Distinguishing Parallel and Distributed Computing Performance

CCDSC 2016 http://www.netlib.org/utk/people/JackDongarra/CCDSC-2016/

La maison des contes Lyon France

Geoffrey Fox, Saliya Ekanayake, Supun Kamburugamuve October 4, 2016

gcf@indiana.edu

http://www.dsc.soic.indiana.edu/, http://spidal.org/ http://hpc-abds.org/kaleidoscope/

Department of Intelligent Systems Engineering

School of Informatics and Computing, Digital Science Center

Indiana University Bloomington

HPC-ABDS
High Performance Computing Enhanced Big Data Apache Stack DataFlow and In-place Runtime

Big Data and Parallel/Distributed Computing

- In one beginning, MapReduce enables easy parallel database like operations and Hadoop provided wonderful open source implementation
- Andrew Ng introduced “summation form” and showed much Machine Learning could be implemented in MapReduce (Mahout in Apache)
- Then noted that iteration ran badly in Hadoop as used disks for communication; Hadoop choice gives great fault tolerance
- This led to a slew of MapReduce improvements using either BSP SPMD (Twister, Giraph) or
- Dataflow: Apache Storm, Spark, Flink, Heron …. and Dryad
- Recently Google open sources Google Cloud Dataflow as Apache Beam using Spark and Flink as runtime or proprietary Google Dataflow
  - Support **Batch** and **Streaming**

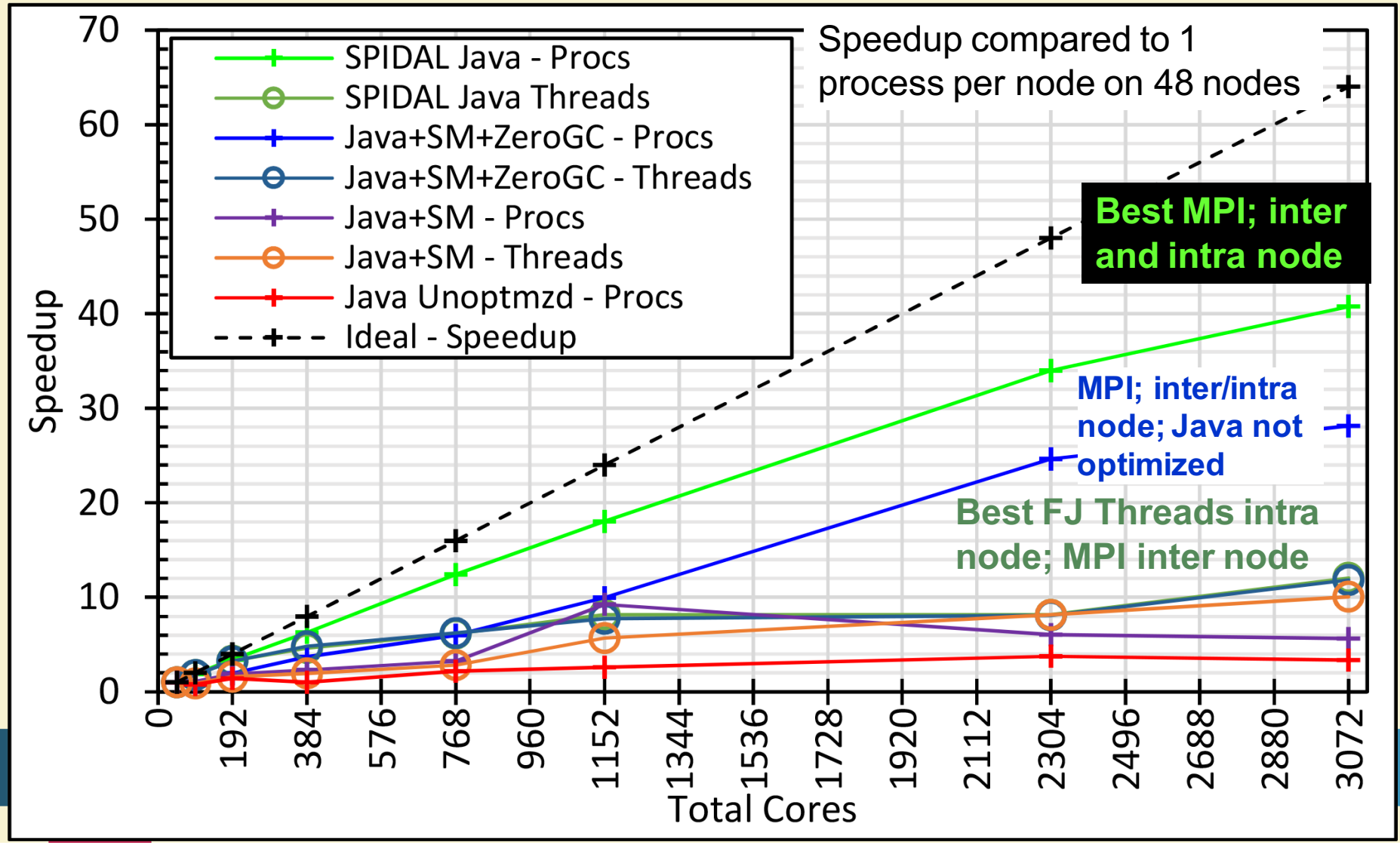
HPC-ABDS Parallel Computing

- Both simulations and data analytics use similar **parallel computing** ideas
- Both do **decomposition** of both model and data
- Both tend use **SPMD** and often use **BSP** Bulk Synchronous Processing
- One has computing (called **maps** in big data terminology) and **communication/reduction** (more generally collective) **phases**
- **Big data** thinks of problems as **multiple linked queries** even when queries are small and uses **dataflow** model
- **Simulation** uses dataflow for multiple linked applications but small steps such as iterations are done **in place**
- **Reduction** in HPC (**MPIReduce**) done as optimized tree or pipelined communication between same processes that did computing
- **Reduction** in Hadoop or Flink done as separate map and reduce processes using dataflow
  - This leads to 2 forms (**In-Place** and **Flow**) of runtime discussed later
- Interesting **Fault Tolerance** issues highlighted by Hadoop-MPI comparisons – not discussed here!

Java MPI performance is good if you work hard

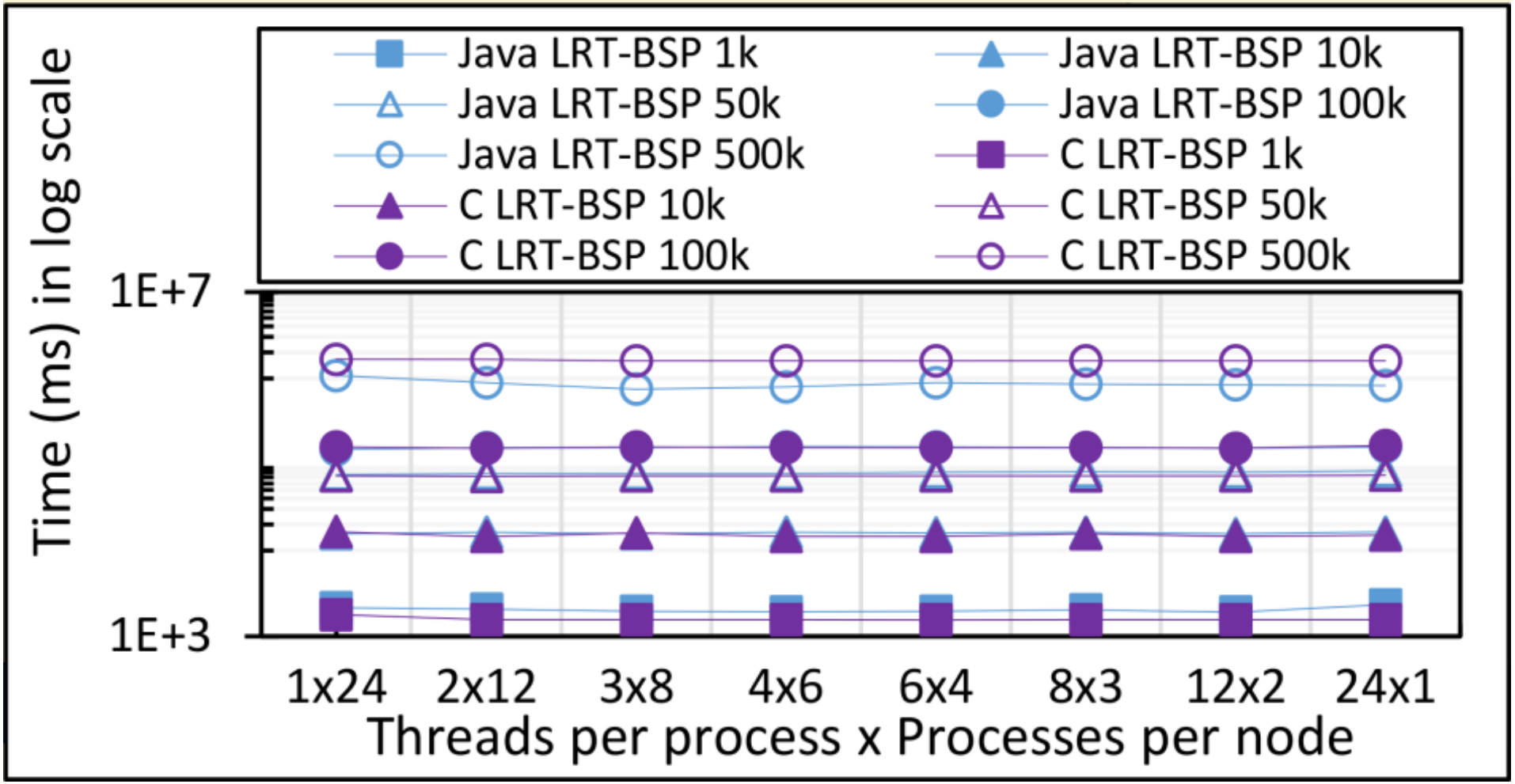
128 24 core Haswell nodes on SPIDAL 200K DA-MDS Code

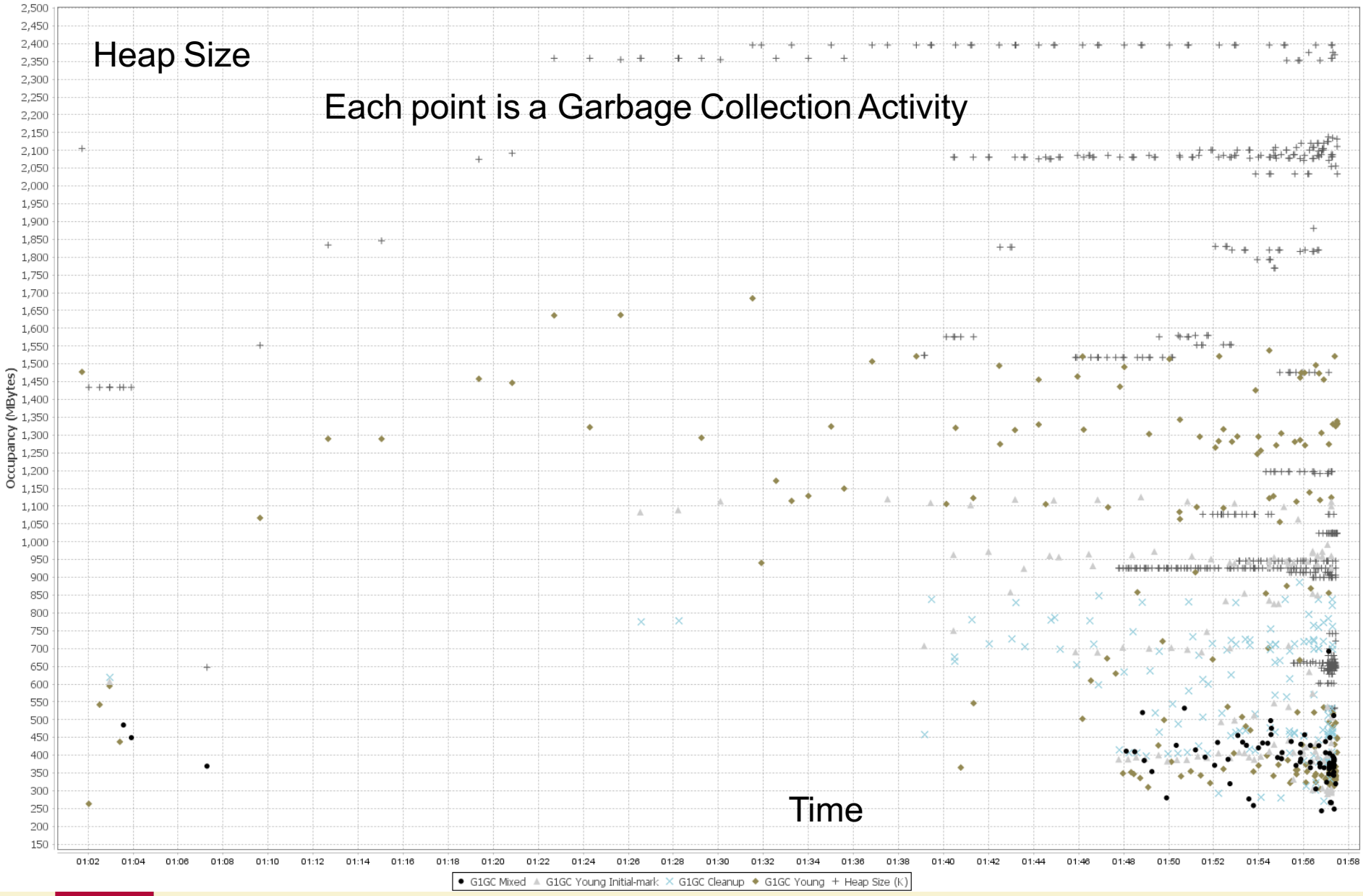
Communication dominated by MPI **Collective performance**

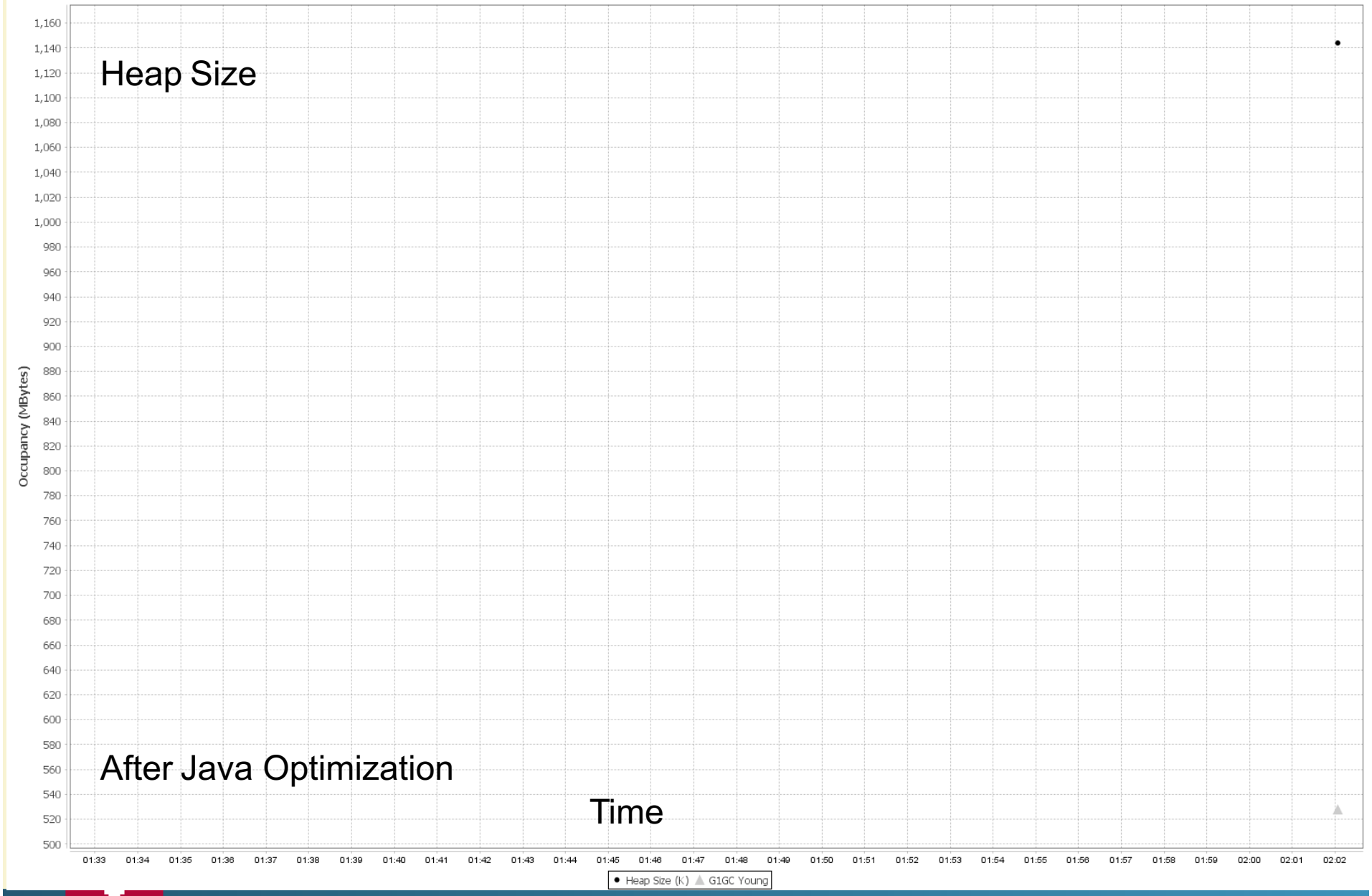


Java versus C Performance

- C and Java Comparable with Java doing better on larger problem sizes
- LRT-BSP affinity CE; one million point dataset 1k,10k,50k,100k, and 500k centers on 16 24 core Haswell nodes over varying threads and processes.




Heap Size
Each point is a Garbage Collection Activity
Occupancy (MBytes)
Time
G1GC Mixed
G1GC Young Initial-mark
G1GC Cleanup
G1GC Young
Heap Size (K)


Heap Size
After Java Optimization
Time
Occupancy (MBytes)
1,160
1,140
1,120
1,100
1,080
1,060
1,040
1,020
1,000
980
960
940
920
900
880
860
840
820
800
780
760
740
720
700
680
660
640
620
600
580
560
540
520
500
01:33
01:34
01:35
01:36
01:37
01:38
01:39
01:40
01:41
01:42
01:43
01:44
01:45
01:46
01:47
01:48
01:49
01:50
01:51
01:52
01:53
01:54
01:55
01:56
01:57
01:58
01:59
02:00
02:01
02:02
Heap Size (K)
G1GC Young

Apache Flink and Spark Dataflow Centric Computation

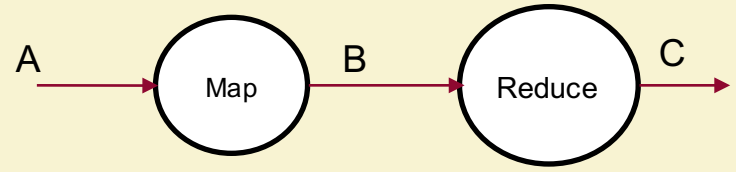
- Both express a computation as a data-flow graph
- Graph Nodes → User defined operators
- Graph Edges → Data
- Data source nodes and Data sink nodes (i.e. File read, File write, Message Queue read)
- Automatic placement of partitioned data in the parallel tasks

Dataflow Operation

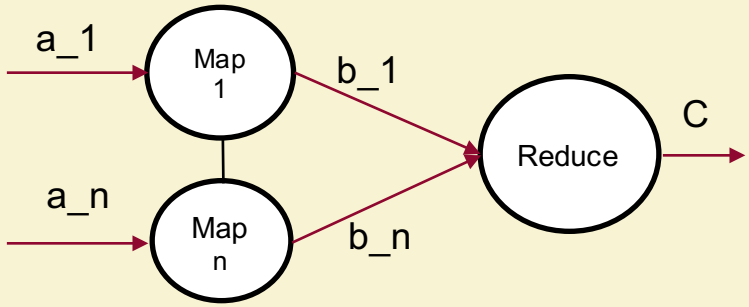
- The operators in API define the computation as well how nodes are connected
  - For example lets take map and reduce operators and our initial data set is A
  - Map function produces a distributed dataset B by applying the user defined operator on each partition of A. If A had N partitions, B can contain N elements.
  - The Reduce function is applied on B, producing a data set with a single partition.

```
B = A.map() {
    User defined code to execute on a partition of A
};

C = B.reduce() {
    User defined code to reduce two elements in B
}
```



Logical graph



Execution graph

Dataflow operators

- **Map:** Parallel execution
- **Filter:** Filter out data
- **Project**: Select part of the data unit
- **Group:** Group data according to some criteria like keys
- **Reduce:** Reduce the data into a small data set.
- **Aggregate:** Works on the whole data set to compute a value, SUM, MIN
- **Join:** Join two data sets based on Keys.
- **Cross:** Join two data sets as in the cross product
- **Union**: Union of two data sets

Apache Spark

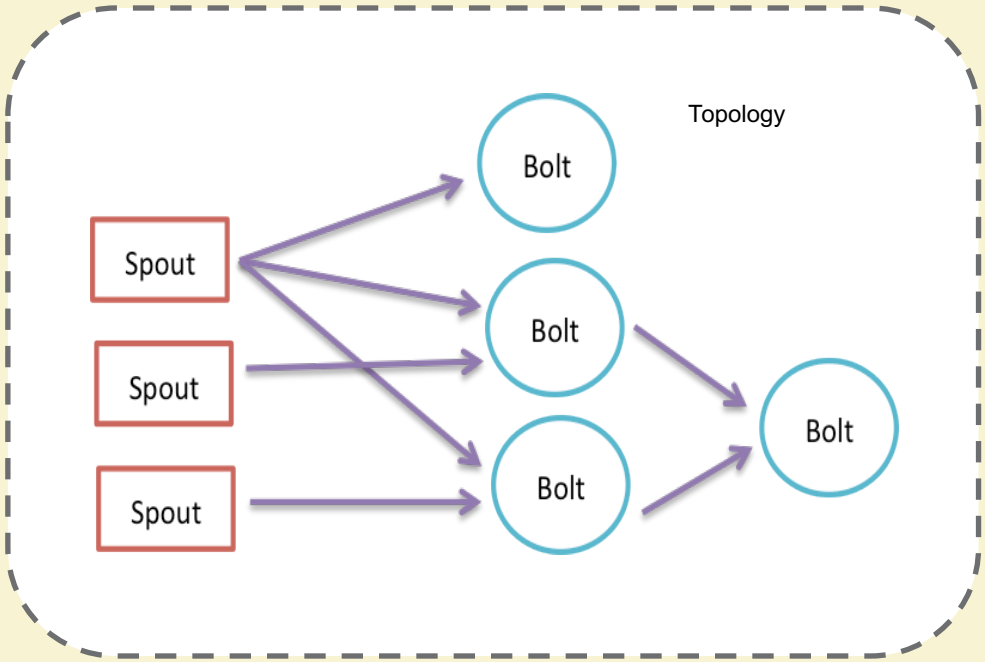
- Dataflow is executed by a driver program
- The graph is created on the fly by the driver program
- The data is represented as Resilient Distributed Data Sets (RDD)
- The data is on the worker nodes and operators applied on this data
- These operators produce other RDDs and so on
- Using lineage graph of RDDs for fault tolerance

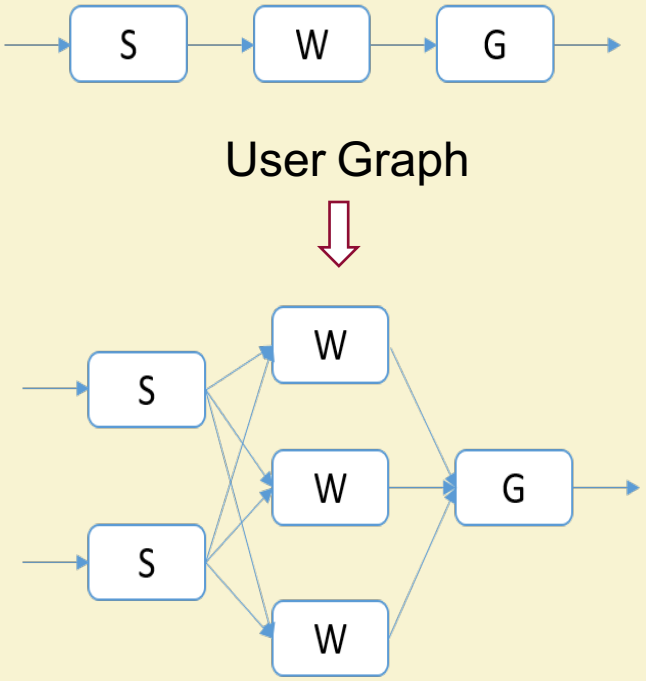
Apache Flink

- The data is represented as a DataSet
- User creates the dataflow graph and submits to cluster
- Flink optimizes this graph and creates an execution graph
- This graph is executed by the cluster
- Support Streaming natively
- Check-pointing based fault tolerance

Twitter (Apache) Heron

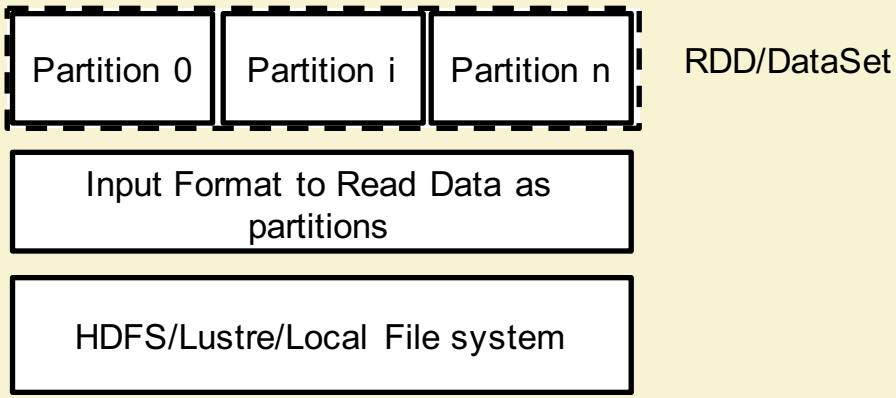
- Extends Storm
- Pure data streaming
- Data flow graph is called a topology





Data abstractions (Spark RDD & Flink DataSet)

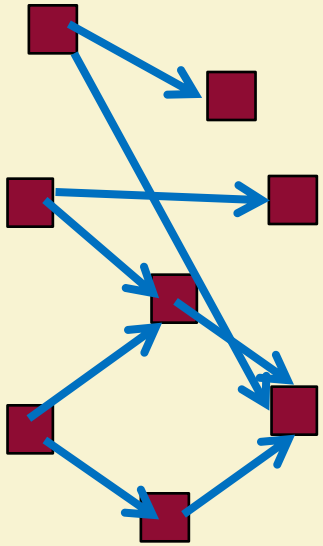
- In memory representation of the partitions of a distributed data set
- Has a high level language type (Integer, Double, custom Class)
- Immutable
- Lazy loading
- Partitions are loaded in the tasks
- Parallelism is controlled by the no of partitions (parallel tasks on a data set <= no of partitions)

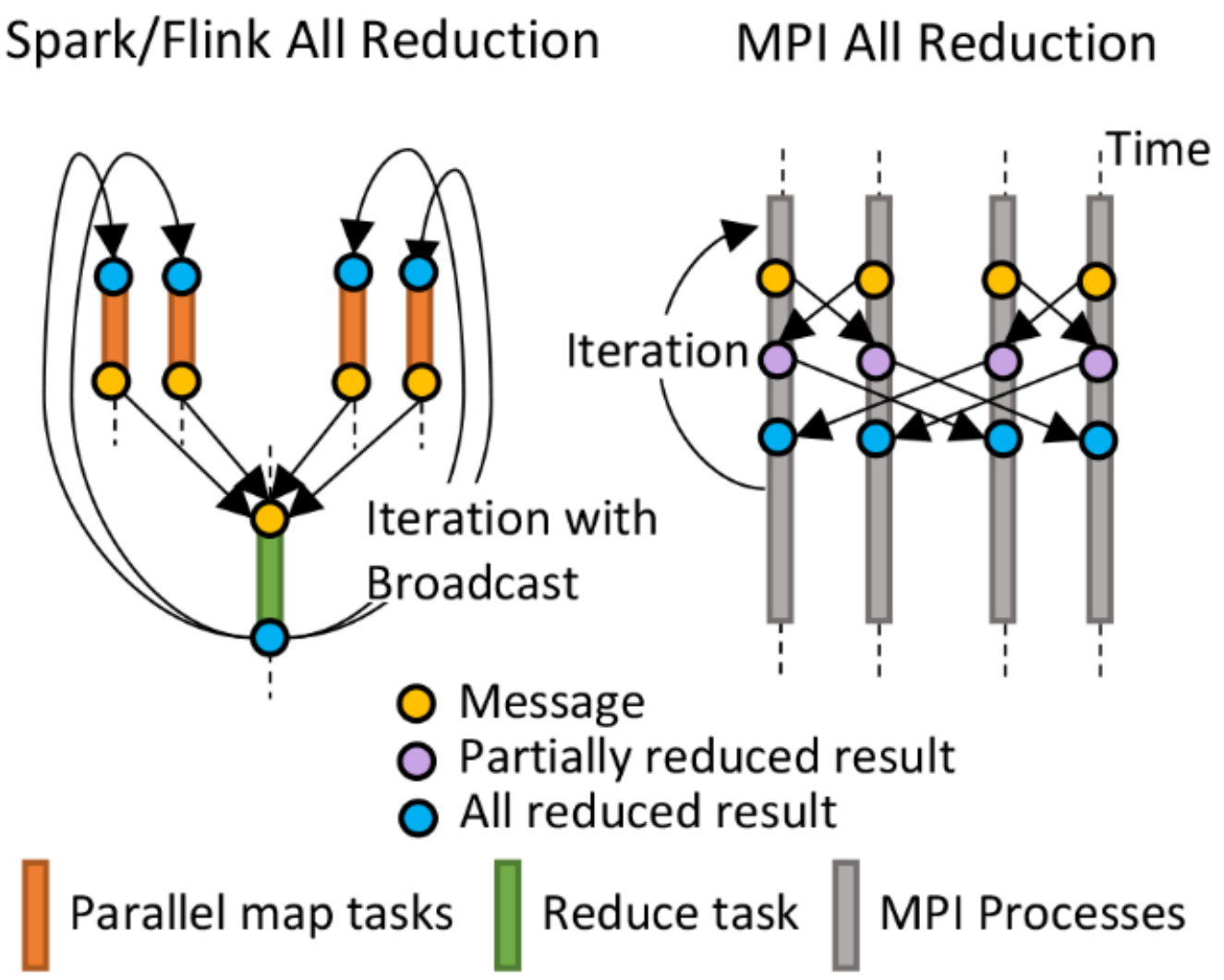


Breaking Programs into Parts

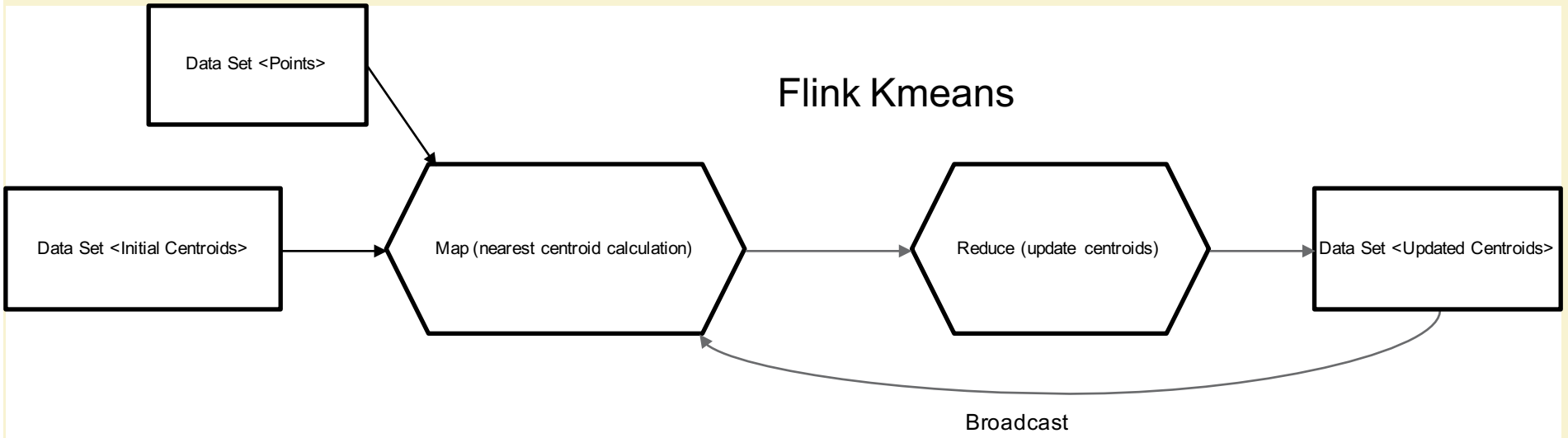
Fine Grain Parallel Computing Data/model parameter decomposition

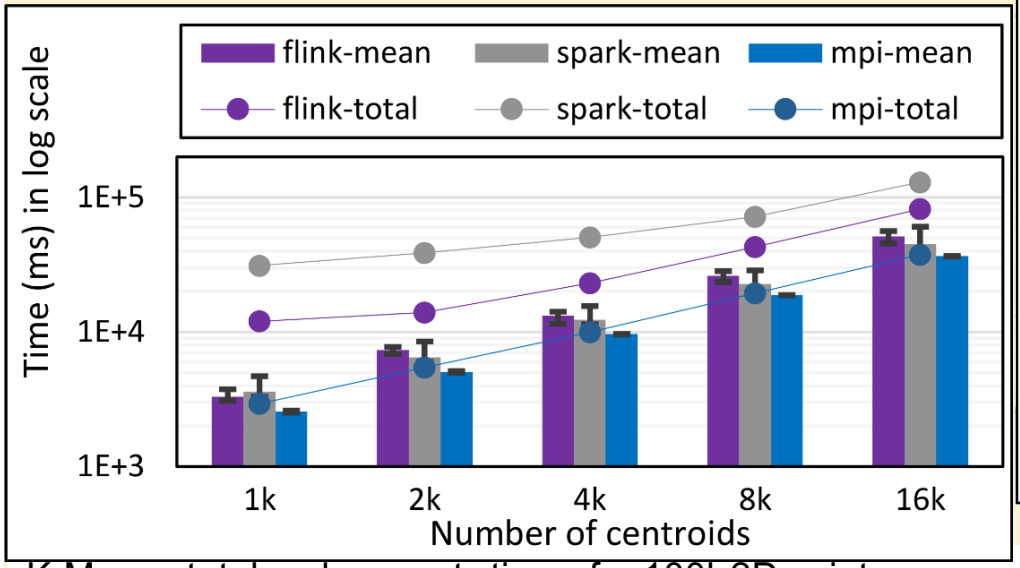
Coarse Grain Dataflow HPC or ABDS



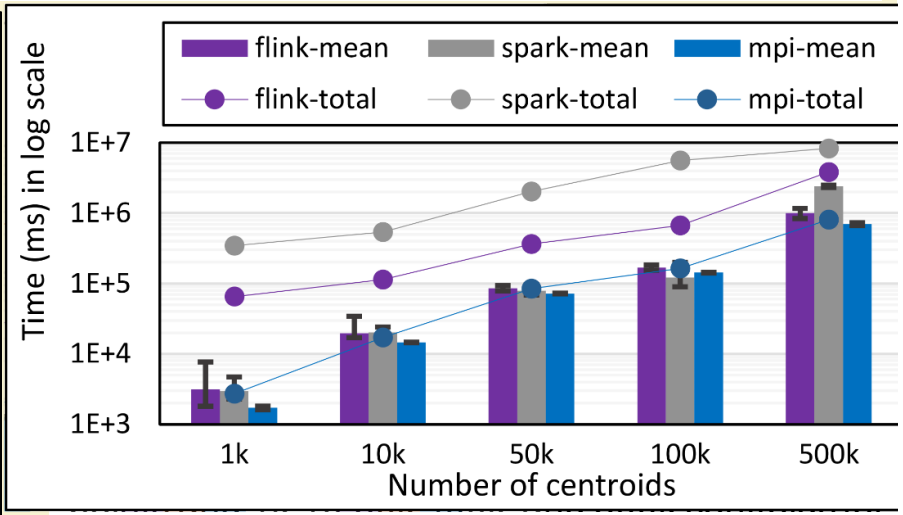


K-Means Clustering in Spark, Flink, MPI





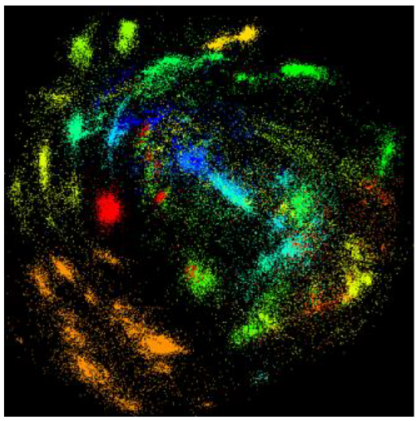
K-Means total and compute times for 100k 2D points and1k,2k,4k,8k, and 16k centroids for Spark, Flink, and MPI Java LRT-BSP CE. Run on 1 node as 24x1.



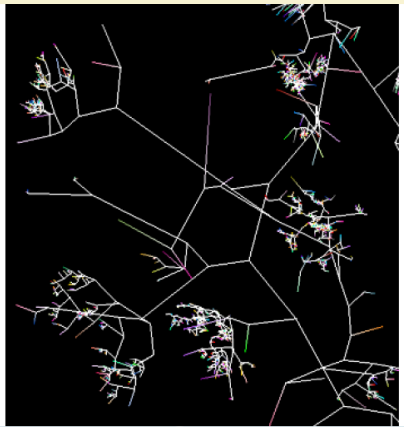
K-Means total and compute times for 1 million 2D points and 1k,10,50k,100k, and 500k centroids for Spark, Flink, and MPI Java LRT-BSP CE. Run on 16 nodes as 24x1.

Flink Multi Dimensional Scaling (MDS)

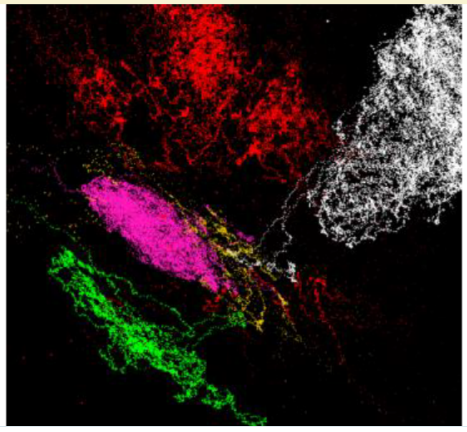
- Projects NxN distance matrix to NxD matrix where N is the number of points and D is the target dimension.
- The input is two NxN matrices (Distance and Weight) and one NxD initial point file.
- The NxN matrices are partitioned row wise
- 3 NxD matrices are used in the computations which are not partitioned.
- For each operation on NxN matrices, one or more of NxD matrices are required.
- All three iterations have dynamic stopping criteria. So loop unrolling is not possible.
- Our algorithm uses deterministic annealing and has three nested loops called Temperature, Stress and CG (Conjugate Gradient) in that order.



(a) 100,000 fungi sequences

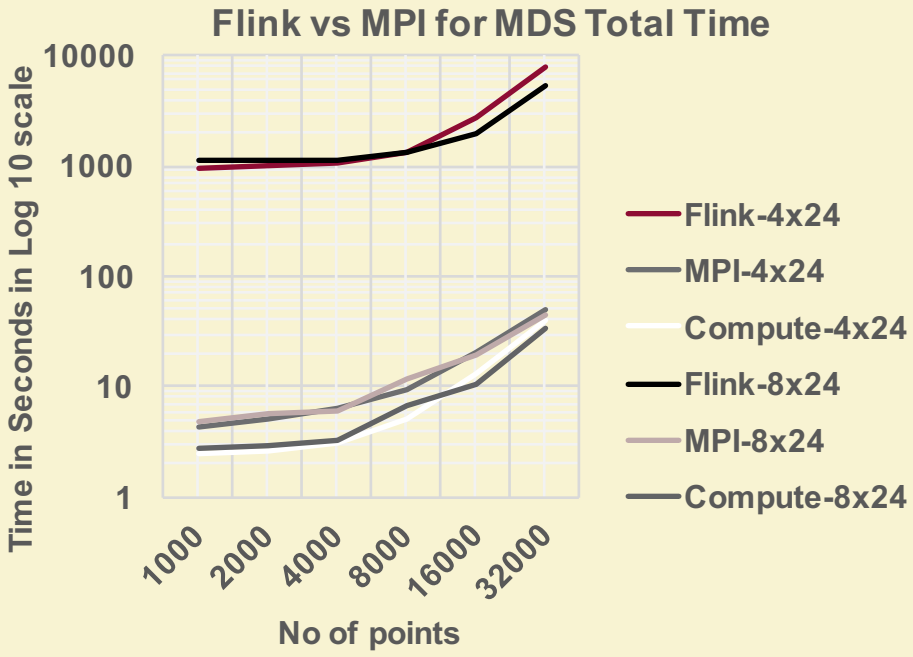


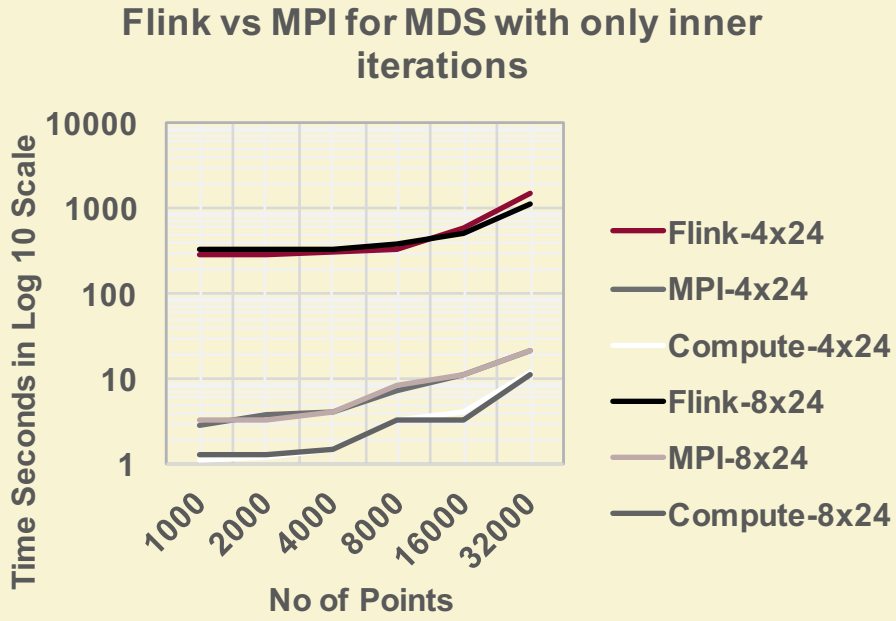
(b) 3D phylogenetic tree



(c) 3D plot of vector data

Flink vs MPI DA-MDS Performance





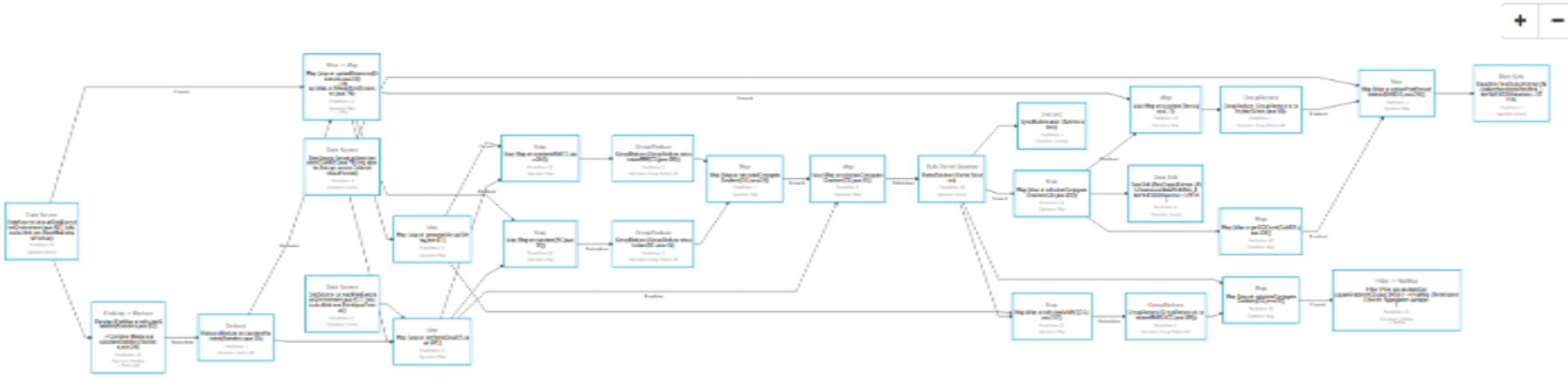
Total time of MPI Java and Flink MDS implementations for 96 and 192 parallel tasks with no of points ranging from 1000 to 32000. The graph also show the computation time. The total time includes computation time, communication overhead, data loading and framework overhead. In case of MPI there is no framework overhead. This test has 5 Temperature Loops, 2 Stress Loops and 16 CG Loops.

Total time of MPI Java and Flink MDS implementations for 96 and 192 parallel tasks with no of points ranging from 1000 to 32000. The graph also show the computation time. The total time includes computation time, communication overhead, data loading and framework overhead. In case of MPI there is no framework overhead. This test has 1 Temperature Loop, 1 Stress Loop and 32 CG Loops.

Flink MDS Dataflow Graph for MDS inner loop

Flink Java Job at Mon Oct 03 12:27:10 EDT 2016 | b3057e083f05a3b5c4ee2f515ab7d7b8 | 0 0 0 0 27 0 0 0 | 2016-10-03, 12:27:10 - 2016-10-03, 12:39:34 | 12m 23s

Plan | Timeline | Exceptions | Properties | Configuration



Subtasks | TaskManagers | Accumulators | Checkpoints

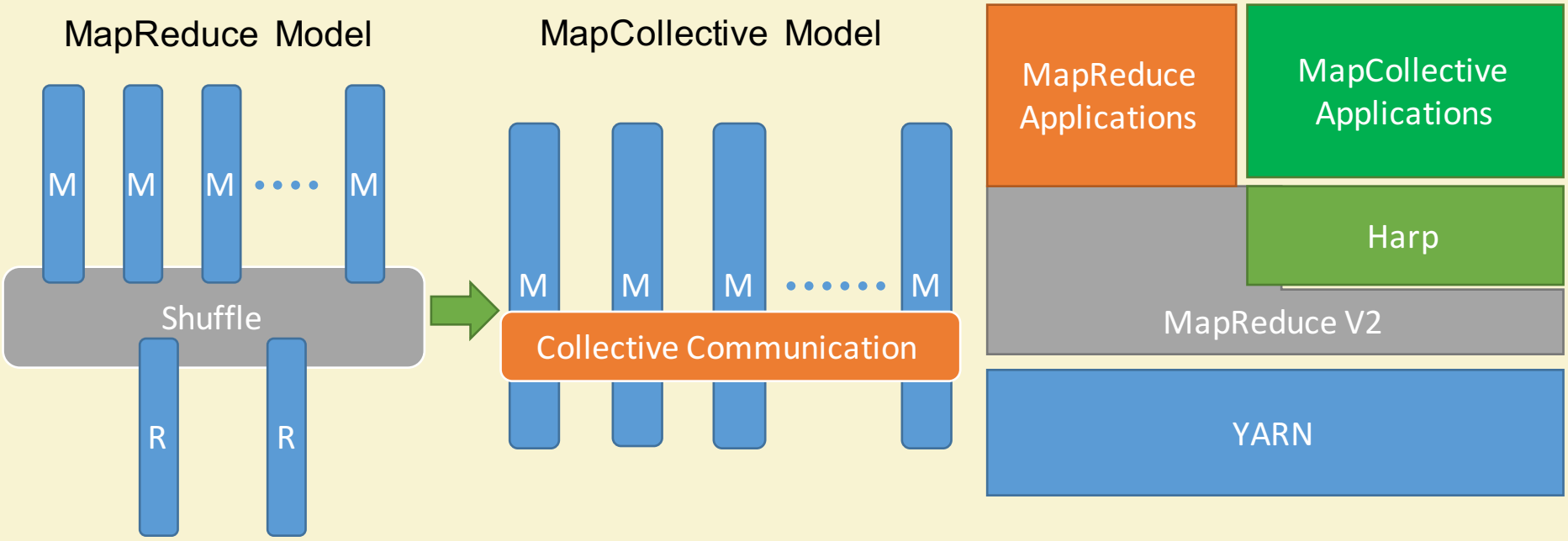
Start Time	End Time	Duration	Name	Bytes received	Records received	Bytes sent	Records sent	Tasks	Status
2016-10-03, 12:27:10	2016-10-03, 12:33:18	6m 7s	DataSource (at readFile(ExecutionEnvironment.java:517) (edu.iu.dsc.flink.mm.ShortMatrixInputFormat))	0 B	0	3.81 GB	64	0 0 0 32 0 0 0	FINISHED
2016-10-03, 12:27:10	2016-10-03, 12:27:10	45ms	DataSource (at setupStressIteration(DAMDS.java:71) (org.apache.flink.api.java.io.CollectionInputFormat))	0 B	0	1.68 KB	33	0 0 0 1 0 0 0	FINISHED
2016-10-03, 12:27:10	2016-10-03, 12:27:11	739ms	DataSource (at readFile(ExecutionEnvironment.java:517) (edu.iu.dsc.flink.mm.PointInputFormat))	0 B	0	750 KB	1	0 0 0 32 0 0 0	FINISHED
2016-10-03, 12:33:18	2016-10-03, 12:33:23	5s	CHAIN FlatMap (FlatMap at calculateStatistics(Statistics.java:12)) -> Combine (Reduce at calculateStatistics(Statistics.java:20))	1.91 GB	32	2.56 KB	32	0 0 0 32 0 0 0	FINISHED
2016-10-03, 12:33:23	2016-10-03, 12:33:23	426ms	Reduce (Reduce at calculateStatistics(Statistics.java:20))	2.56 KB	32	5.13 KB	64	0 0 0 1 0 0 0	FINISHED
2016-10-03, 12:33:18	2016-10-03, 12:33:57	38s	CHAIN Map (Map at updateDistances(Distances.java:33)) -> Map (Map at fillReadJoin(Distances.java:74))	1.91 GB	32	11.4 GB	96	0 0 0 32 0 0 0	FINISHED
2016-10-03, 12:33:57	2016-10-03, 12:34:29	32s	Map (Map at generateVArray(VArray.java:17))	3.81 GB	32	3.82 GB	64	0 0 0 32 0 0 0	FINISHED
2016-10-03, 12:27:10	2016-10-03, 12:33:24	6m 13s	Map (Map at joinStats(DAMDS.java:345))	750 KB	1	47.6 MB	65	0 0 0 32 0 0 0	FINISHED
2016-10-03, 12:33:24	2016-10-03, 12:34:40	1m 16s	Map (Map at calculateMM(CG.java:260))	1.91 GB	32	752 KB	32	0 0 0 32 0 0 0	FINISHED

HPC-ABDS Parallel Computing

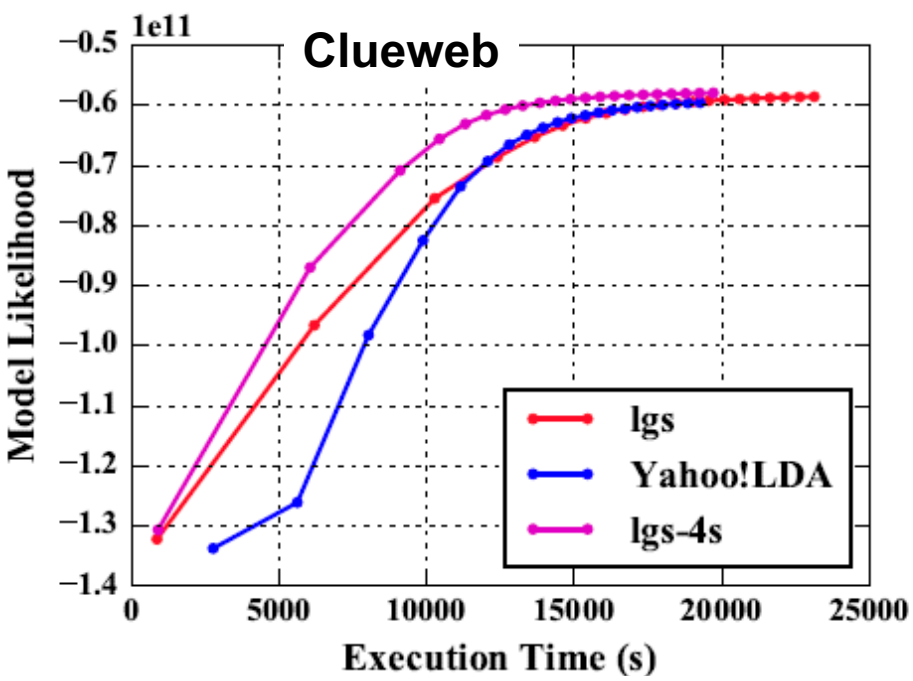
- **MPI** designed for fine grain case and typical of parallel computing used in large scale simulations
  - **Only change in model parameters** are transmitted
  - **In-place** implementation
  - Synchronization important as parallel computing
- **Dataflow** typical of distributed or Grid computing workflow paradigms
  - Data sometimes and model parameters certainly transmitted
  - If used in workflow, large amount of computing and no synchronization constraints
  - Caching in iterative MapReduce avoids data communication and in fact systems like TensorFlow, Spark or Flink are called dataflow but often implement **"model-parameter" flow**
- **Inefficient** to use **same runtime mechanism** independent of characteristics
  - Use **In-Place** implementations for parallel computing with high overhead and Flow for flexible low overhead cases
- **HPC-ABDS** plan is to keep current user interfaces (say to Spark Flink Hadoop Storm Heron) and **transparently use HPC** to improve **performance**

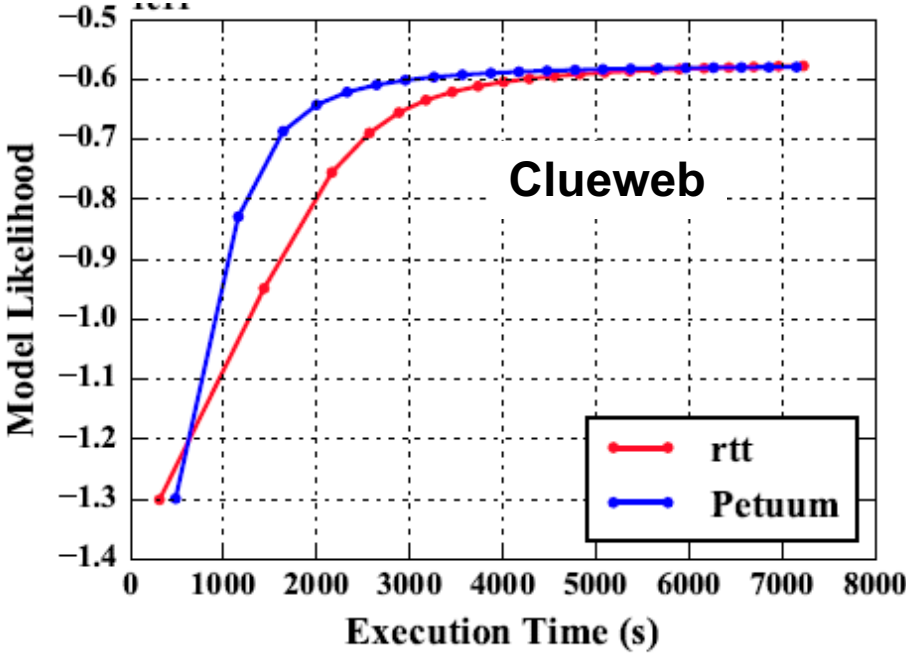
Harp (Hadoop Plugin) brings HPC to ABDS

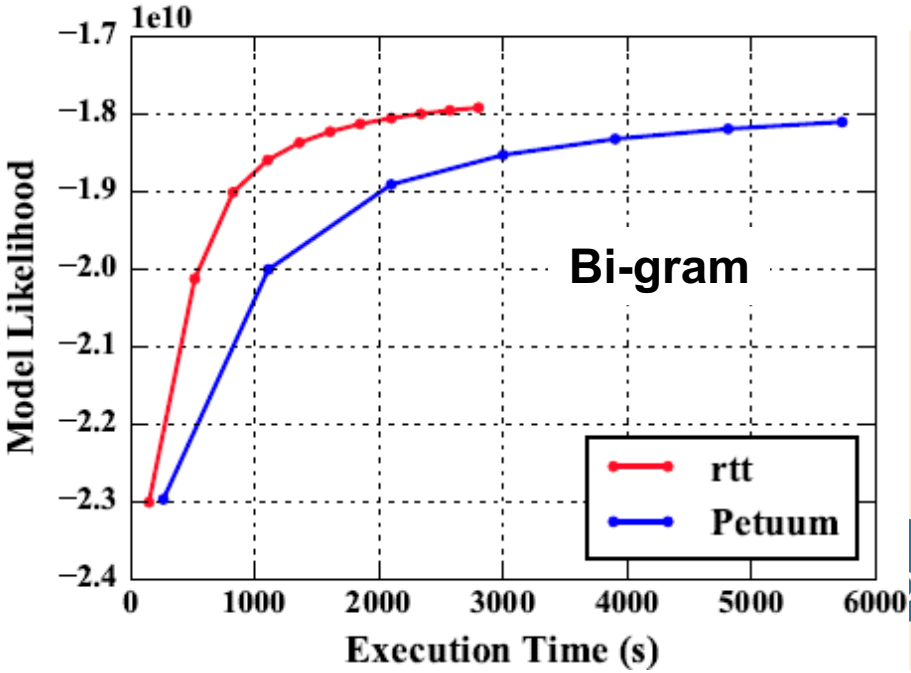
- **Basic Harp: Iterative HPC communication; scientific data abstractions**
- Careful support of distributed data AND distributed model
- Avoids parameter server approach but distributes model over worker nodes and supports collective communication to bring global model to each node
- Applied first to Latent Dirichlet Allocation LDA with large model and data

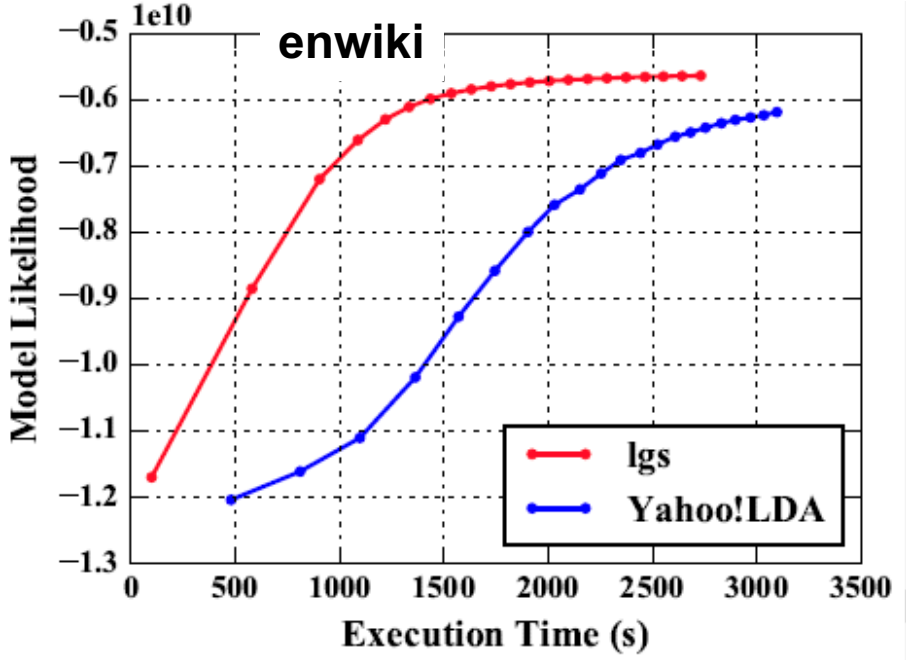


Latent Dirichlet Allocation on 100 Haswell nodes: red is Harp (lgs and rtt)



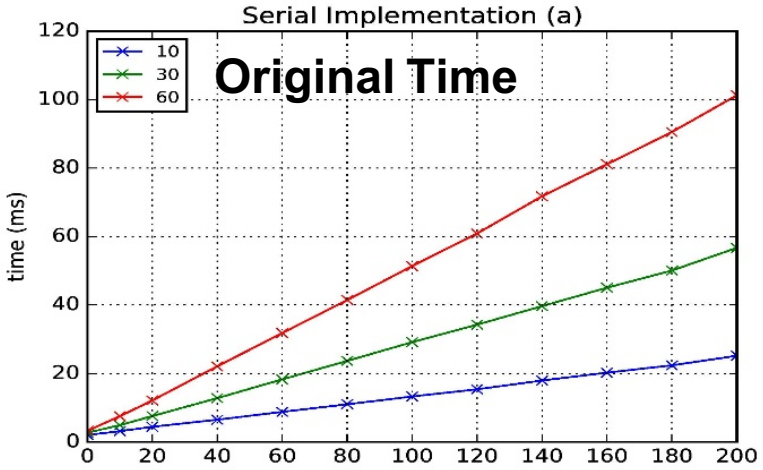


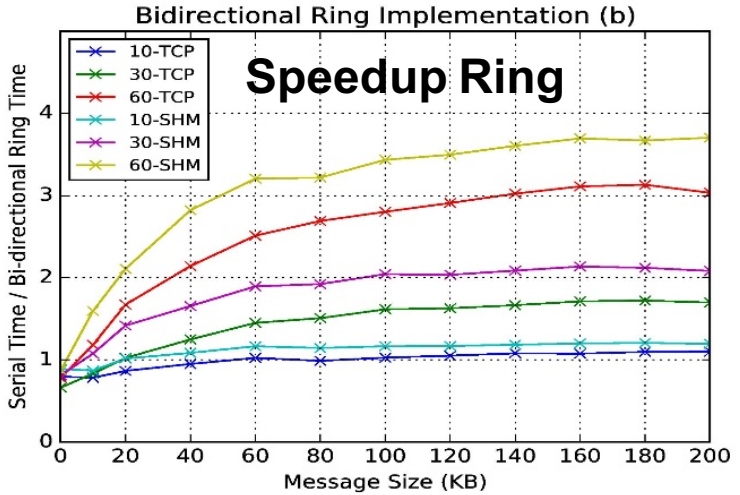


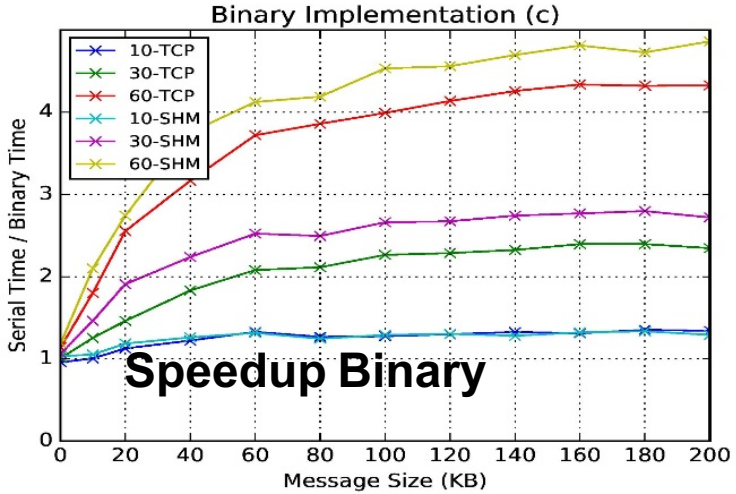


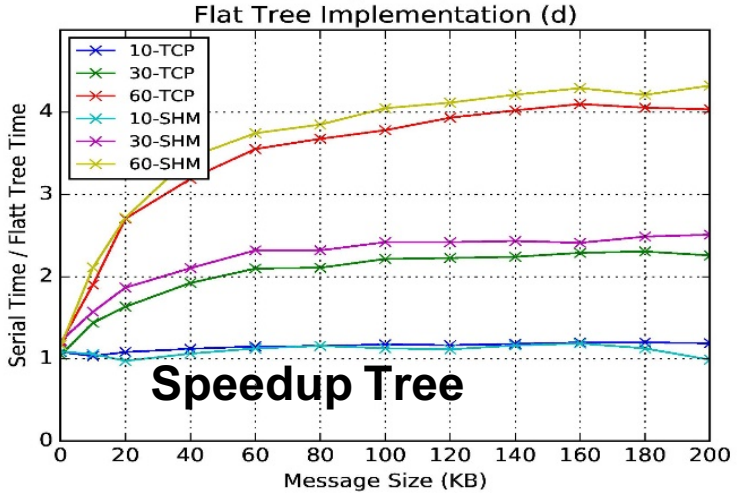
Improvement of Storm (Heron) using HPC communication algorithms

Latency of binary tree, flat tree and bi-directional ring implementations compared to serial implementation. Different lines show varying # of parallel tasks with either TCP communications and shared memory communications(SHM).









Next Steps in HPC-ABDS

- Currently we are circumventing the dataflow and adding HPC runtime to Apache systems (Storm, Hadoop, Heron)
- We can aim exploit better Apache capabilities and additionally
  1) Allow more careful data specification such as separating "model parameters" (variable) from "data" (fixed)
  2) Allowing richer (less automatic) dataflow and data placement operations. In particular allow equivalent of HPF DECOMPOSITION directive to align decompositions
  3) Implement HPC dataflow runtime "in-place" to implement "classic parallel computing" and higher performance dataflow
- This is all as modifications to Apache source
- We should be certain to be less ambitious than HPF and not spend 5 years making an inadequate compiler
- Resultant system will support parallel computing and orchestration (workflow), batch and streaming.

Application Requirements

- It would be good to understand which applications fit
  a) MapReduce
  b) Current Spark/Flink/Heron but not MapReduce
  c) HPC-ABDS (MPI) but not current Spark/Flink/Heron

- a) could be Database (Hive etc.), Data-lake and
- b) is certainly Streaming and Spark ec. outperform Hadoop on many big data applications
- c) is Global Machine Learning (large scale and iterative like Deep learning, clustering, hidden factors, topic models) and graph problems