

Jointly Optimizing Task Granularity and Concurrency for In-Memory MapReduce Frameworks

Jonghyun Bae*§ Hakbeom Jang†§ Wenjing Jin* Jun Heo* Jaeyoung Jang†
Joo-Young Hwang‡ Sangyeun Cho‡ Jae W. Lee*



*Seoul National University



†Sungkyunkwan University

SAMSUNG

‡Samsung Electronics Co.

§ Equal contributions

December 12th, 2017

Big Data 2017, Boston, MA, USA

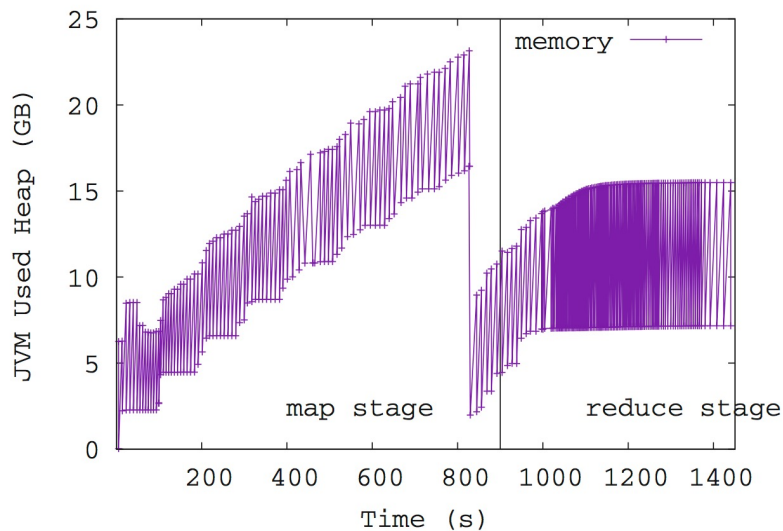
In-memory Big Data Processing (1)

- **Embraced by big data analytics frameworks**
 - Examples: Apache Spark (MapReduce for iterative algorithms), Ignite (Distributed SQL), SAP HANA (In-memory DB), etc.
- **Benefits of eliminating expensive I/Os**
 - High-throughput batch processing
 - Low-latency interactive data analytics

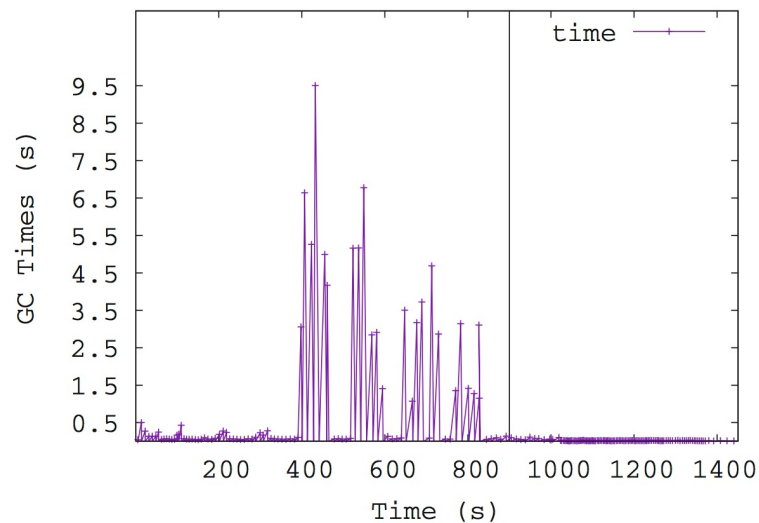


In-memory Big Data Processing (2)

- **To yield best performance memory usage must be carefully tuned**
 - Example: Spark heap usage and GC time*
- **Two major sources of overhead**
 - Spills: Serializing and storing heap objects to the local disk
 - Garbage collection (GC): Reclaiming memory occupied by orphaned objects



JVM heap usage

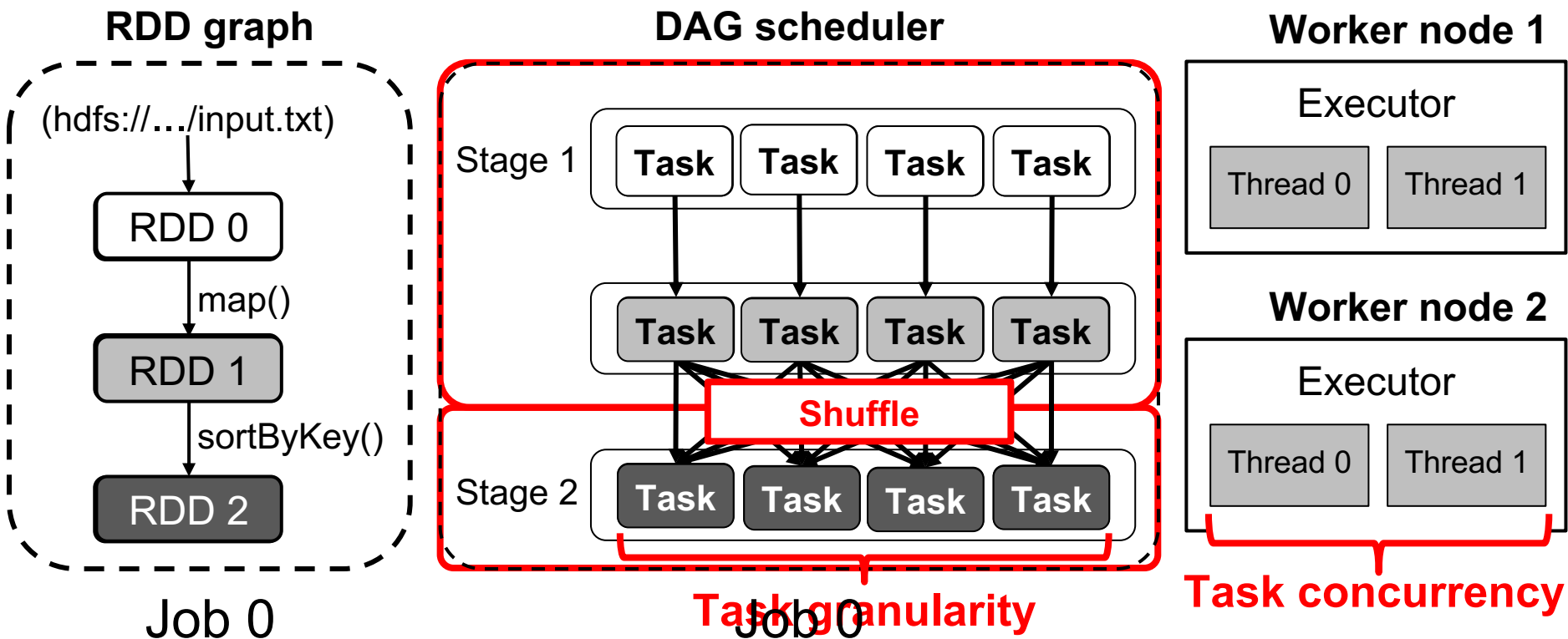


GC time

* [ICA3PP'15] C. Pei, et al., "Improving the Memory Efficiency of In-Memory MapReduce based HPC Systems"

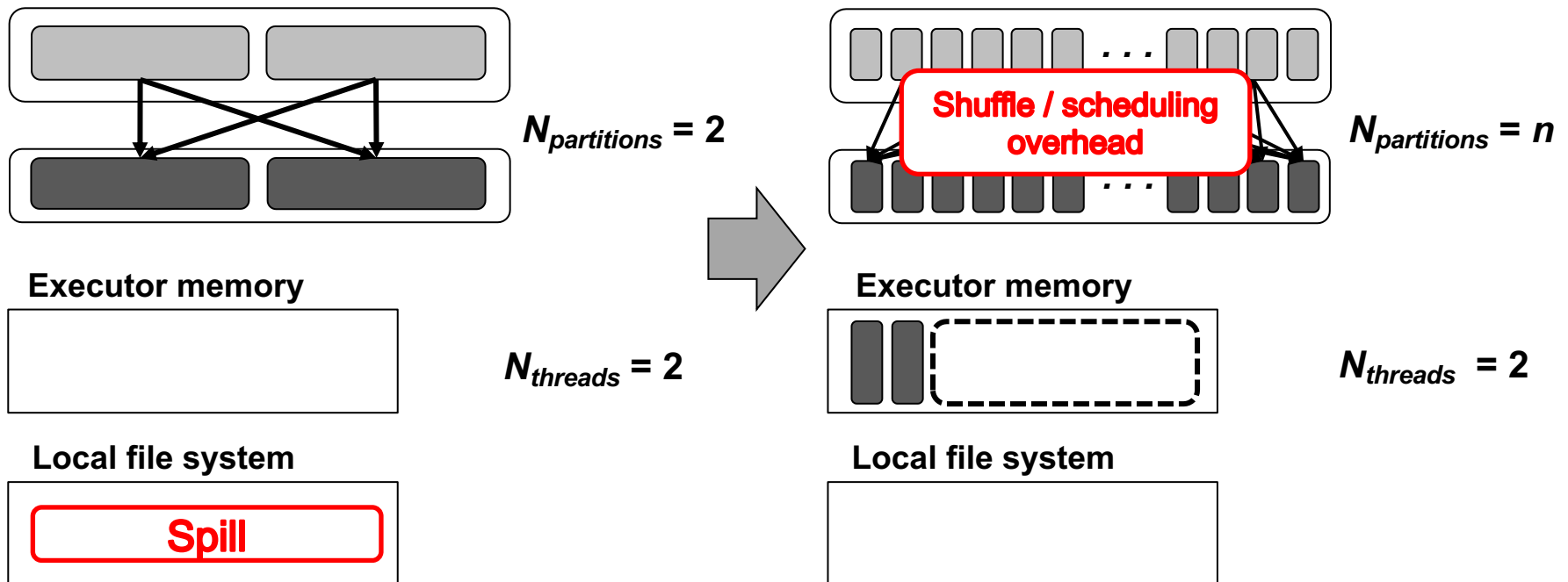
Example: Apache Spark Execution Flow

- **Job:** Created whenever an action is invoked in user program
- **Stage:** A job is split into multiple stages at every shuffle boundary
- **Task:** Each stage has a set of tasks, processing different RDD partitions



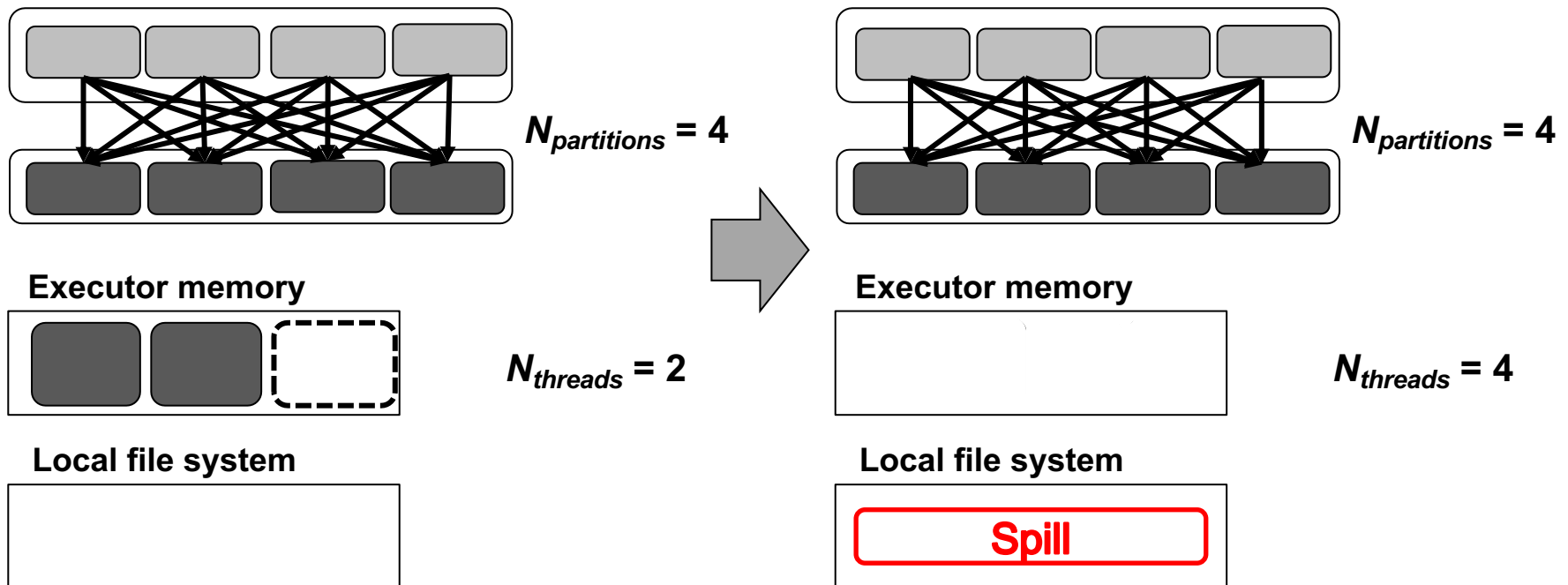
Two Key Parameters Controlling Memory Usage (1)

- **Parameter #1: Task granularity ($N_{partitions}$)**
 - Determines how many data partitions are created from a single RDD
 - $N_{partitions}$ set too low \rightarrow Causes excessive spills and GCs
 - $N_{partitions}$ set too high \rightarrow Incurs overhead for scheduling and shuffle operations



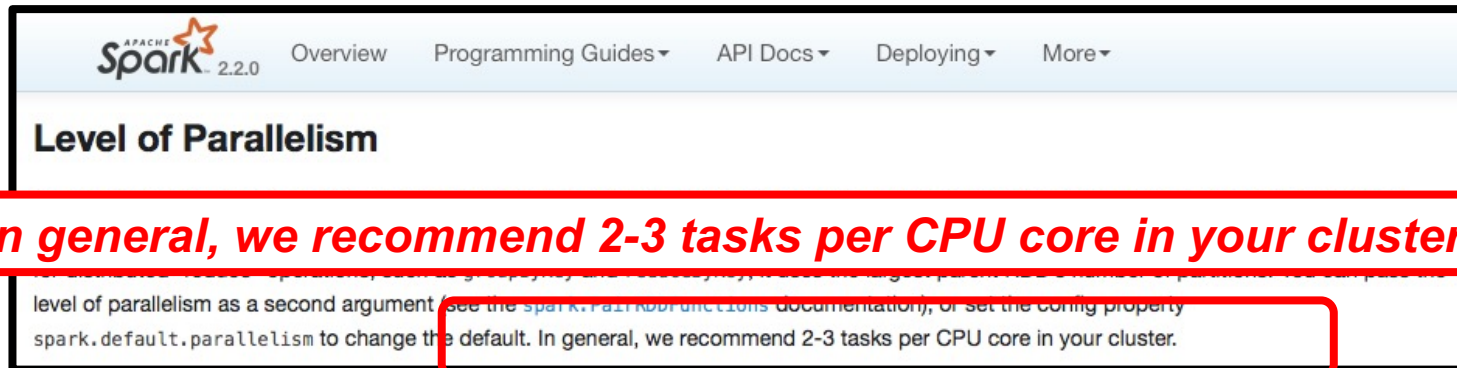
Two Key Parameters Controlling Memory Usage (2)

- **Parameter #2: Task concurrency ($N_{threads}$)**
 - Determines how many threads are allocated to a single executor
 - $N_{threads}$ set too low \rightarrow Yields suboptimal performance due to underutilization
 - $N_{threads}$ set too high \rightarrow Degrades performance due to resource contentions



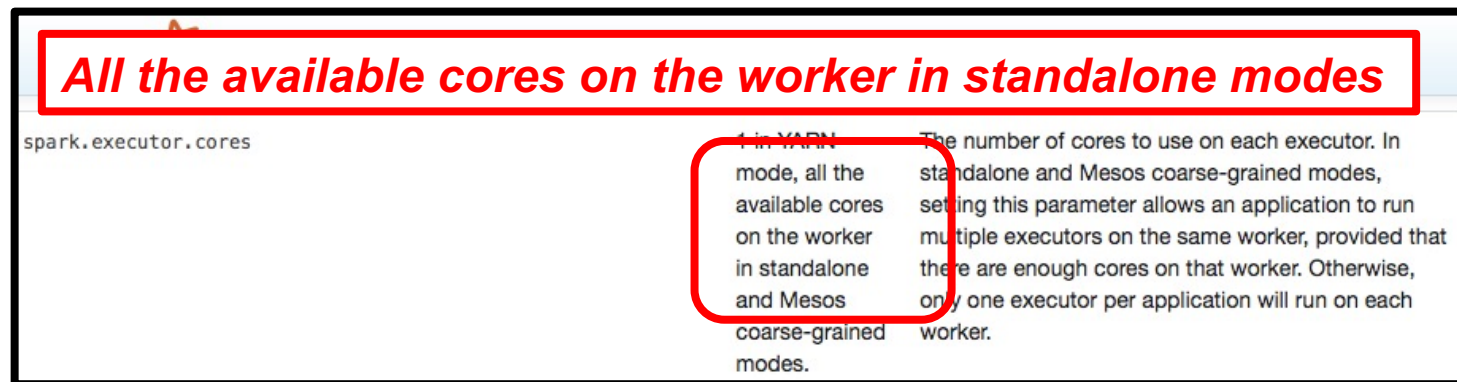
Two Key Parameters Controlling Memory Usage (3)

- Spark performance tuning guidelines*
 - $N_{partitions}$: “`spark.default.parallelism`”



The screenshot shows the Apache Spark 2.2.0 documentation page for "Level of Parallelism". The page title is "Level of Parallelism". A red box highlights the text: "In general, we recommend 2-3 tasks per CPU core in your cluster". Below this, the text says: "level of parallelism as a second argument (see the `spark.parallelism` documentation), or set the config property `spark.default.parallelism` to change the default. In general, we recommend 2-3 tasks per CPU core in your cluster."

- $N_{threads}$: “`spark.executor.cores`”

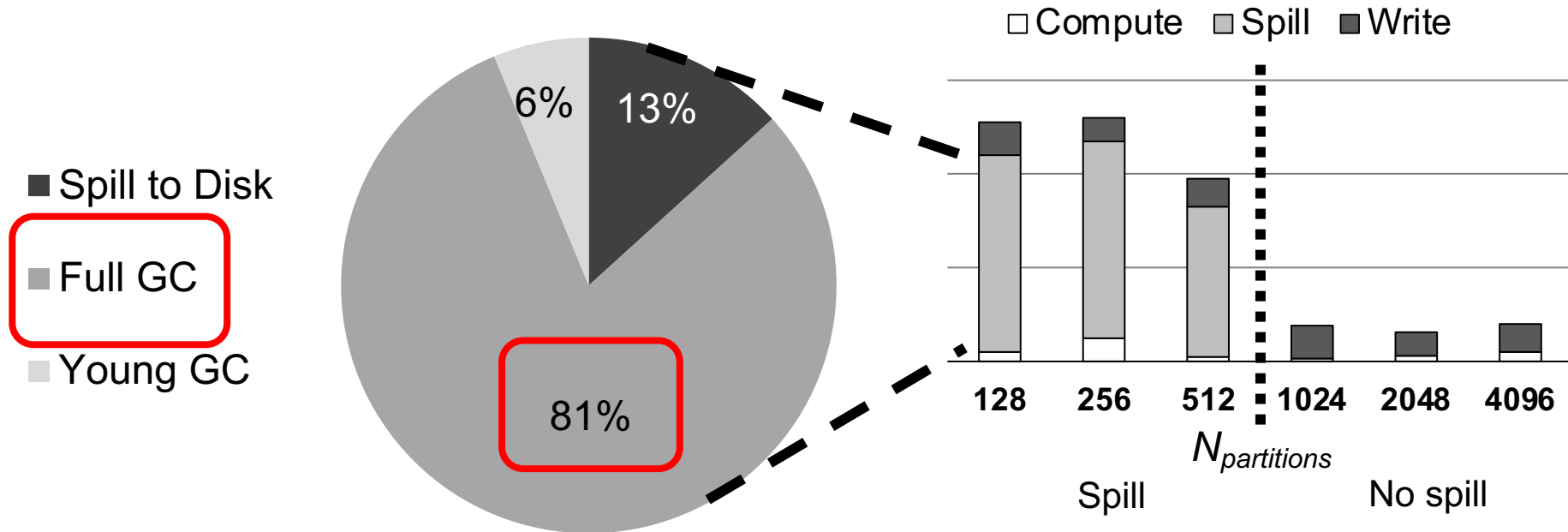


The screenshot shows the Apache Spark 2.2.0 documentation page for the configuration property `spark.executor.cores`. A red box highlights the text: "All the available cores on the worker in standalone modes". Below this, the text says: "1 in YARN mode, all the available cores on the worker in standalone and Mesos coarse-grained modes. The number of cores to use on each executor. In standalone and Mesos coarse-grained modes, setting this parameter allows an application to run multiple executors on the same worker, provided that there are enough cores on that worker. Otherwise, only one executor per application will run on each worker."

* <http://spark.apache.org/docs/latest/index.html>

Two Key Parameters Controlling Memory Usage (4)

- $N_{partitions}$ and $N_{threads}$ must be jointly optimized
 - Exhaustive search - finding the optimal $N_{partitions}$ and $N_{threads}$



- TeraSort: Up to 3.64x ($\langle N_{partitions}, N_{threads} \rangle = \langle 4096, 6 \rangle$)
- Bayesian Classification: Up to 1.28x ($\langle N_{partitions}, N_{threads} \rangle = \langle 64, 8 \rangle$)

Our Proposal: Workload-Aware Scheduler and Partitioner (WASP)

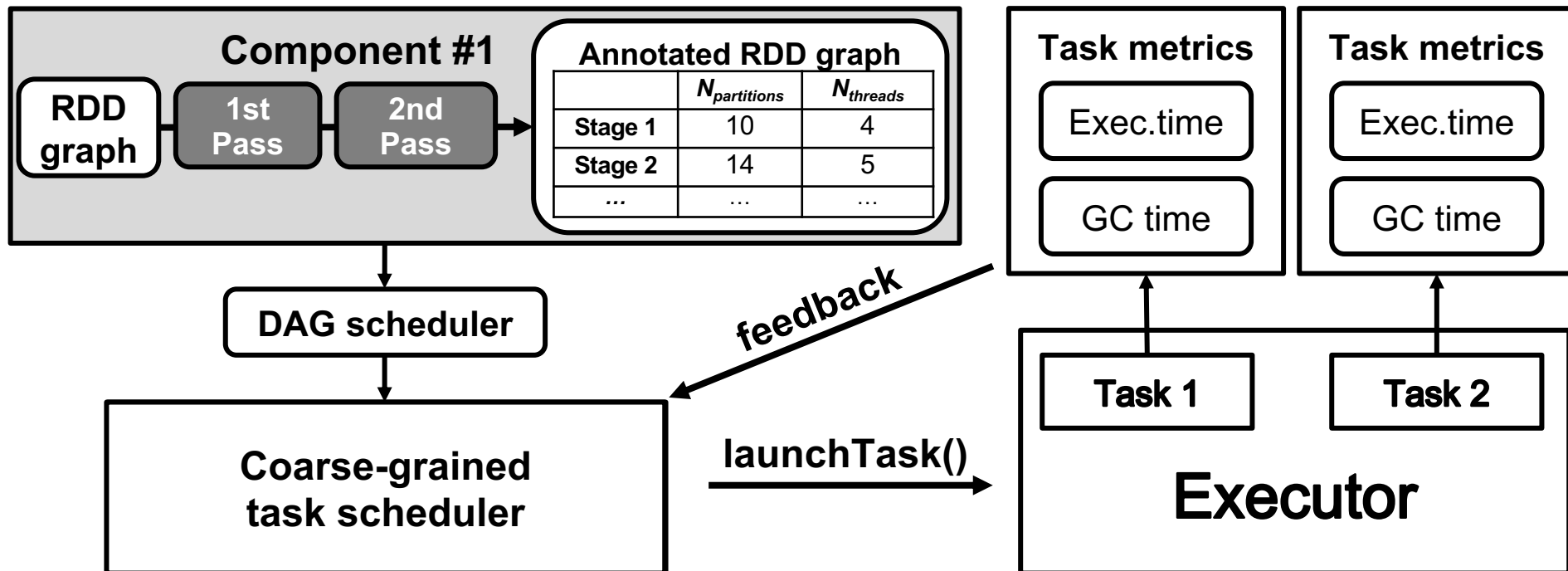
- **WASP: Runtime scheduler jointly optimizing $N_{partitions}$ and $N_{threads}$**
 - Goal: Maximizing concurrency without causing excessive spills and GCs
 - Consists of two components
- **Component #1: Analytical model**
 - Analyzes DAG and predicts optimal parameters for all stages at program launch
- **Component #2: Runtime scheduler**
 - Fine-tunes $N_{threads}$ using hill climbing algorithm at runtime
- **WASP achieves near-optimal performance for both shuffle-heavy and shuffle-light workloads.**

Outline

- **Motivation and Key Idea**
- **Workload-Aware Scheduler and Partitioner (WASP)**
- **Evaluation**
 - Methodology
 - Performance results
- **Summary**

Master node

Worker node



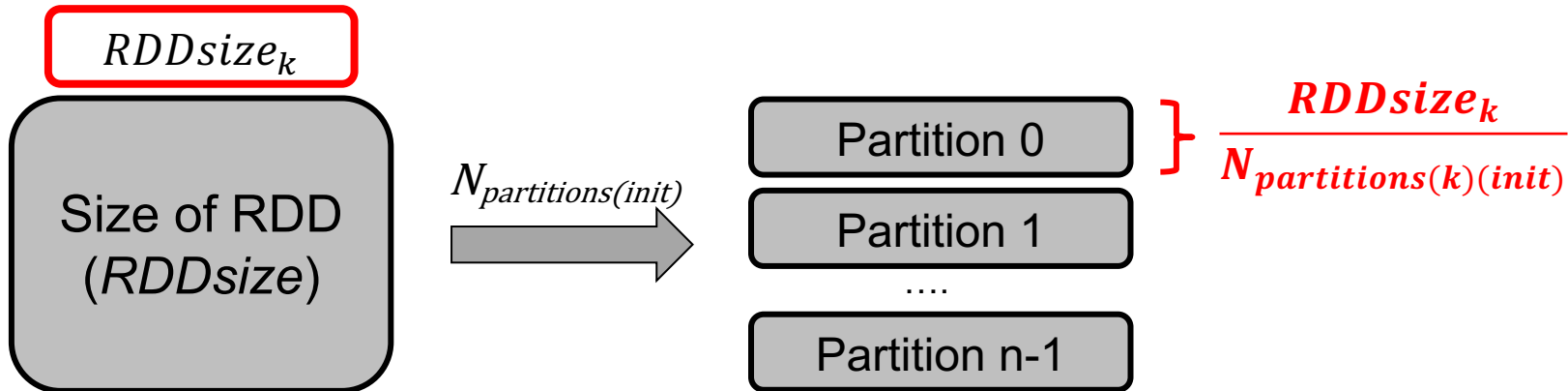
- **Component #1: Analytical model**
 - **First Pass:** Predicting initial $N_{partitions}$ and $N_{threads}$
 - **Second Pass:** Searching $N_{partitions}$ and $N_{threads}$ via gradient search
- **Component #2: Runtime scheduler**
 - GC-aware optimization of $N_{threads}$

Component #1: Analytical Model (1st Pass) – $N_{partitions}$ and $N_{threads}$

	Example value
<i>Execution memory</i>	8GB
$N_{threads}(init)$	12 physical cores*

$$N_{threads(k)}(init) = \max \left(\begin{array}{l} \text{Execution memory} \\ \text{RDDsize}_k \\ N_{partitions(k)}(init) \end{array}, 1 \right)$$

Missing parameter is $RDDsize_k$



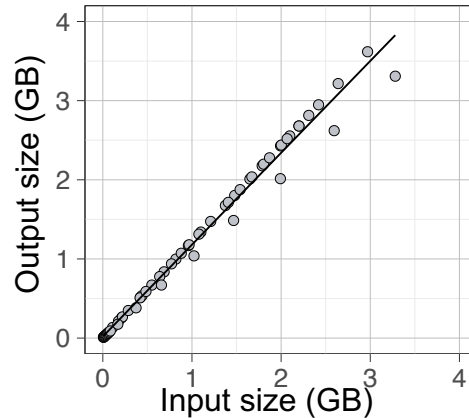
* [NSDI'15] K. Ousterhout, et al., "Making Sense of Performance in Data Analytics Frameworks"



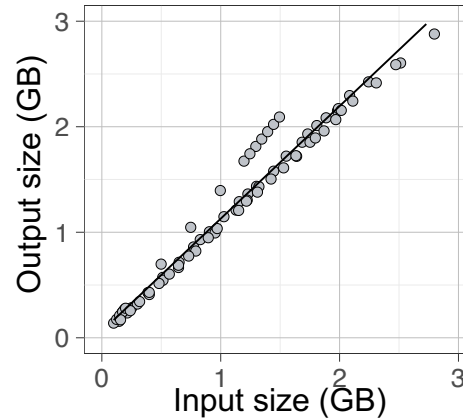
Component #1: Analytical Model (1st Pass) – Memory Amplification Factor

- Ratio of output RDD size to the input RDD size

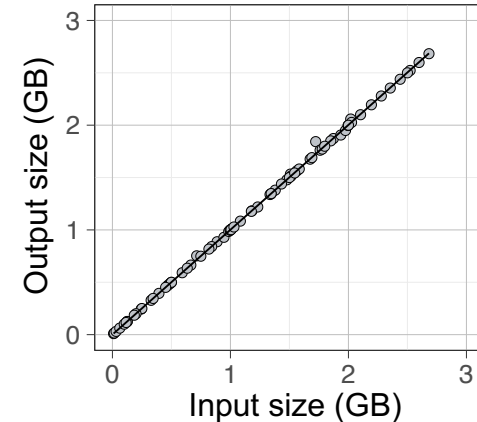
zip (1.161)



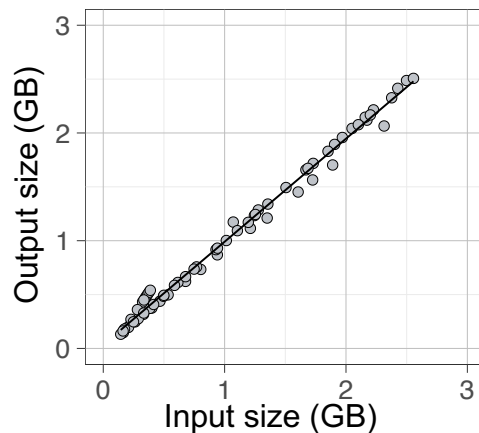
mapValue (1.067)



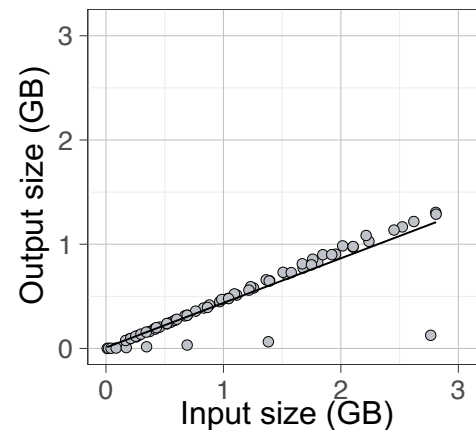
sortByKey (1.0)



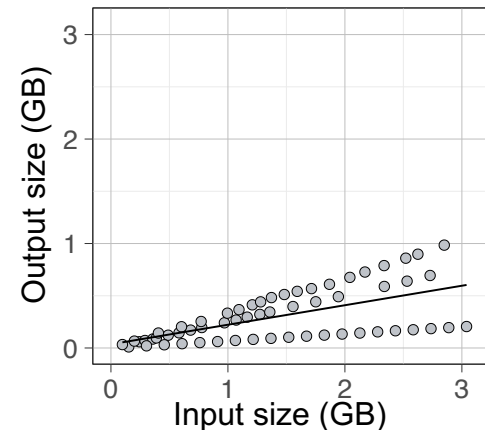
join (0.853)



groupByKey (0.428)



filter (0.187)



Component #1: Analytical Model (1st Pass) – RDDsize and Input size

	Example value
<i>Input size</i> ₁	800GB

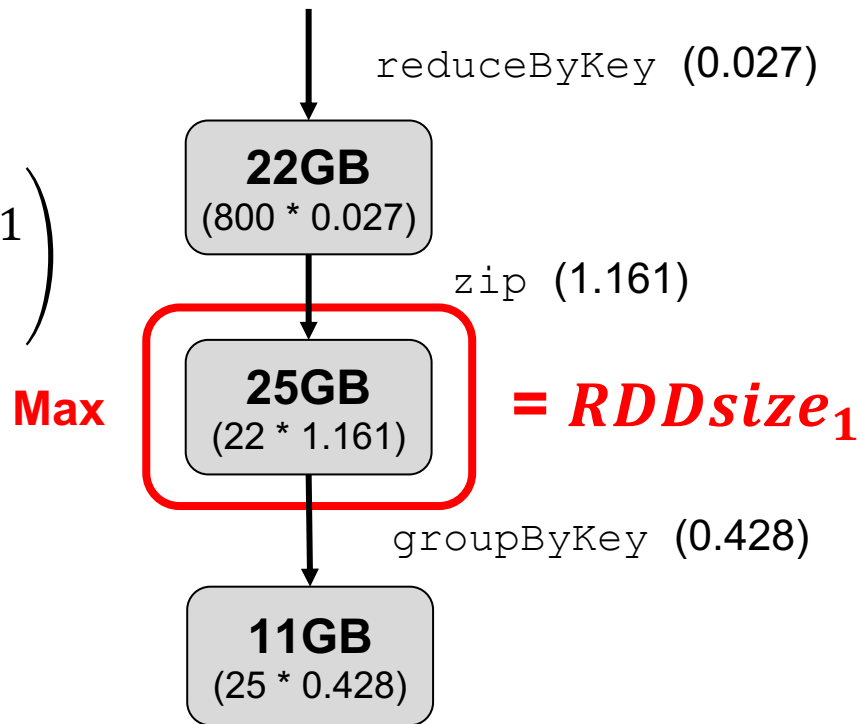
$$RDDsize_k = \text{Input size}_k \times \max_{0 \leq i \leq n_k} \left(\prod MAF(f_i^k) \right)$$

$$N_{threads(k)(init)} = \max \left(\max \left(\frac{\text{Execution memory}}{25 \text{ GB} \cdot RDDsize_k}, 1 \right), \frac{N_{partitions(1)(init)}}{N_{partitions(k)(init)}} \right)$$

↓

$$N_{partitions(1)(init)} = 36$$

*Input size*₁: 800GB

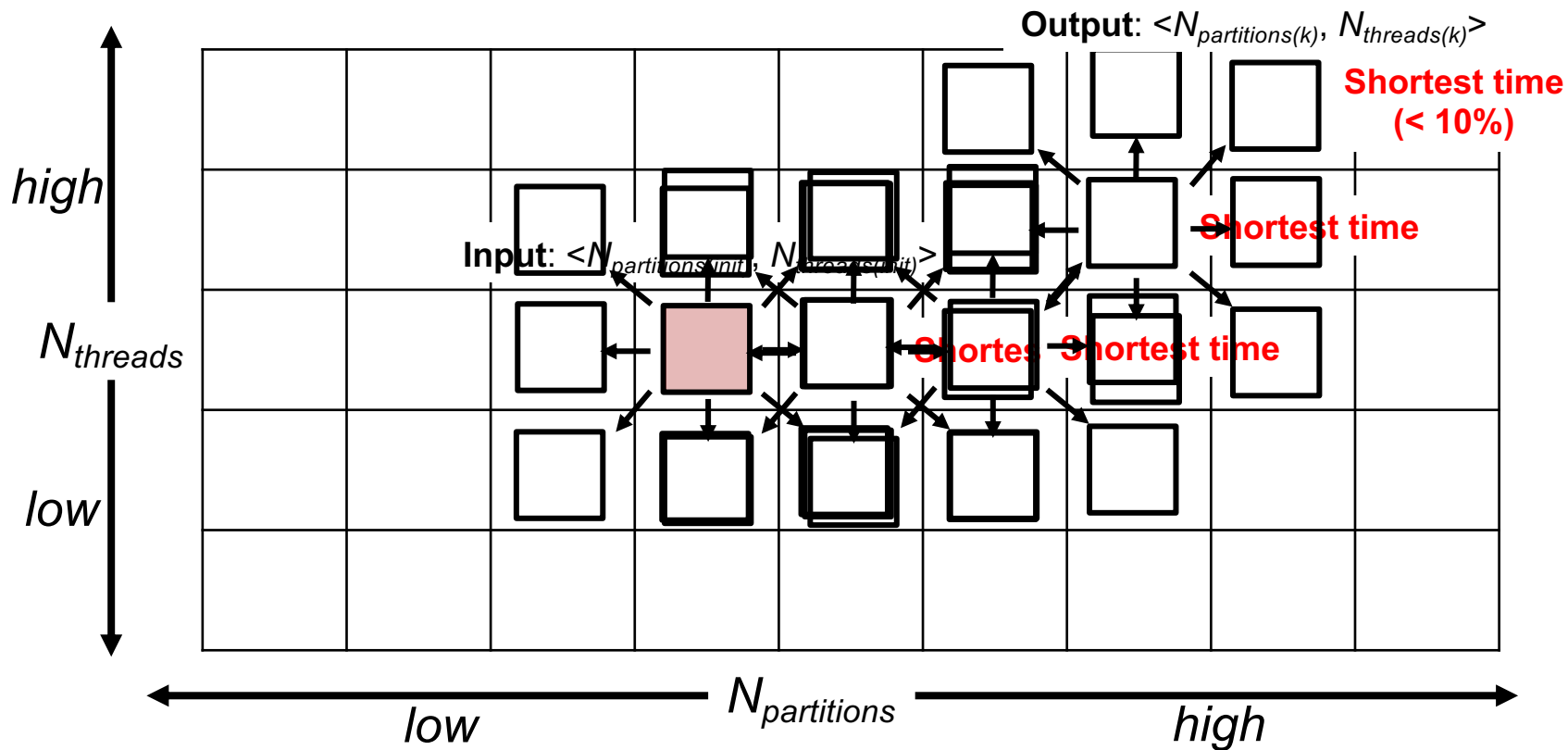


Component #1: Analytical Model (2nd Pass) – Gradient Search

- Estimating execution time and finding optimal $N_{partitions}$ and $N_{threads}$

$$ExecTime_k \propto \frac{RDDsize_k}{N_{partitions}(k)} \times \frac{\alpha_k}{N_{threads}(k)} \times \frac{N_{partitions}(k)}{N_{threads}(k)}$$

$$\alpha_k = \max \left(\frac{RDDsize_k}{N_{partitions}(k)} \times N_{threads}(k), \text{Execution memory} \right)$$



Component #2: Runtime Scheduler

- **Optimizing $N_{threads}$ at runtime using hill climbing algorithm**
 - Maximize the degree of parallelism without causing excessive spills and GCs

Annotated RDD graph

	$N_{partitions(k)}$	$N_{threads(k)}$
Stage 1	10	4
Stage 2	14	5
...

$$N_{partitions(1)} = 10$$

$$N_{threads(1)} = 4$$

$$N_{partitions(2)} = 14$$

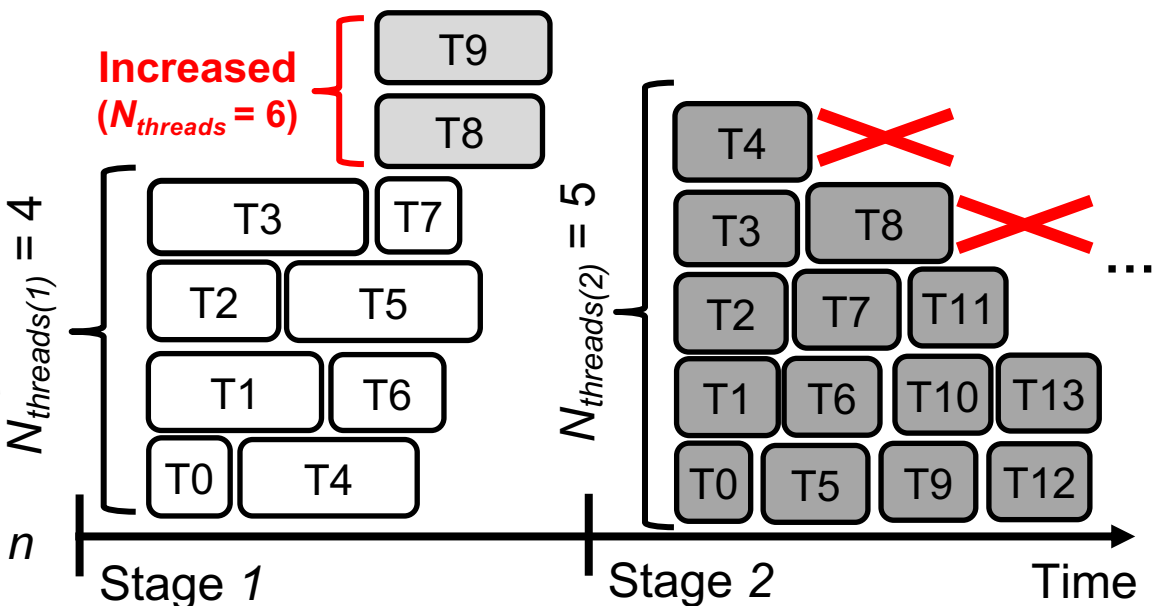
$$N_{threads(2)} = 5$$

GC-aware task scheduler

Update
freeCores

GC time & execution time

T_n : Task n



Outline

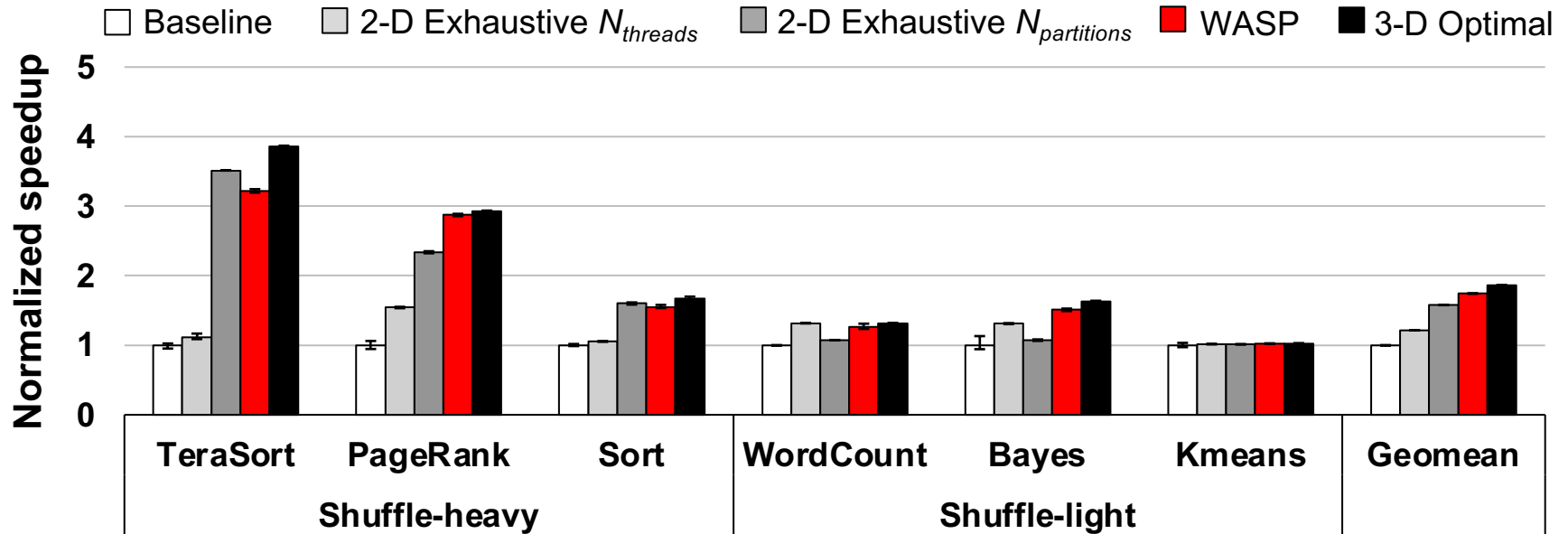
- **Motivation and Key Idea**
- **Workload-Aware Scheduler and Partitioner (WASP)**
- **Evaluation**
 - Methodology
 - Performance results
- **Summary**

Methodology: Evaluation Platforms

	Native Cluster	Intel Knights Landing (KNL)	Amazon EC2
Server	Dell R730 x 4 (+ 1 master)	SuperServer 5038k-i	m4.xlarge x 64 (+ 1 master)
CPU	Intel Xeon CPU E5-2640 v3 8 cores @ 2.60GHz x 2	Intel Xeon Phi (Knights Landing) 7210 64 cores @ 1.30GHz	Intel Xeon E5-2676 v4 4 cores @ 2.4GHz
Memory	16GB DDR4 x 8 (128GB in total)	16GB MCDRAM & 32GB DDR4 x 6	16GB

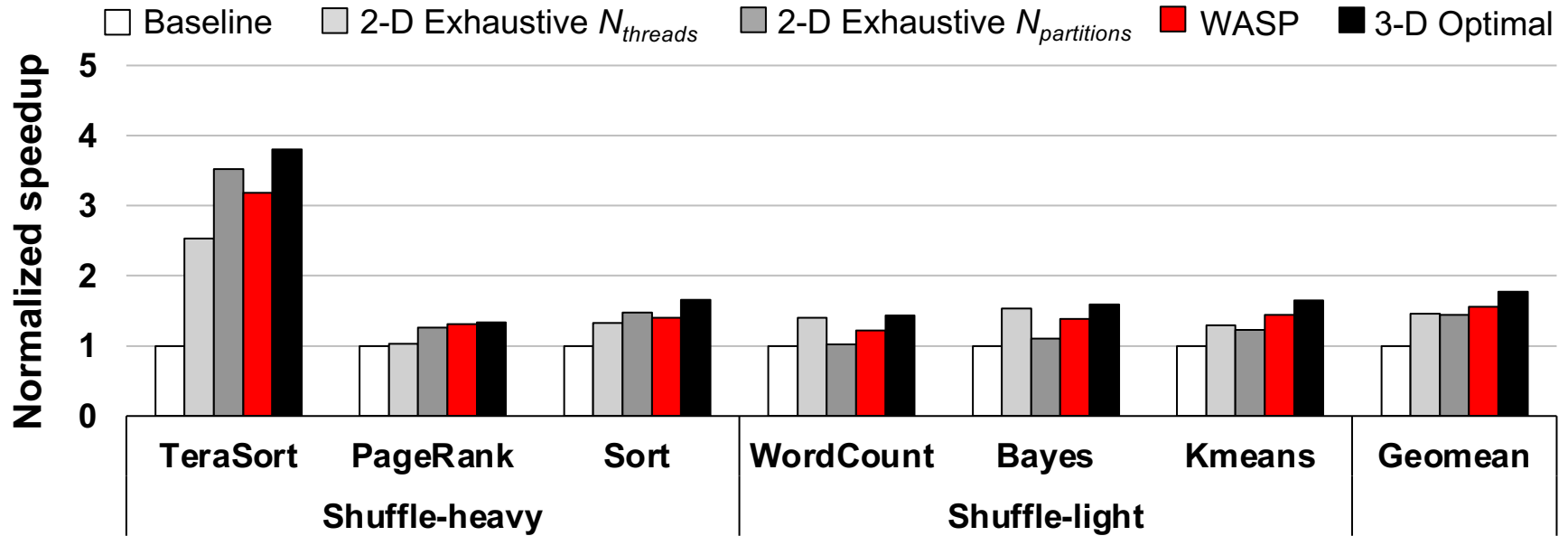
- **Comparing WASP to the following four designs**
 - Baseline: Out-of-the-box Spark following Spark tuning guidelines
 - 2-D Exhaustive $N_{threads}$: Two-dimensional exhaustive search (stage, $N_{threads}$)
 - 2-D Exhaustive $N_{partitions}$: Two-dimensional exhaustive search (stage, $N_{partitions}$)
 - 3-D Optimal: Three-dimensional exhaustive search (stage, $N_{partitions}$, $N_{threads}$)
- **Intel HiBench workloads with different characteristics**
 - Shuffle-heavy: TeraSort, PageRank, Sort
 - Shuffle-light: WordCount, Bayesian Classification, Kmeans

Overall Speedups: Performance on Native Cluster



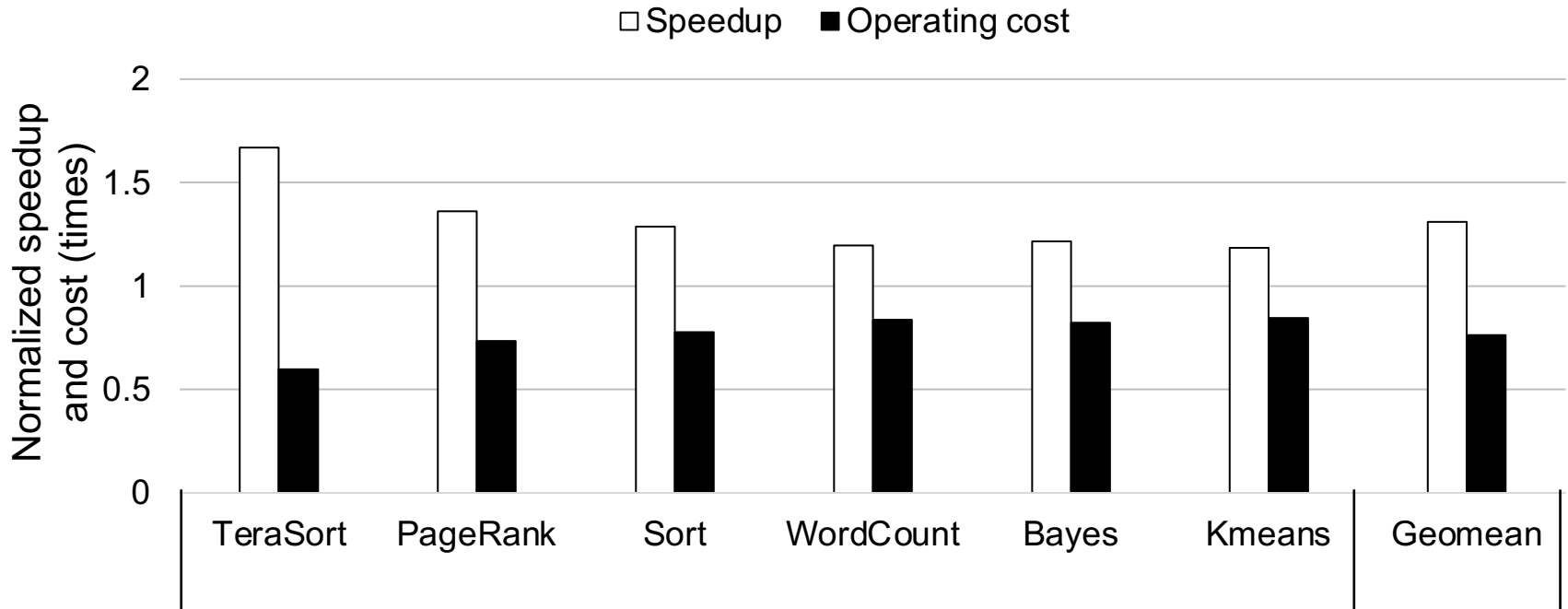
- Geomean speedup: **1.74x**
- Maximum speedup: **3.22x** for TeraSort
- Compared to 3-D Optimal: falling within **6%**

Overall Speedups: Performance on Knights Landing



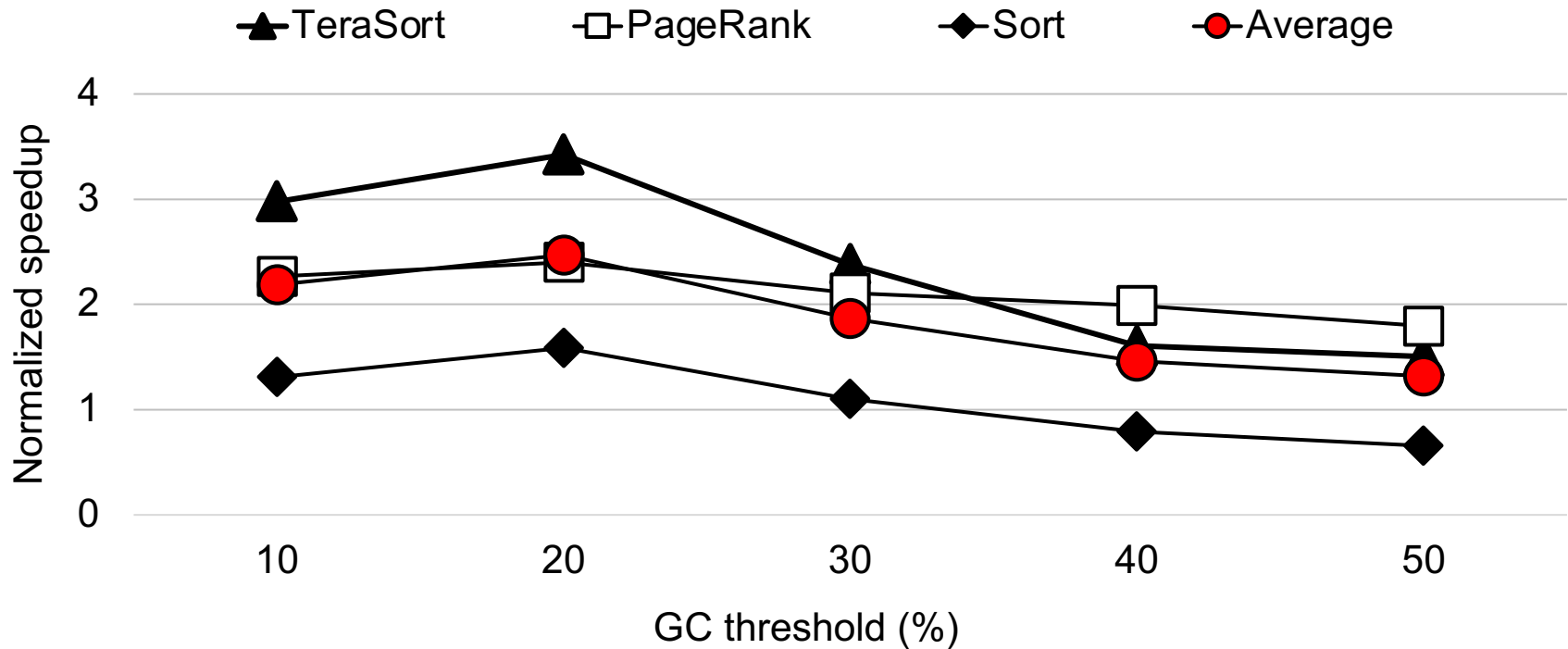
- Geomean speedup: **1.56x**
- Maximum speedup: **3.18x** for TeraSort
- Compared to 3-D Optimal: falling within **12%**

Overall Speedups: Performance on Amazon EC2 Cluster



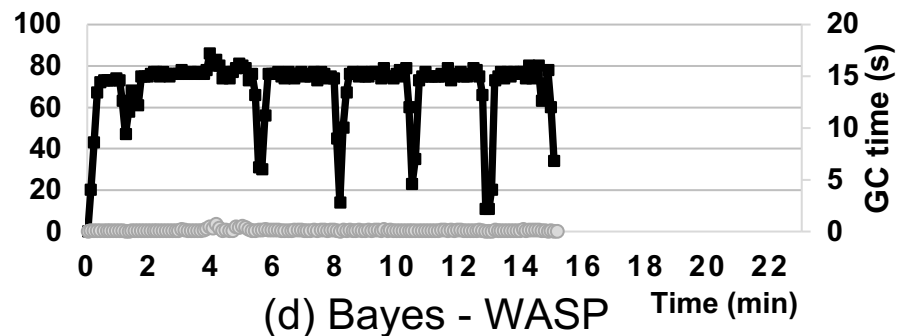
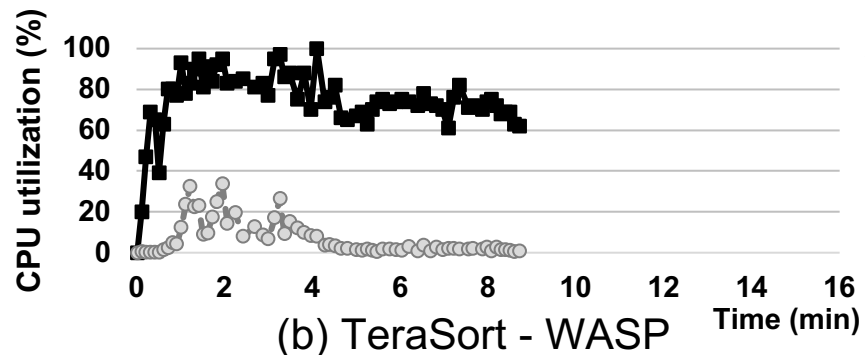
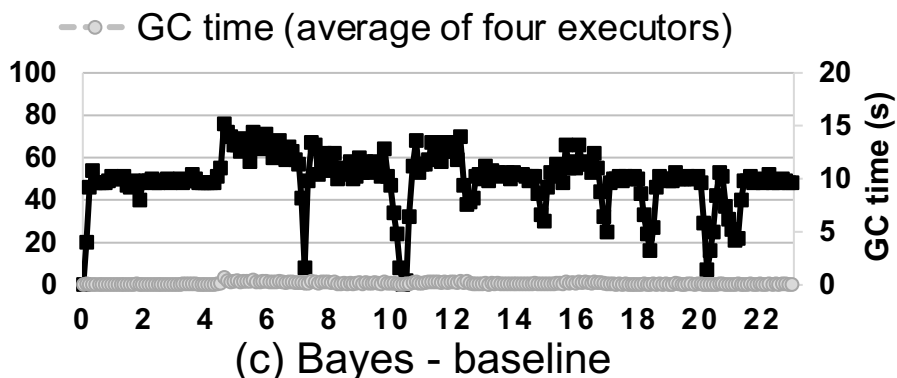
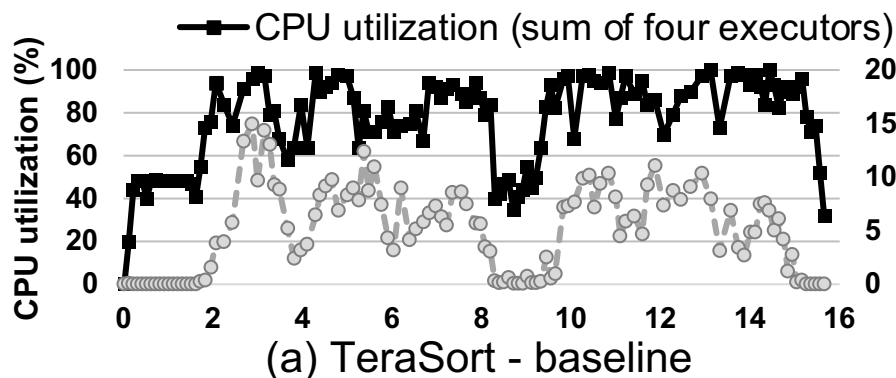
- **Geomean speedup: 1.31x**
- **Maximum speedup: 1.67x for TeraSort**
- **Operating cost reduction: 24% on average, 40% at maximum**

Sensitivity on GC Threshold



- **Performance over varying GC threshold for GC-aware task scheduler**
 - **20%** is the optimal value balancing task concurrency and GC overhead

Impact on GC time and CPU Utilization



- TeraSort: WASP reduces GC pause time by **72%**
- Bayes: CPU utilization is improved by **30%** over the baseline

Summary

- **Following Spark tuning guideline yields suboptimal performance**
 - Workload-oblivious, memory usage-oblivious, and coarse-grained

- **WASP jointly optimizes task granularity and concurrency**
 - Analytical model that predicts an optimal setting of $N_{partitions}$ and $N_{threads}$
 - Runtime GC-aware task scheduler to find an optimal $N_{threads}$

- **WASP allows a user to focus on program logic instead of tedious tasks of parameter tuning**
 - Geomean speedups: **1.74x** for 4-node cluster, **1.56x** for KNL, **1.31x** for AWS EC2

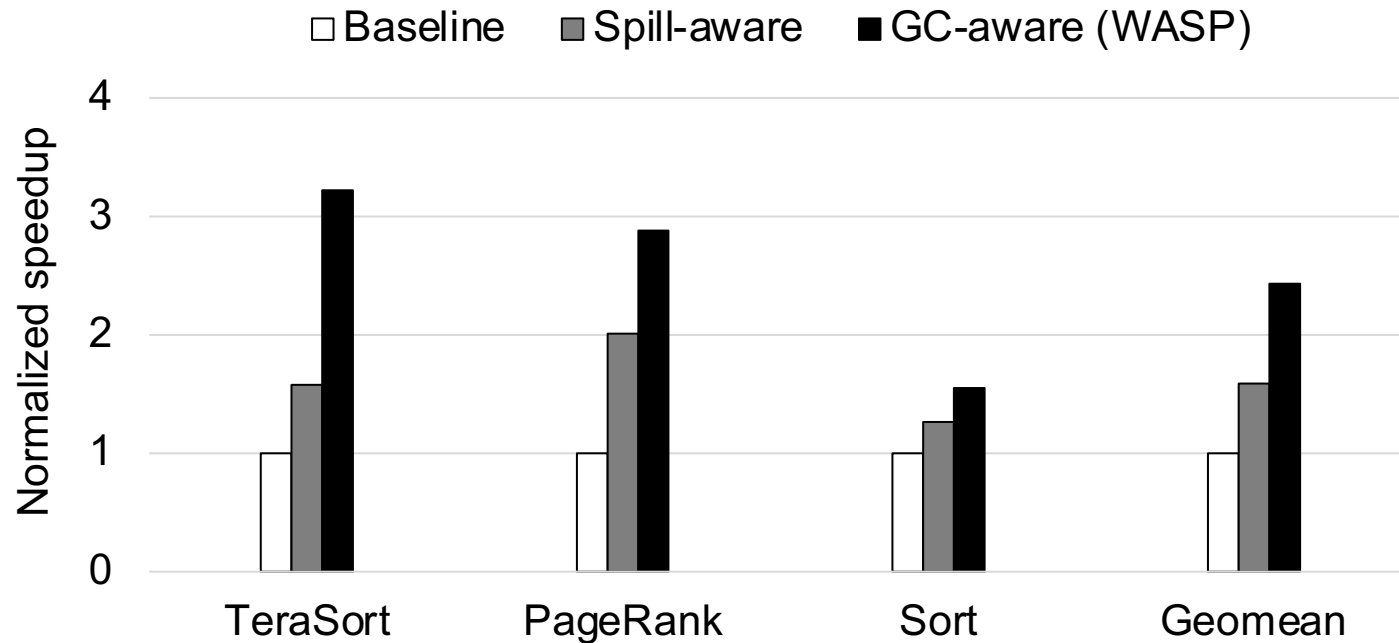
EXTRA SLIDES



Appendix A: 3-D Optimal <Npartitions, Nthreads>

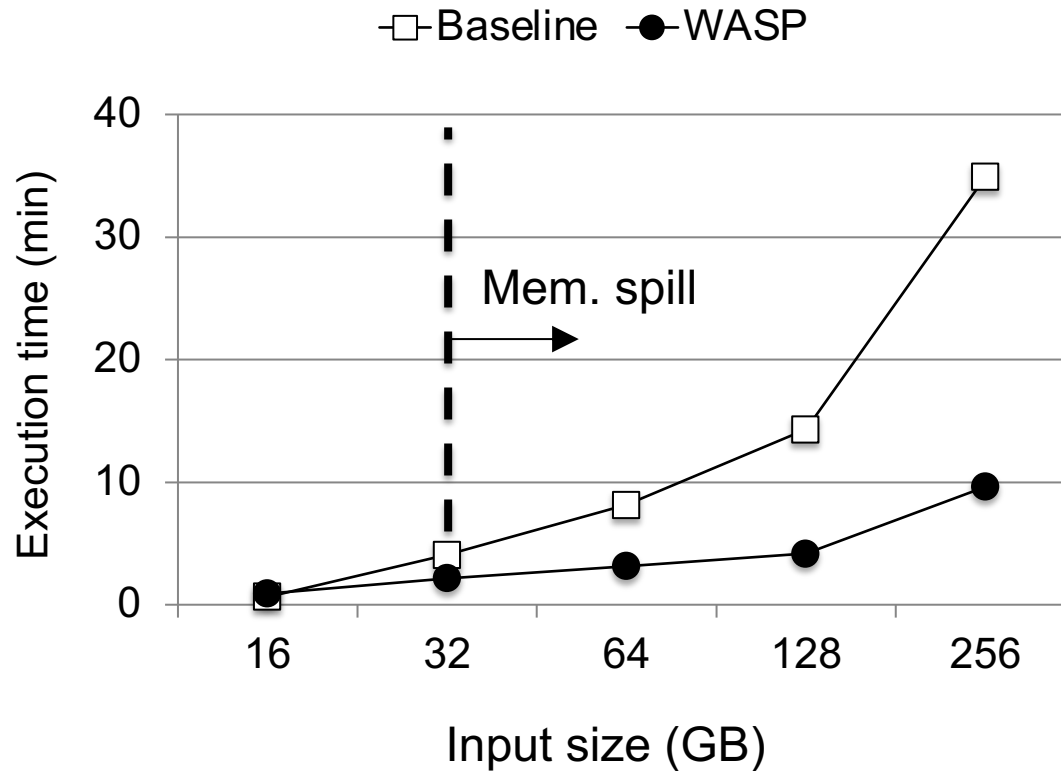
Benchmarks	Optimal $N_{partitions}$ and $N_{threads}$	
WordCount	3-D Optimal	Job 0: (Stage 0: (128, 8), Stage 1: (64, 4))
	WASP	Job 0: (Stage 0: (128, -), Stage 1: (256, -))
TeraSort	3-D Optimal	Job 0: (Stage 0: (64, 4)), Job 1: (Stage 1: (128, 4), Stage 2: (4096, 8))
	WASP	Job 0: (Stage 0: (128, -)), Job 1: (Stage 1: (128, -), Stage 2: (4096, -))
PageRank	3-D Optimal	Job 0: (Stage 0: (256, 4), Stage 1: (512, 8), Stage 2: (512, 8), Stage 3: (256, 8), Stage 4: (256, 8), Stage 5: (64, 4))
	WASP	Job 0: (Stage 0: (128, -), Stage 1: (512, -), Stage 2: (512, -), Stage 3: (256, -), Stage 4: (256, -), Stage 5: (64, -))
Sort	3-D Optimal	Job 0: (Stage 0: (128, 8), Stage 1: (2048, 4))
	WASP	Job 0: (Stage 0: (128, -), Stage 1: (2048, -))

Appendix B: Spill-Aware vs. GC-Aware Scheduler



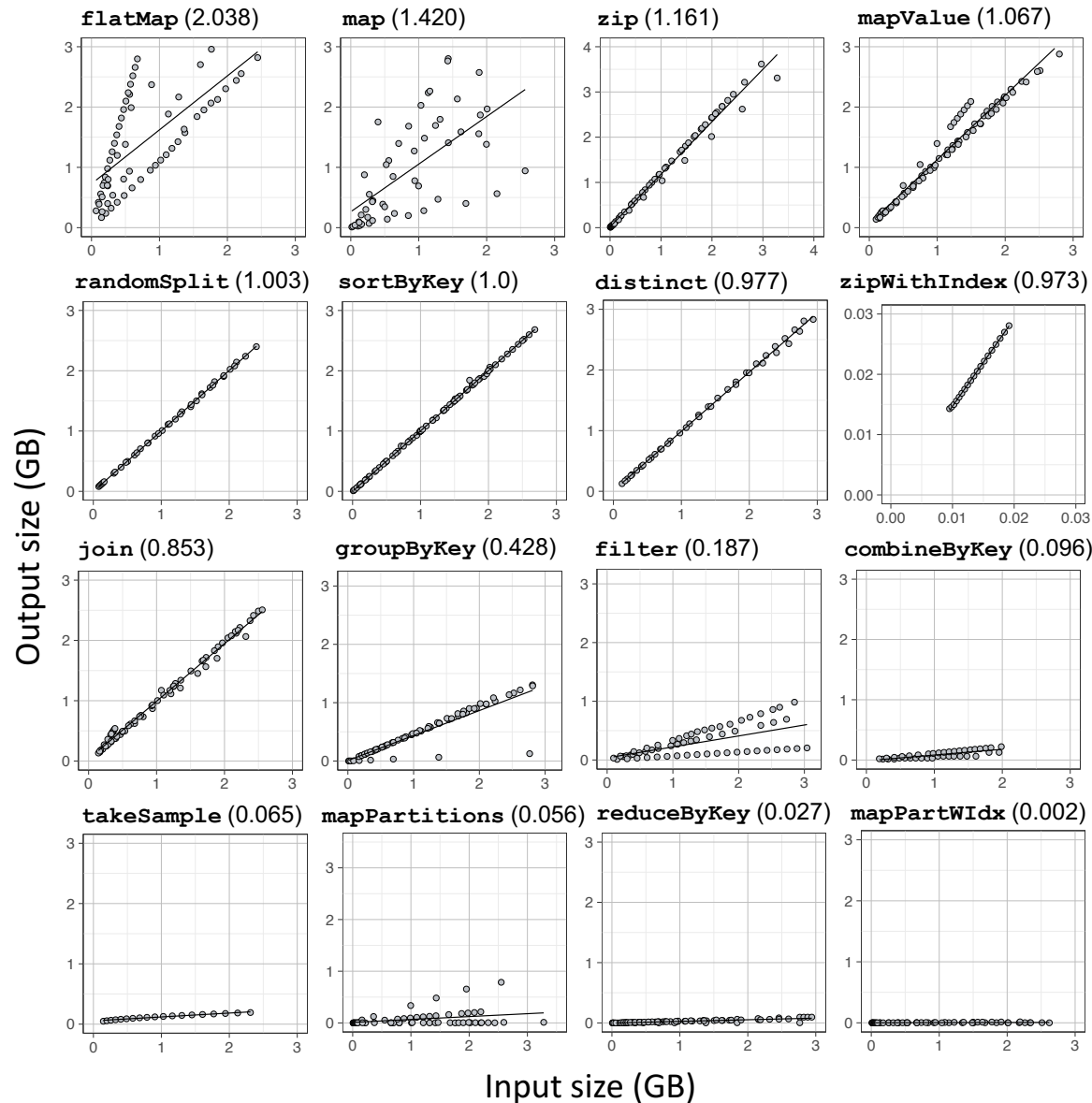
- A spill-aware scheduler yields suboptimal performance due to too conservative setting of $N_{threads}$.

Appendix C: Input Sensitivity of WASP



- **WASP adjusts $N_{partitions}$ and $N_{threads}$ using the GC-aware scheduler to achieve robust performance over all input sizes**

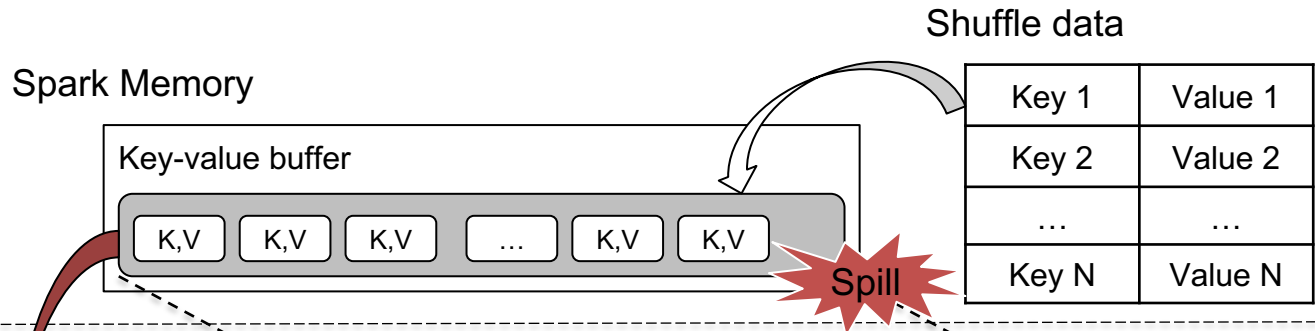
Appendix D: Memory Amplification Factor



Appendix E: Relation Between Spill and GC

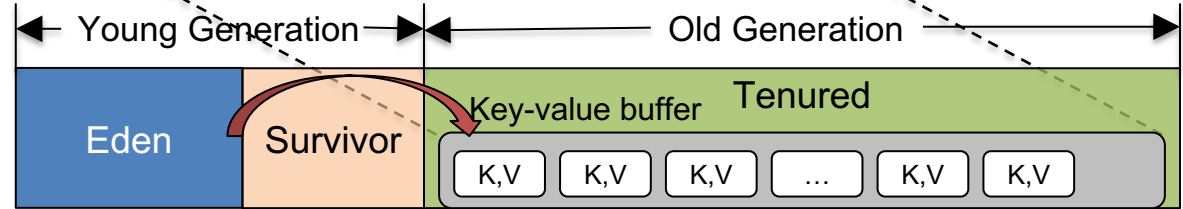
- Breakdown about relation between spill and GC

Apache Spark



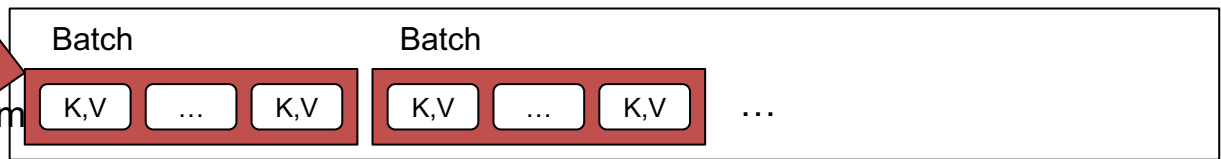
JVM heap layout

(1) Spill in-memory object to disk



(2) KV buffer is not freed until the spill is finished

Storage



In-memory Big Data Processing (1)

- **Recently embraced by big data analytics frameworks**
 - Apache Spark: Large-scale computation with in-memory caching
 - Apache Tez: Complex DAG for processing data built atop Apache YARN
 - Apache Ignite: Distributed database with SQL
- **Benefits**
 - Real-time computation with high throughput
 - Interactive analytics with low latency

