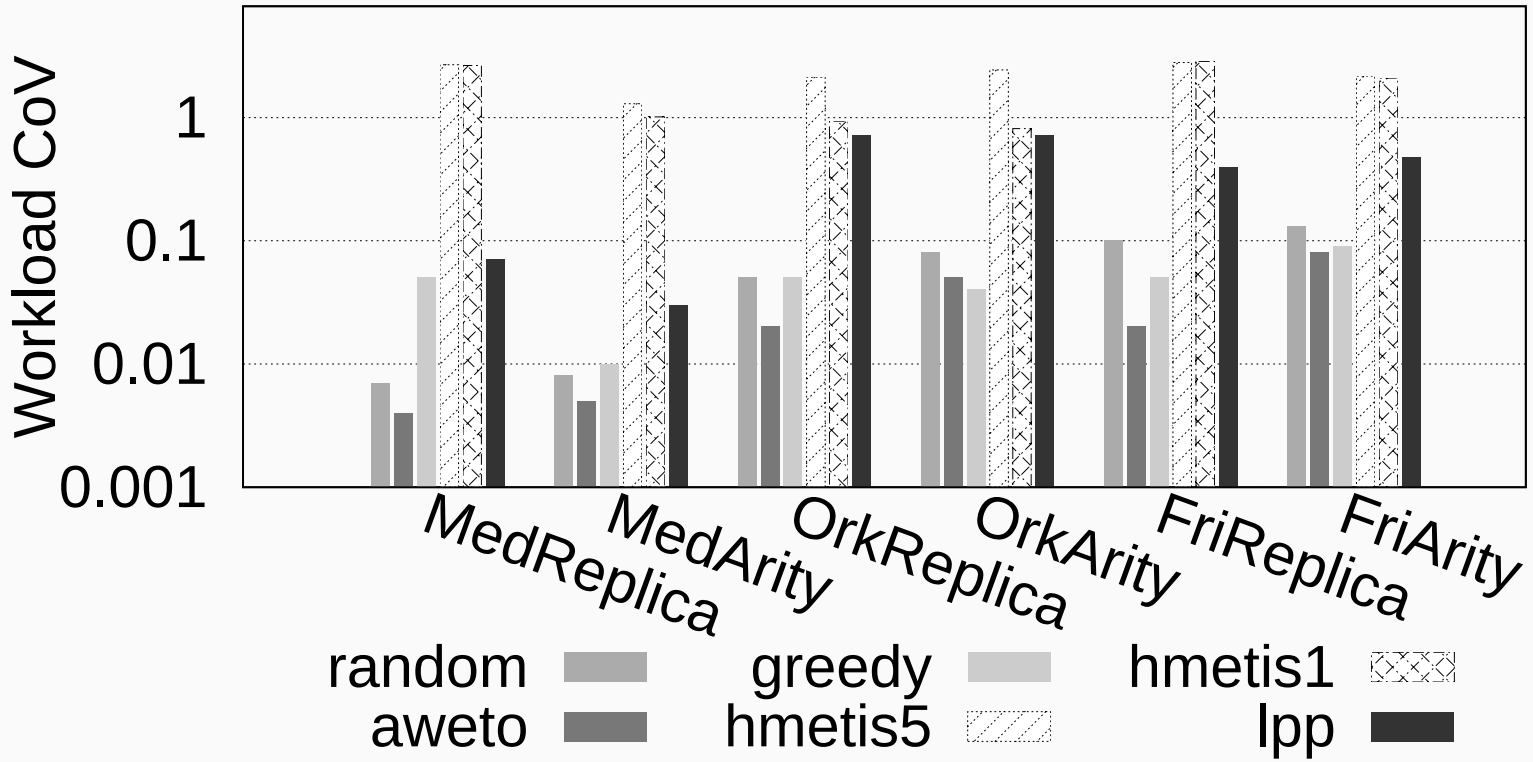


**Figure 9:** Different partitioning algorithms, load balance

LPP prodcues 1.1 to 37.7 times more balanced loads than hMetis.

# EVALUATING PARTITIONING EFFECTIVENESS: SPACE

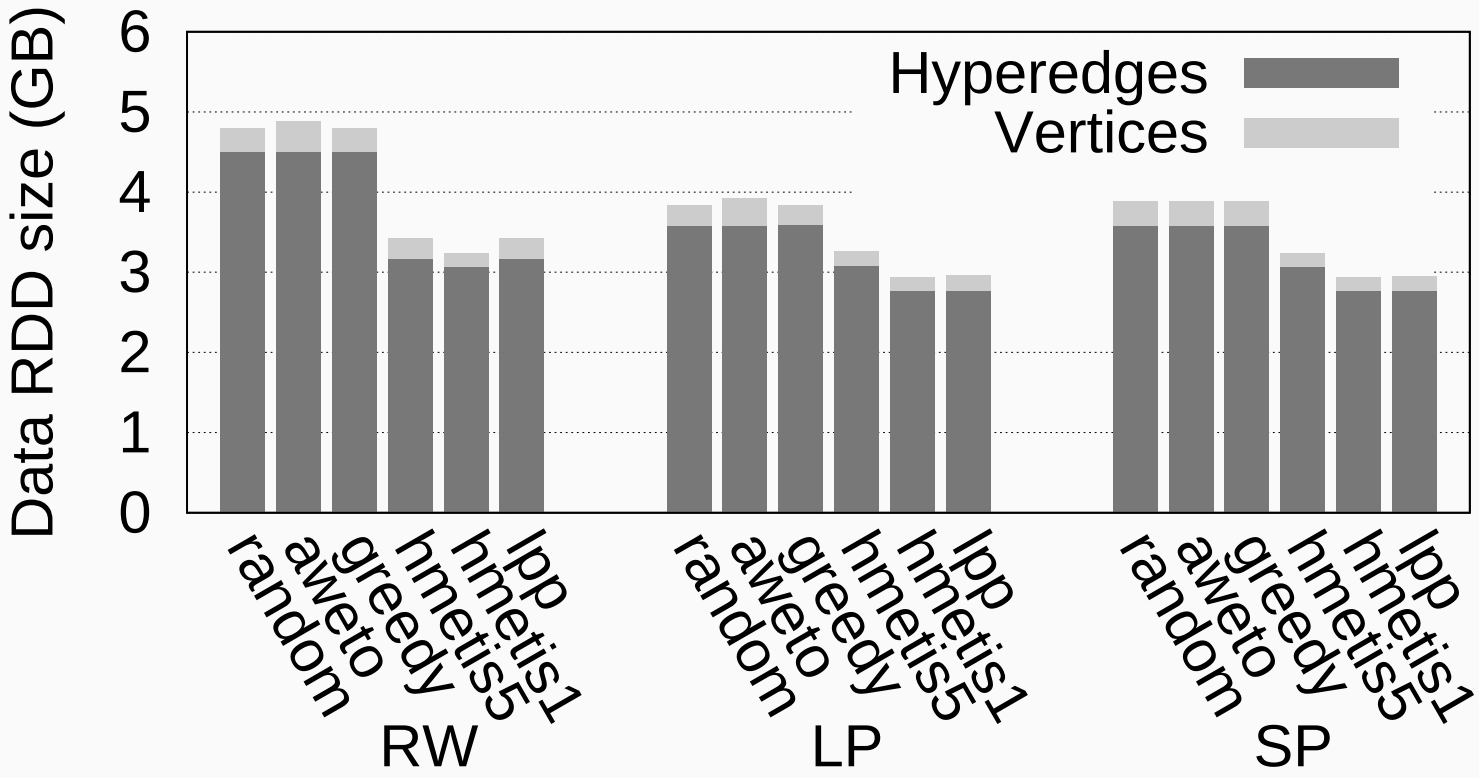


**Figure 10:** Different partitioning algorithms on Orkut, space

LPP and hMetis both outperform simplistic methods.

# EVALUATING PARTITIONING EFFECTIVENESS: COMMUNICATION

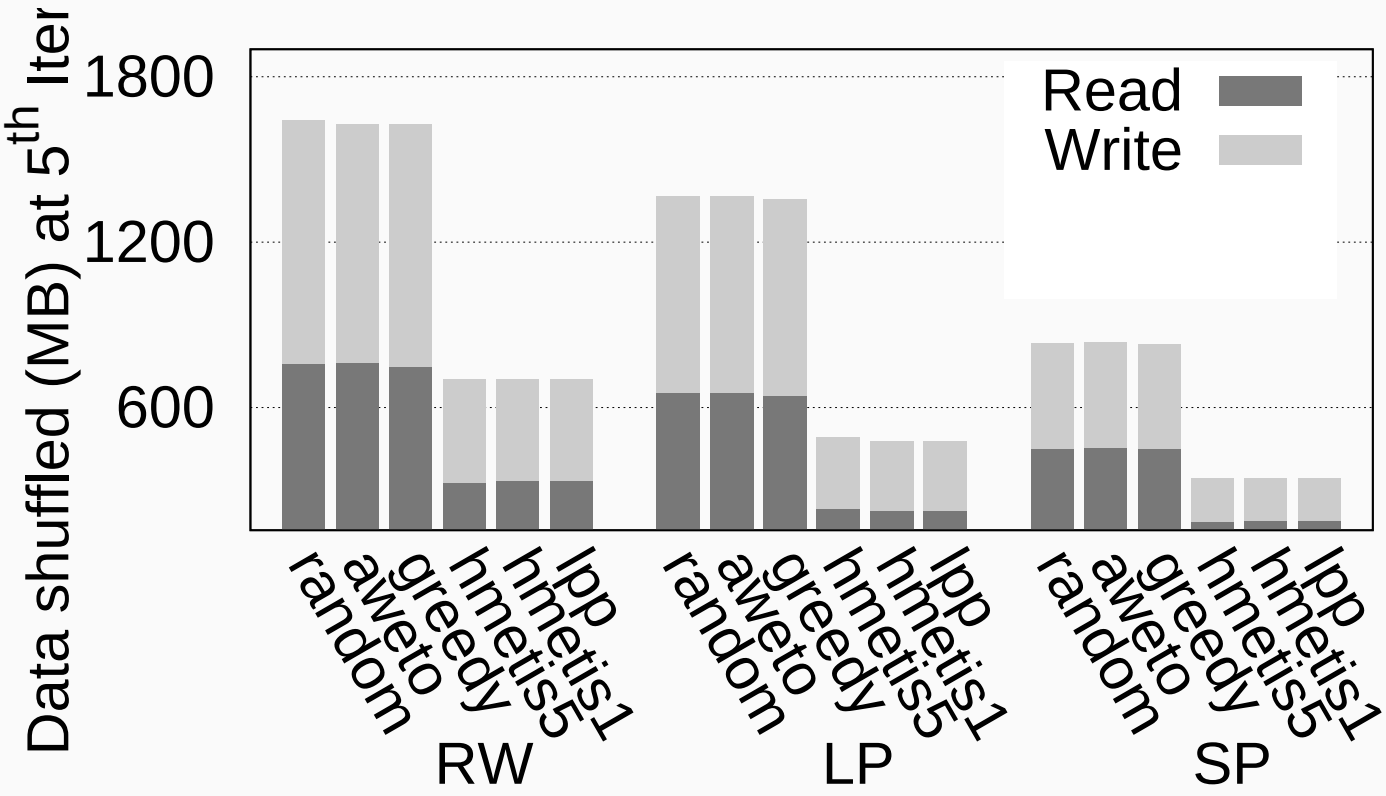


**Figure 11:** Different partitioning algorithms on Orkut, communication

LPP and hMetis both significantly outperform simplistic methods.

# EVALUATING PARTITIONING EFFECTIVENESS: TIME

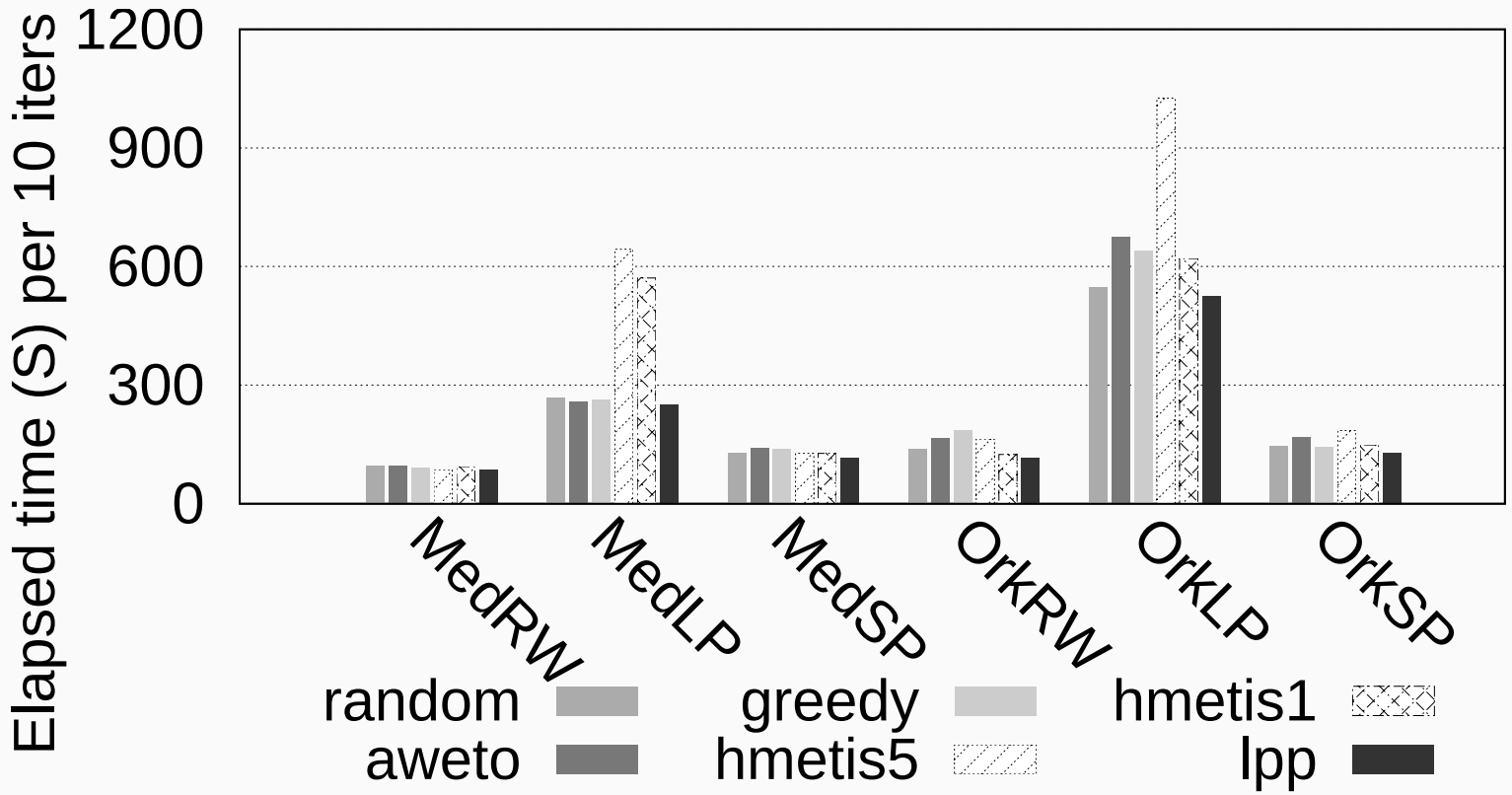


**Figure 12:** Different partitioning algorithms, time

LPP results to up to 2.6 times speedup over hMetis.

LPP in Scala, run on JVM; hMetis in C

**Table 4:** Partitioning time of different algorithms

| Dataset | Algorithm | Time $t$ (s) | $w$ | w.r.t. LPP |
|---|---|---|---|---|
| Med | LPP | 356 | 28 | 1.0 |
| | hMetis5 | 14,796 | 1 | 1.5 |
| Ork | LPP | 753 | 28 | 1.0 |
| | hMetis5 | 88,936 | 1 | 4.2 |
| Fri | LPP | 248 | 28 | 1.0 |
| | hMetis5 | 6,766 | 1 | 1.0 |

# EVALUATING LEARNING ALGORITHMS: DATASET CARDINALITY

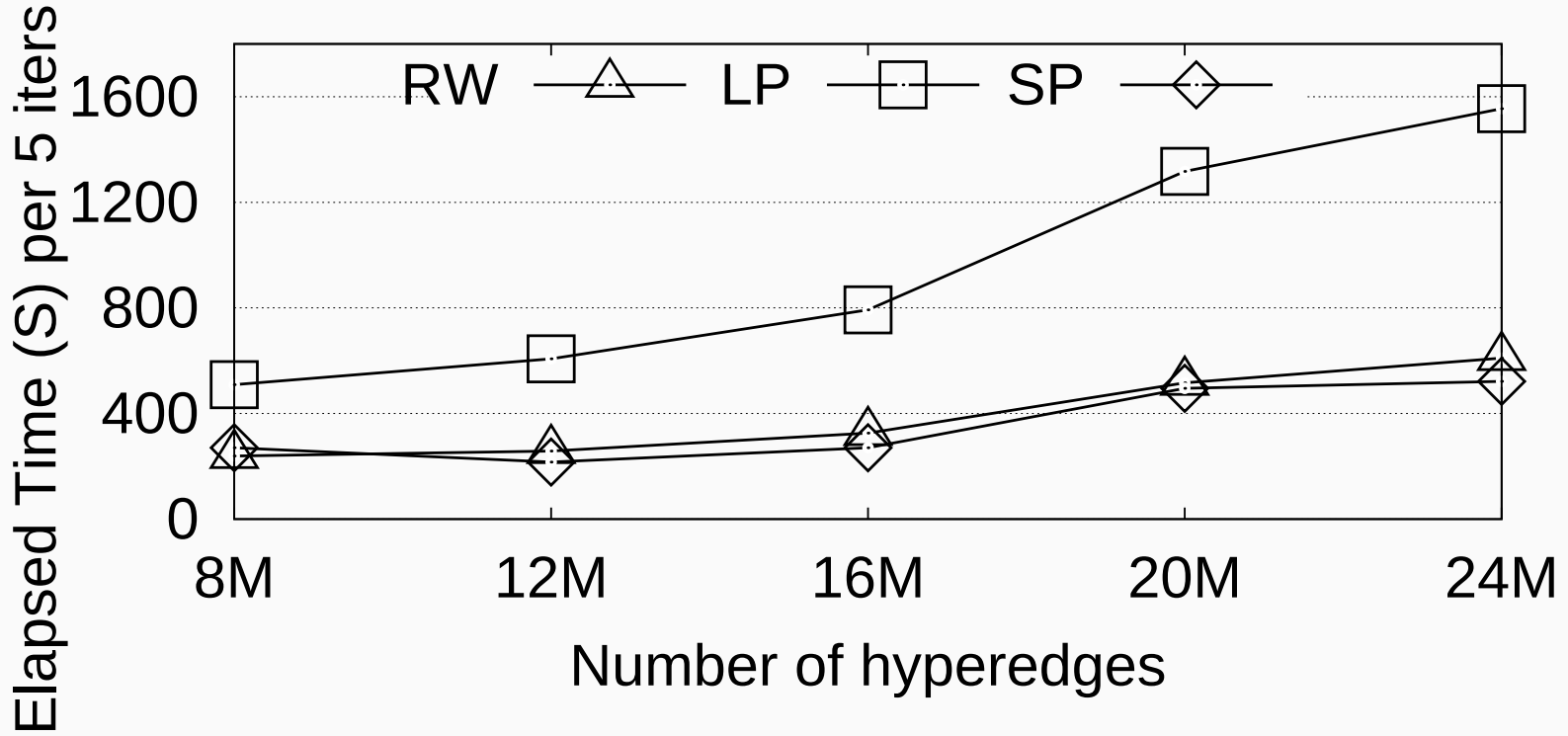


**Figure 13:** Elapsed time running algorithms on varying dataset cardinality, synthetic

# EVALUATING LEARNING ALGORITHMS: NUMBER OF WORKERS

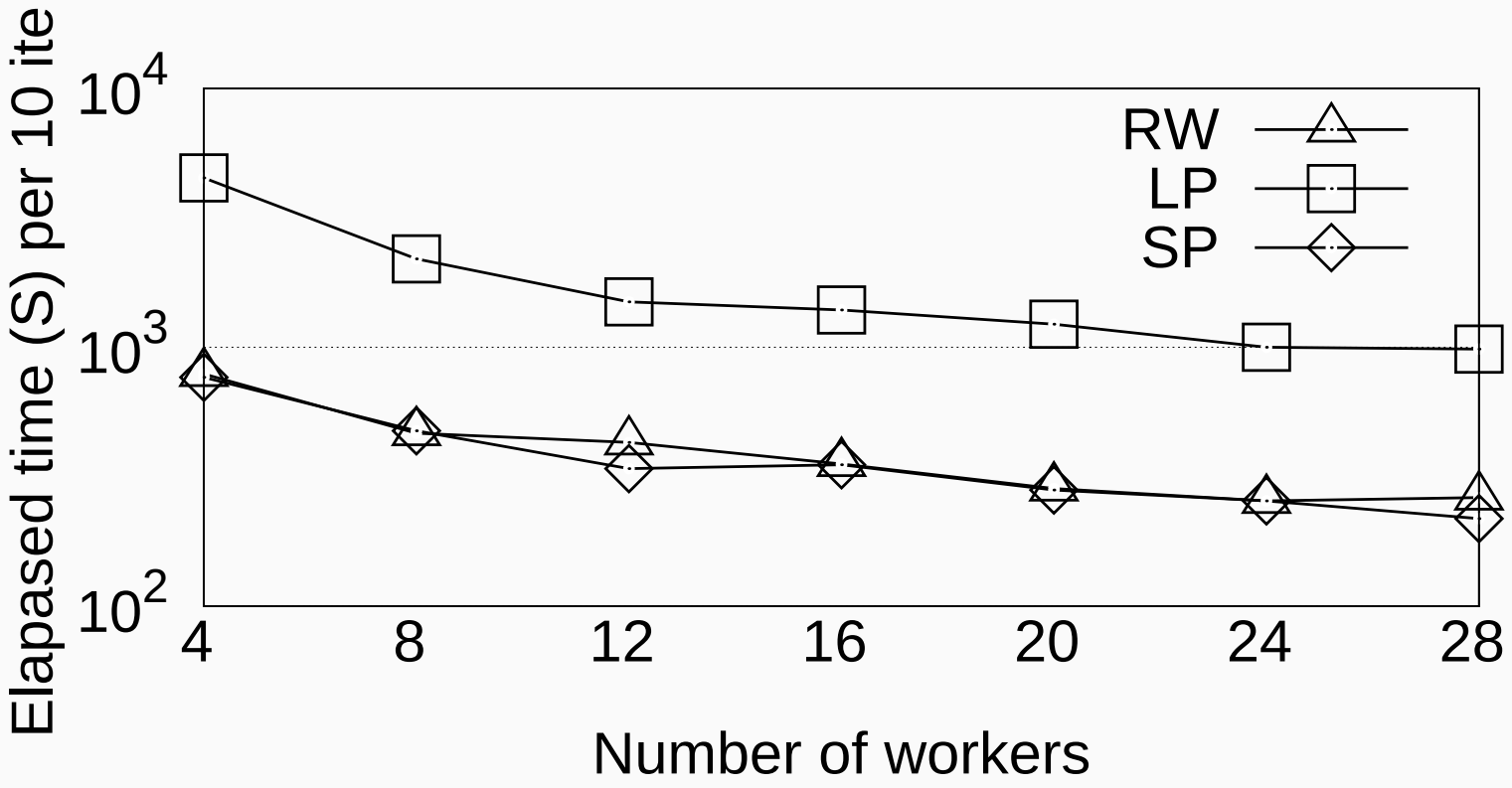


**Figure 14:** Elapsed time running algorithms on varying number of workers, Orkut

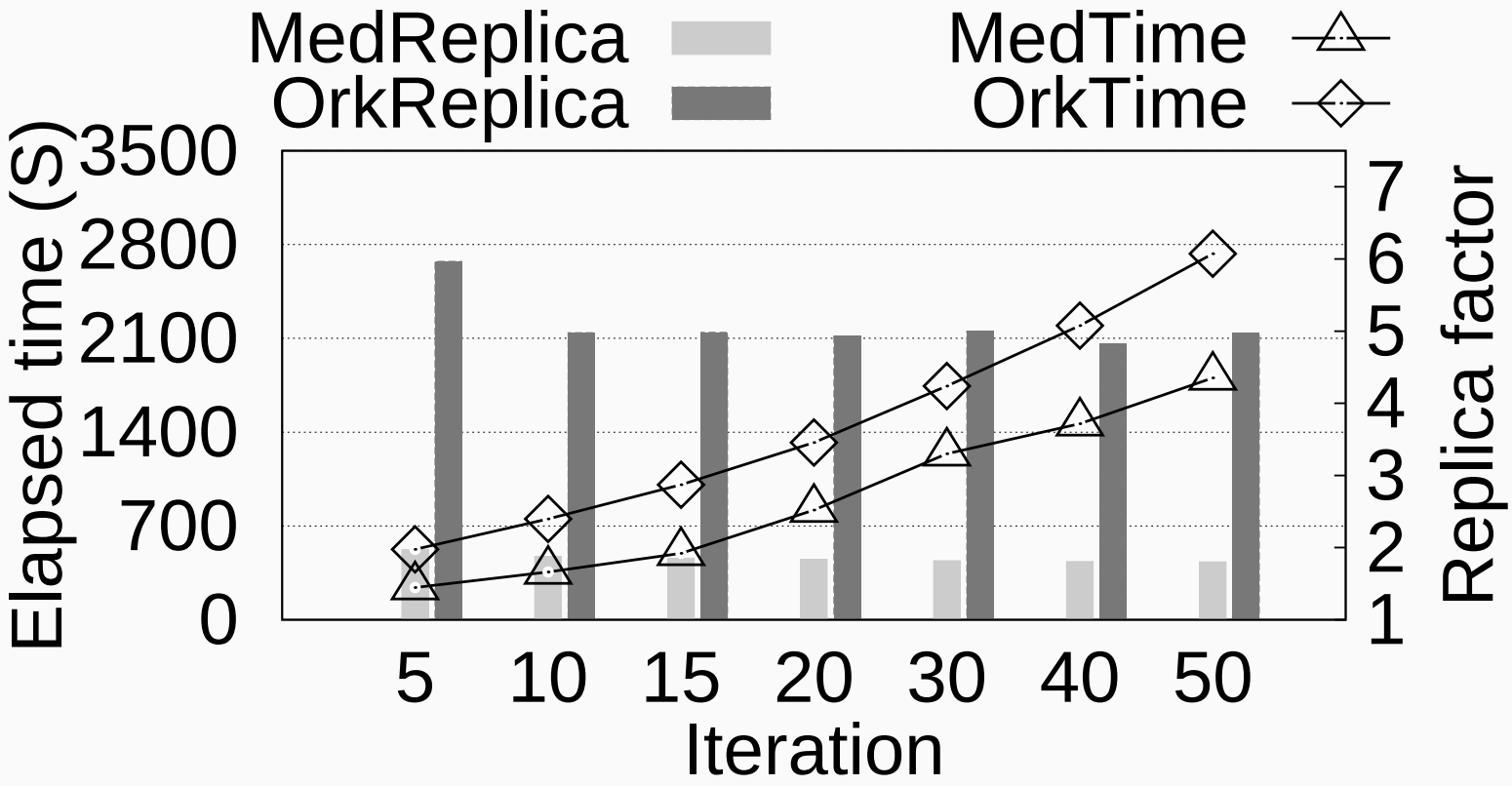


**Figure 15:** Elapsed time and replication factor

It only takes LPP a few iteration to achieve reasonable replication ratio.

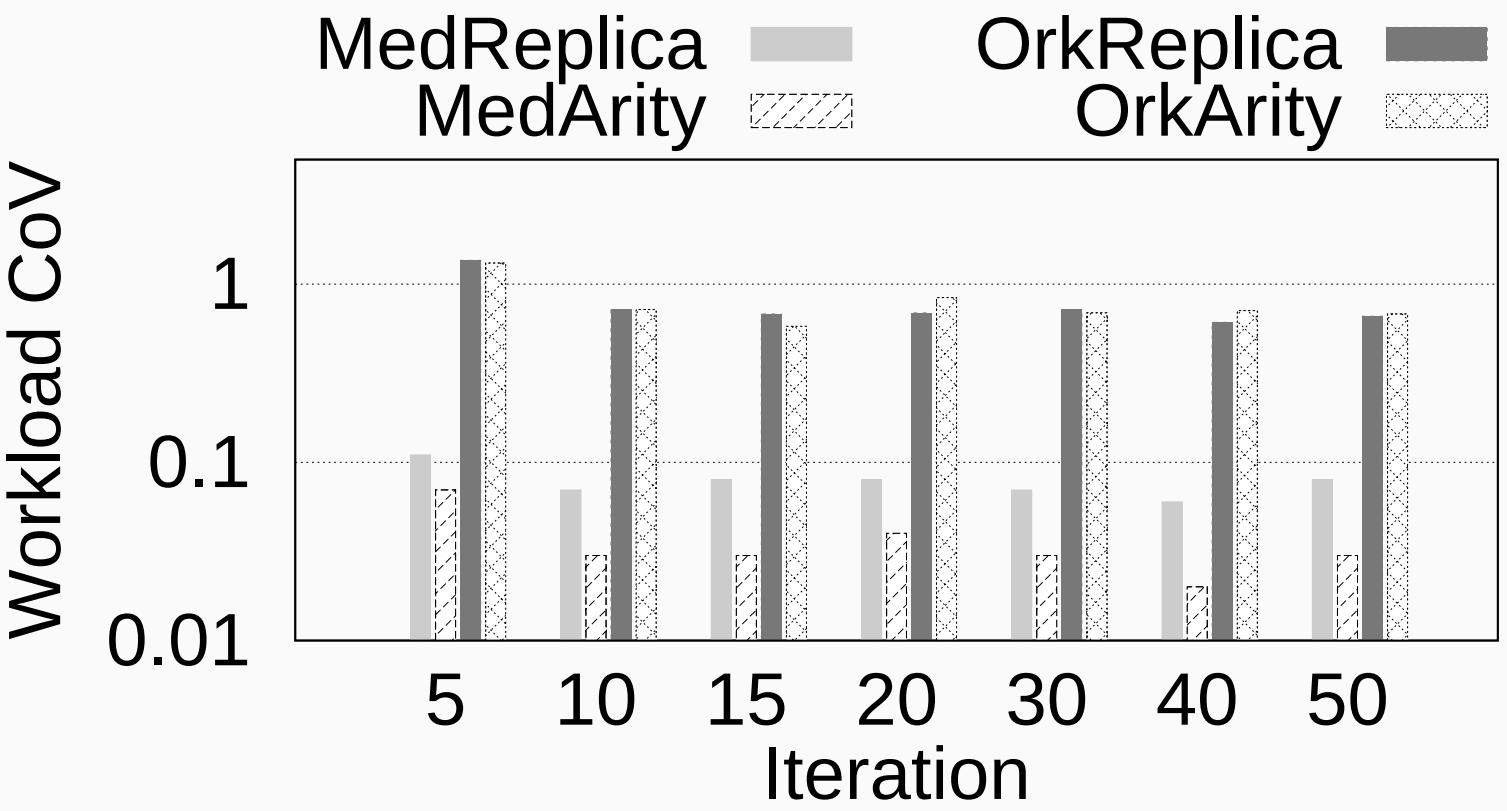


**Figure 16:** Elapsed time and replication factor

It only takes LPP a few iteration to achieve reasonable load balance.

**Problem** Scalable hypergraph learning

**Challenges**

1. Inflated problem size
2. Excessive replication
3. Great difficulty in balancing the loads

**Solutions**

1. Operate on a distributed hypergraph
2. Replicate only vertices
3. Partitioning optimization

**Contribution**

- Efficient and scalable hypergraph framework
- Effective and efficient partitioning algorithm

Thanks!

Any Questions or Comments?