Alchemist

An Apache Spark ⇔ MPI Interface

Kai Rothauge, Alex Gittens, Michael W. Mahoney

Thanks to NERSC and Cray Inc. for help and support!

riselab UC Berkeley

What is MPI?

- **MPI** = **M**essage **P**assing **I**nterface
- A *specification* for the developers and users of message passing libraries
- *Message-Passing Parallel Programming Model*:
  - cooperative operations between processes
  - data moved from address space of one process to that of another
- Dominant model in **high-performance computing**
- Popular *implementations*: MPICH, Open MPI
- Generally regarded as “low-level” for purposes of distributed computing

More on MPI

- Efficient implementations of collective operations
- Works with *distributed memory*, *shared memory*, *GPUs*
- Requires installation of MPI implementation on system
- Communication between MPI processes:
  - via TCP/IP sockets, *or*
  - optimized for underlying interconnects (InfiniBand, Cray Aries, Intel Omni-Path, etc.)
- **Communicator** objects connect groups of MPI processors
- **Con**: No *fault tolerance* or *elasticity*

Case Study: Spark vs. MPI

- **Numerical linear algebra (NLA)** using Spark vs. MPI
- Why do linear algebra in Spark?

NERSC National Energy Research Scientific Computing Center

Spark for data-centric workloads and scientific analysis

riselab UC Berkeley

Characterization of linear algebra in Spark

CRAY

Customers demand Spark; want to understand performance concerns

Case Study: Spark vs. MPI

- **Numerical linear algebra (NLA)** using Spark vs. MPI
- Why do linear algebra in Spark?
  - Pros:
    - Faster development, easier reuse
    - Simple dataset abstractions (RDDs, DataFrames, DataSets)
    - An entire ecosystem that can be used before and after the NLA computations
    - Spark can take advantage of available local linear algebra codes
    - Automatic fault-tolerance, out-of-core support
  - Con:
    - Classical MPI-based linear algebra implementations will be faster and more efficient

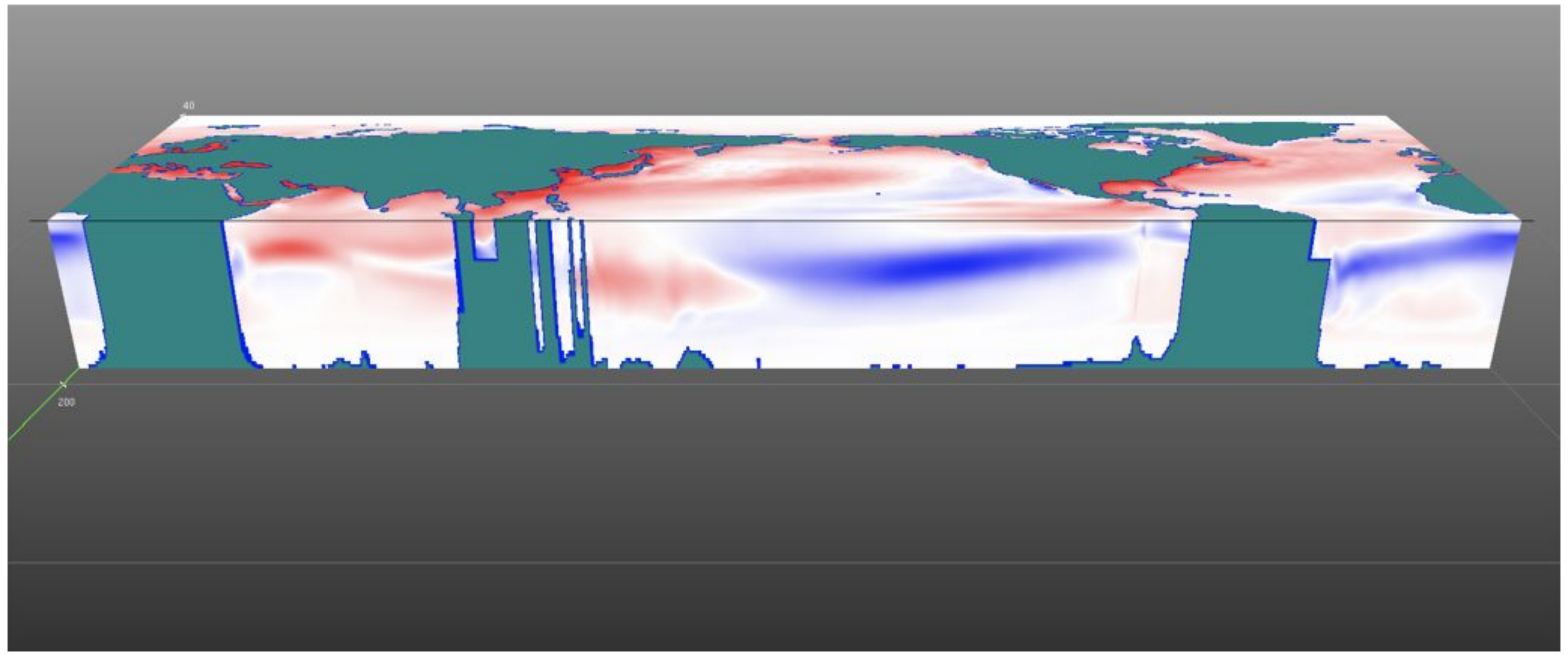
Case Study: Spark vs. MPI

- **Numerical linear algebra (NLA)** using Spark vs. MPI
- Computations performed on NERSC supercomputer **Cori** Phase 1, a Cray XC40
    - 2,388 compute nodes
    - 128 GB RAM/node, 32 2.3GHz Haswell cores/node
    - Lustre storage system, Cray Aries interconnect

A. Gittens et al. “Matrix factorizations at scale: A comparison of scientific data analytics in Spark and C+MPI using three case studies”, 2016 IEEE International Conference on Big Data (Big Data), pages 204–213, Dec 2016.

Case Study: Spark vs. MPI

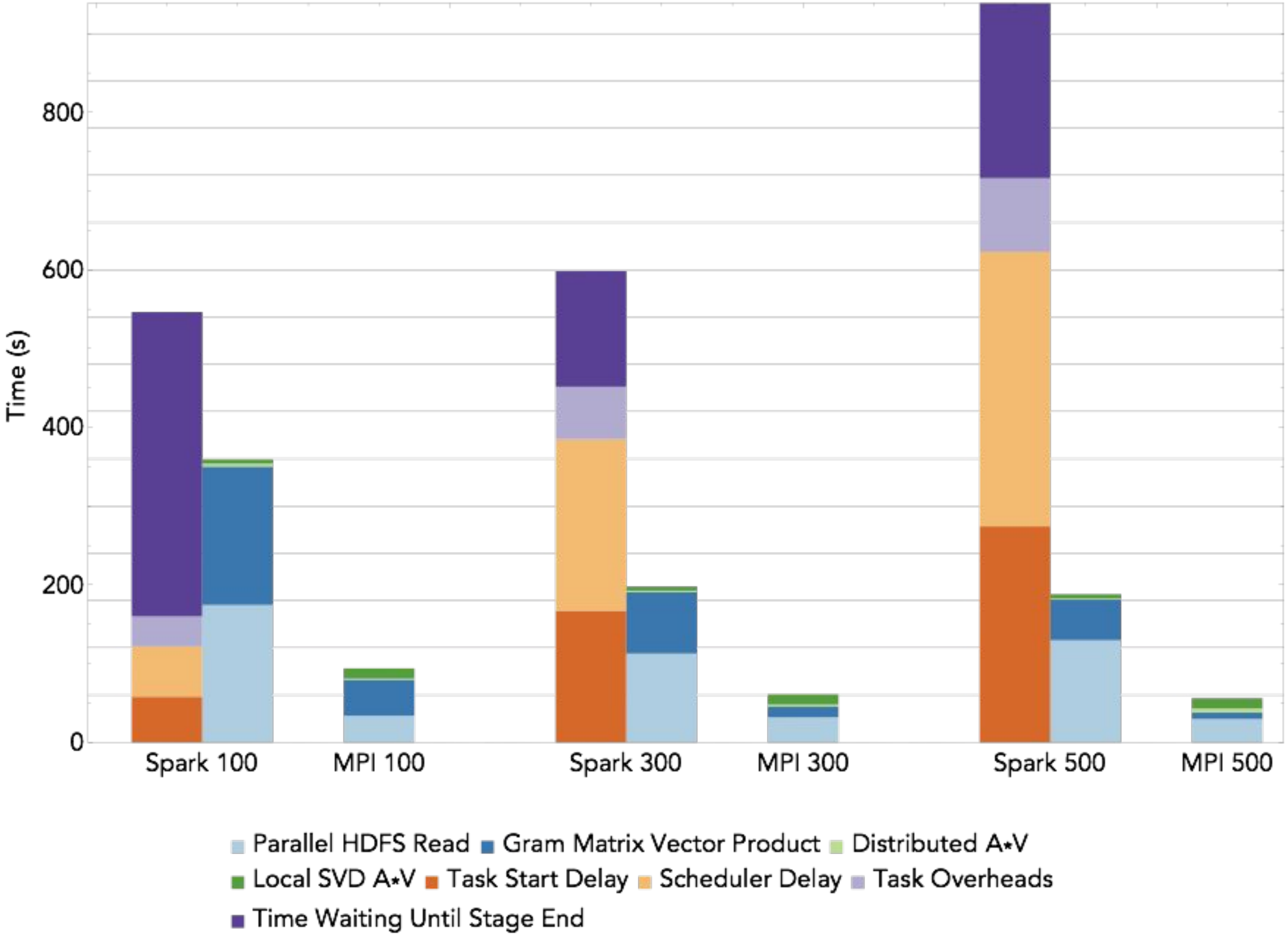
- **Numerical linear algebra (NLA)** using Spark vs. MPI
- Matrix factorizations considered include *truncated Singular Value Decomposition (SVD)*
- Data sets include
  - Oceanic temperature data - 2.2 TB
  - Atmospheric data - 16 TB



A. Gittens et al. “Matrix factorizations at scale: A comparison of scientific data analytics in Spark and C+MPI using three case studies”, 2016 IEEE International Conference on Big Data (Big Data), pages 204–213, Dec 2016.

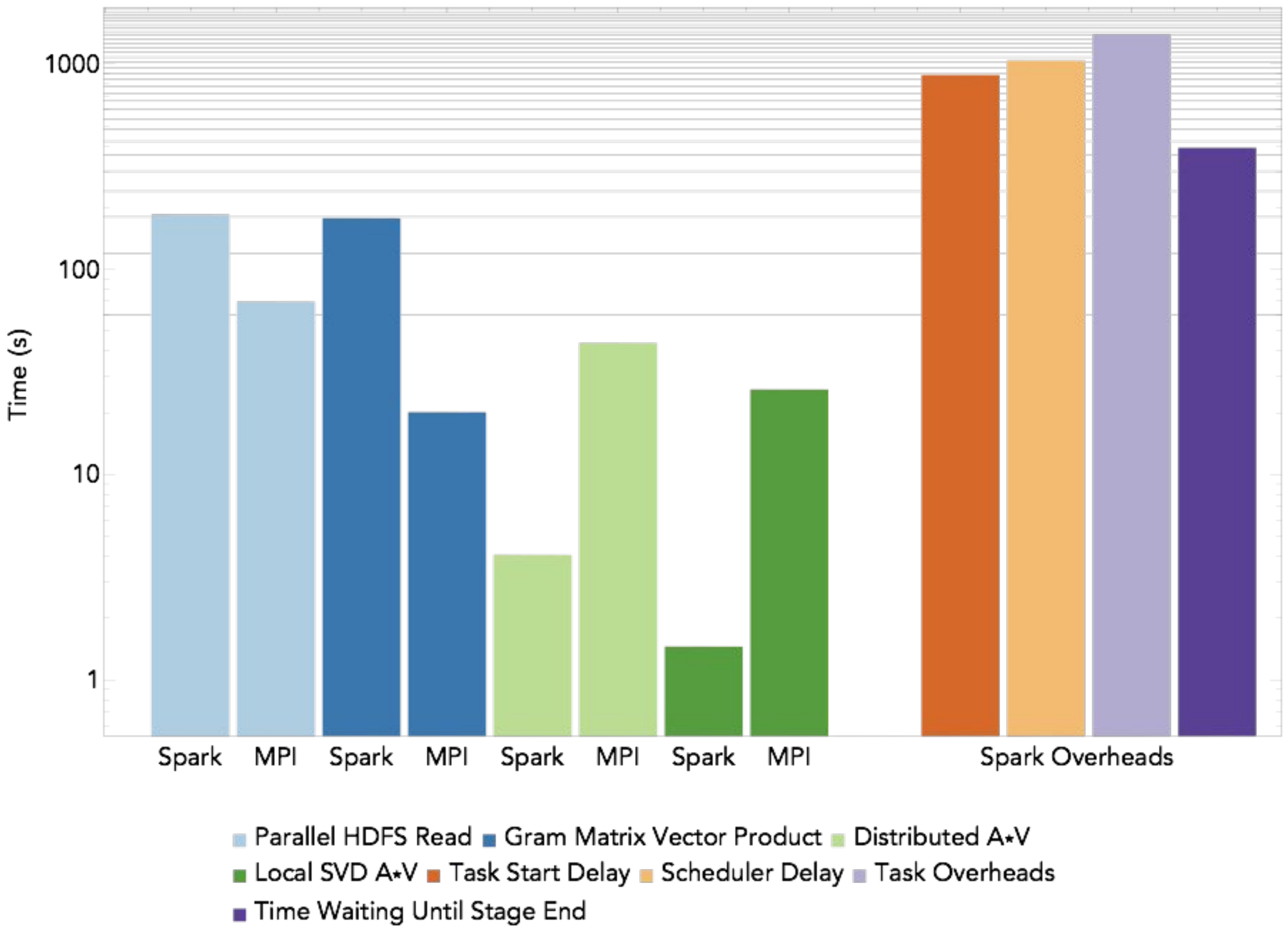
Case Study: Spark vs. MPI

Rank 20 SVD of 2.2TB ocean temperature data



Case Study: Spark vs. MPI

Rank 20 SVD of 16TB atmospheric data using 48K+ cores



Case Study: Spark vs. MPI

- With favorable data (tall and skinny) and well-adapted algorithms, linear algebra in Spark is 2x-26x slower than MPI when I/O is included
- Spark's overheads:
  - Orders of magnitude higher than the actual computation times
  - Anti-scale
- **The gaps in performance suggest it may be better to interface with MPI-based codes from Spark**

- **Alchemist** interfaces between Apache Spark and *existing* or *custom* MPI-based libraries for linear algebra, machine learning, *etc.*
- ***Goal***:
  - Use Spark for regular data analysis workflow
  - When computationally intensive calculations are required, call relevant MPI-based codes from Spark using Alchemist, send results to Spark
- Combine **high productivity** of Spark with **high performance** of MPI

Alchemist

- **Target users:**
    - ***Scientific community:*** Use Spark for analysis of large scientific datasets by calling existing MPI-based libraries where appropriate
    - ***Machine learning practitioners*** and ***data analysts***:
        - Better performance of a wide range of large-scale, computationally intensive ML and data analysis algorithms
        - For instance, SVD for principal component analysis, recommender systems, leverage scores, etc.

Basic Framework

- **Alchemist**: Acts as bridge between Spark and MPI-based libraries
- **Alchemist-Client Interface:** API for user, communicates with Alchemist via TCP/IP sockets
- **Alchemist-Library Interface:** Shared object, imports MPI library, provides generic interface for Alchemist to communicate with library

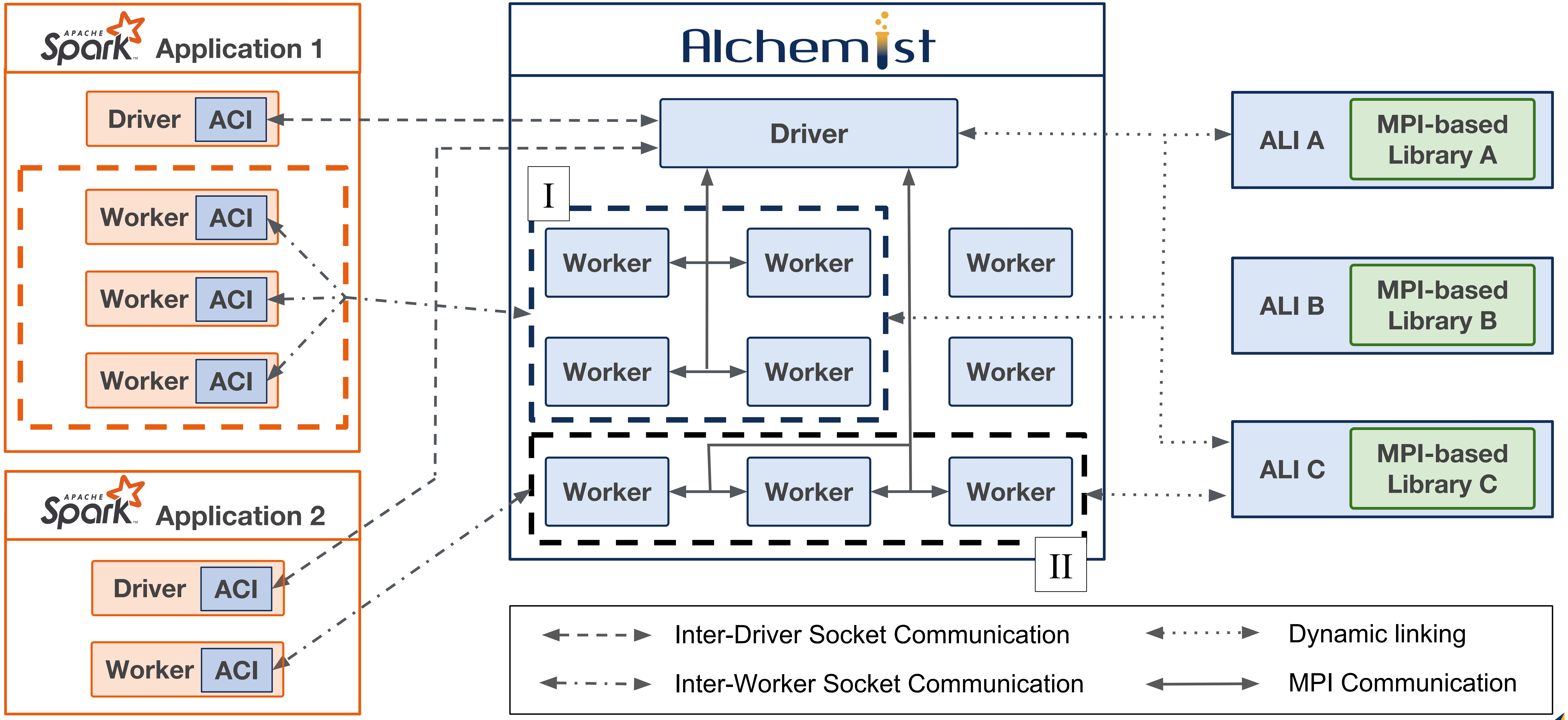
Basic Framework

- **Basic workflow:**
  - Spark application sends distributed dataset from RDD (`IndexedRowMatrix`) to Alchemist via TCP/IP sockets using ACI
  - Spark application tells Alchemist what MPI-based code should be called
  - Alchemist loads relevant MPI-based library, calls function, sends results to Spark

Basic Framework

- Alchemist can also load data from file
- Alchemist needs to store distributed data in appropriate format that can be used by MPI-based libraries:
  - Candidates: ***ScaLAPACK***, ***Elemental***, ***PLAPACK***
  - Alchemist currently uses Elemental, support for ScaLAPACK under development

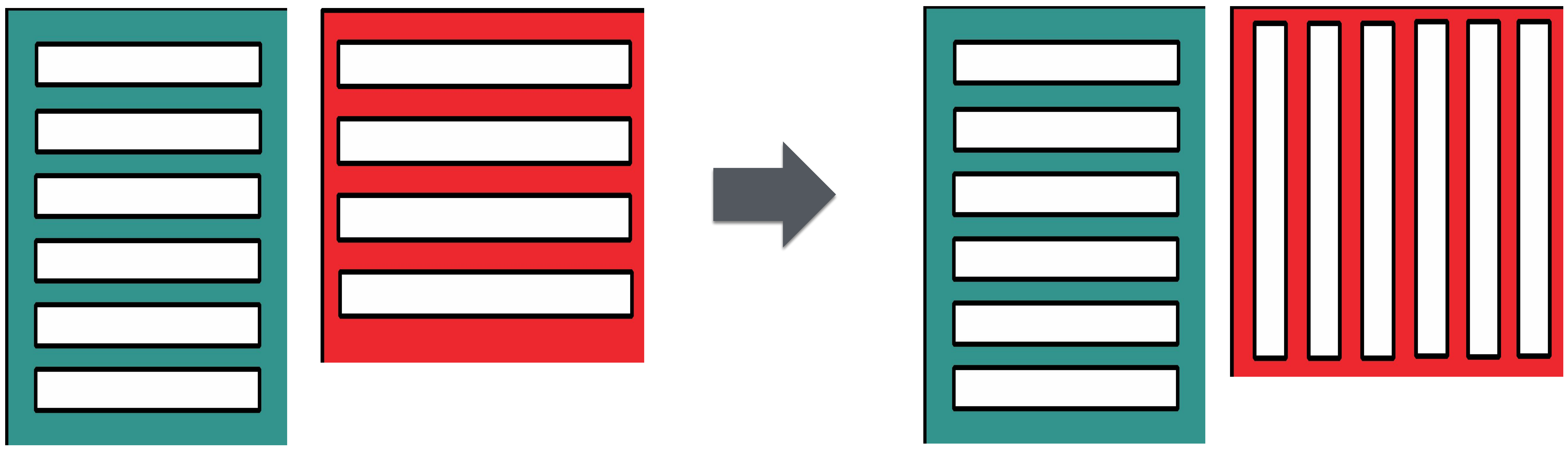
Alchemist Architecture



Sample API

```
import alchemist.{Alchemist, AlMatrix}
import alchemist.libA.QRDecomposition      // libA is sample MPI lib

// other code here ...

// sc is instance of SparkContext
val ac = new Alchemist.AlchemistContext(sc, numWorkers)
ac.registerLibrary("libA", ALIlibALocation)

// maybe other code here ...

val alA = AlMatrix(A)                         // A is IndexedRowMatrix

// routine returns QR factors of A as AlMatrix objects
val (alQ, alR) = QRDecomposition(alA)

// send data from Alchemist to Spark once ready
val Q = alQ.toIndexedRowMatrix()              // convert AlMatrix alQ to RDD
val R = alR.toIndexedRowMatrix()              // convert AlMatrix alR to RDD

// maybe other code here ...

ac.stop()                                     // release resources once no longer required
```

Example: Matrix Multiplication



- Requires expensive shuffles in Spark, which is impractical:
  - Matrices/RDDs are row-partitioned
  - one matrix must be converted to be column-partitioned
  - *Requires an all-to-all shuffle that often fails* once the matrix is distributed

Example: Matrix Multiplication

GB/nodes	*Spark+Alchemist*				*Spark*
	Send (s)	**Multiplication (s)**	**Receive (s)**	**Total (s)**	**Total (s)**
0.8/1	5.90±2.17	6.60±0.07	2.19±0.58	**14.68±2.69**	**160.31±8.89**
12/1	16.66±0.88	75.69±0.42	19.43±0.45	**111.78±1.26**	**809.31±51.9**
56/2	32.50±2.88	178.68±24.8	55.83±0.37	**267.02±27.38**	**-Failed-**
144/4	69.40±1.85	171.73±0.81	66.80±3.46	**307.94±4.57**	**-Failed-**

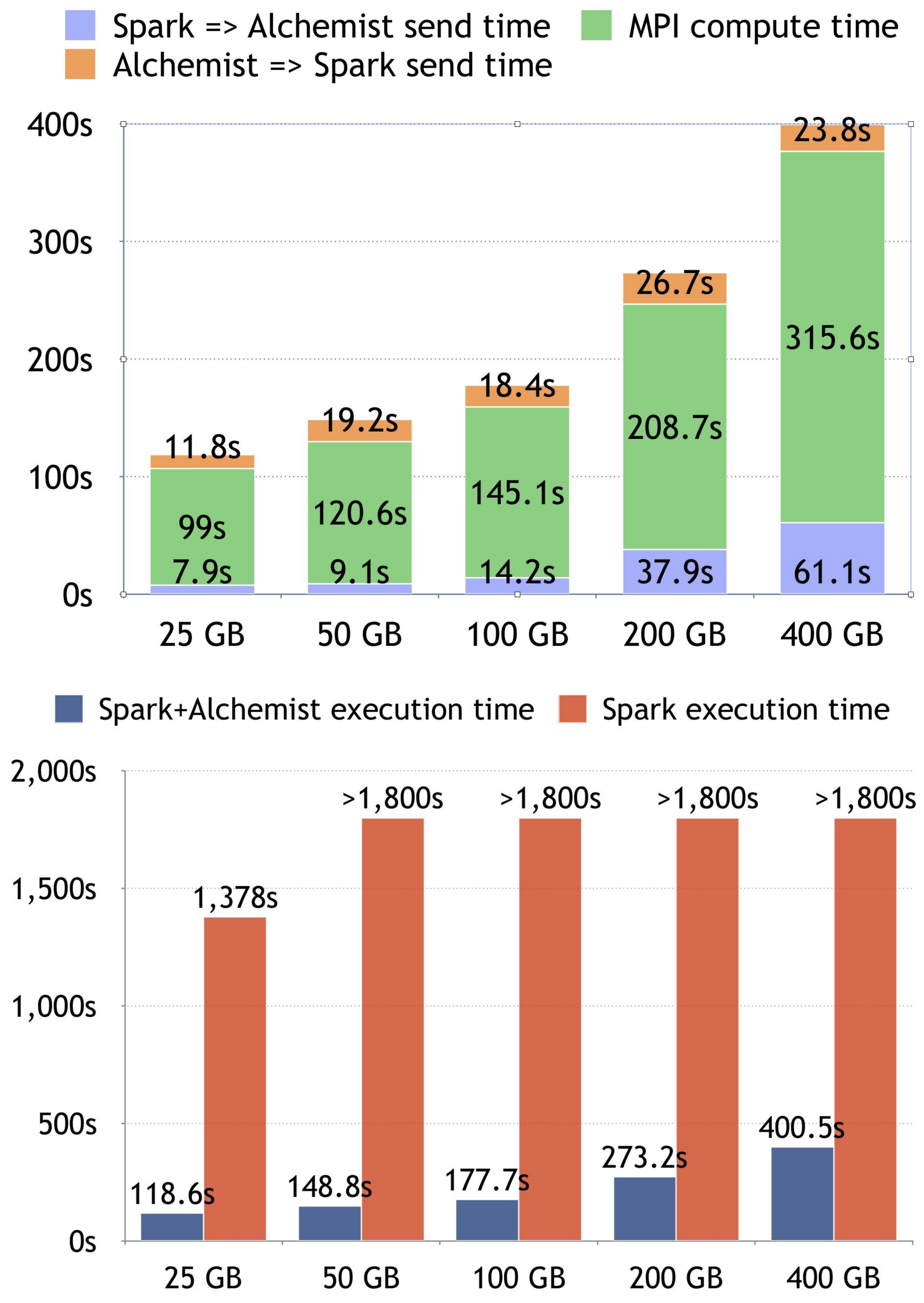
- Generated random matrices and used same number of Spark and Alchemist nodes
- Take-away: *Spark's matrix multiply is slow even on one executor, and unreliable once there are more*

Example: Truncated SVD

- Use Alchemist and MLlib to get rank 20 truncated SVD
- Experiments run on NERSC supercomputer Cori
- Each node of Cori has 128GB RAM and 32 cores

Experiment Setup

- Spark: 22 nodes; Alchemist: 8 nodes
- A: m-by-10K, where m = 5M, 2.5M, 1.25M, 625K, 312.5K
- Ran jobs for at most 30 minutes (1800 s)

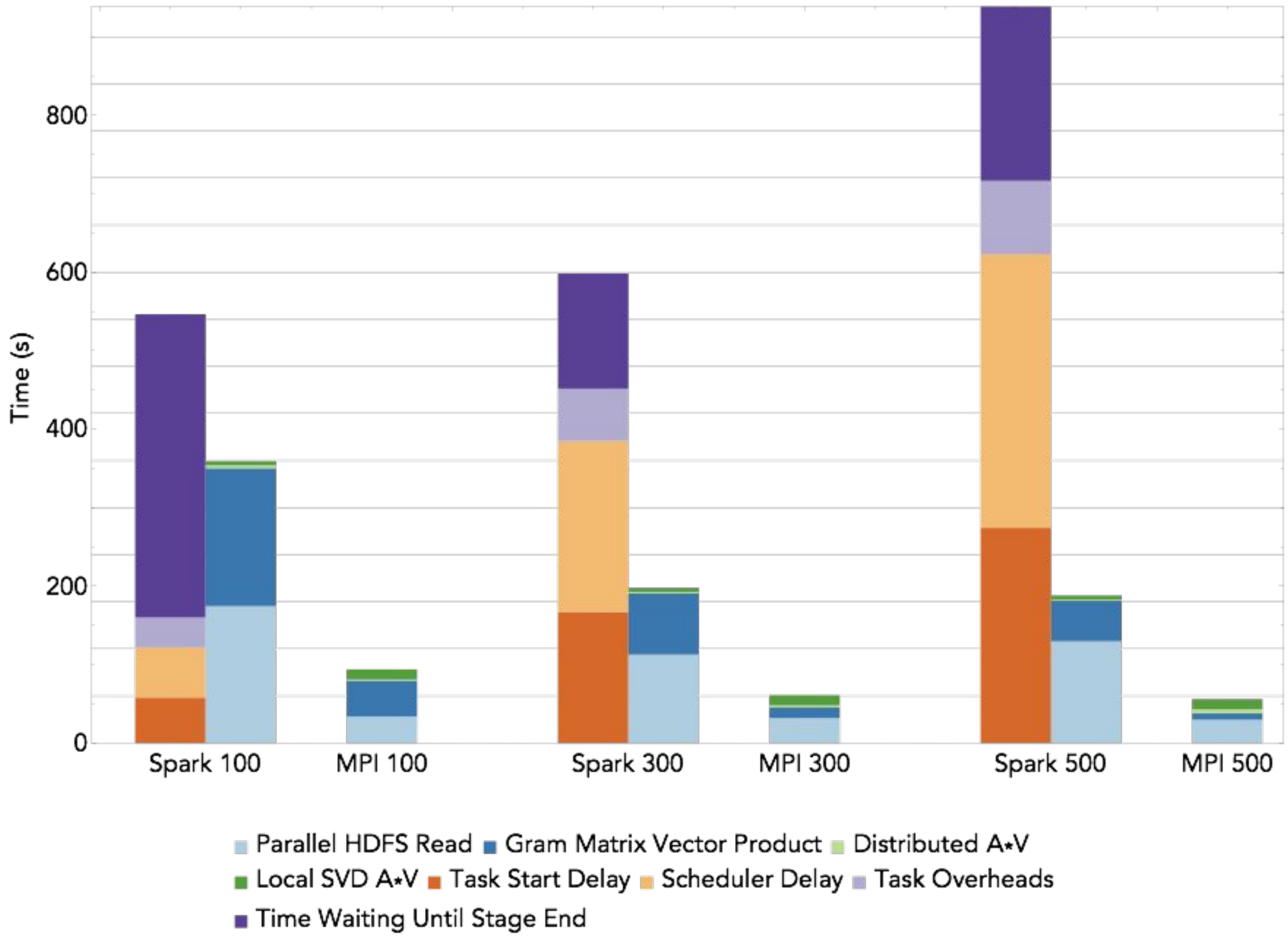


Case Study: Spark vs. MPI

Rank 20 SVD of

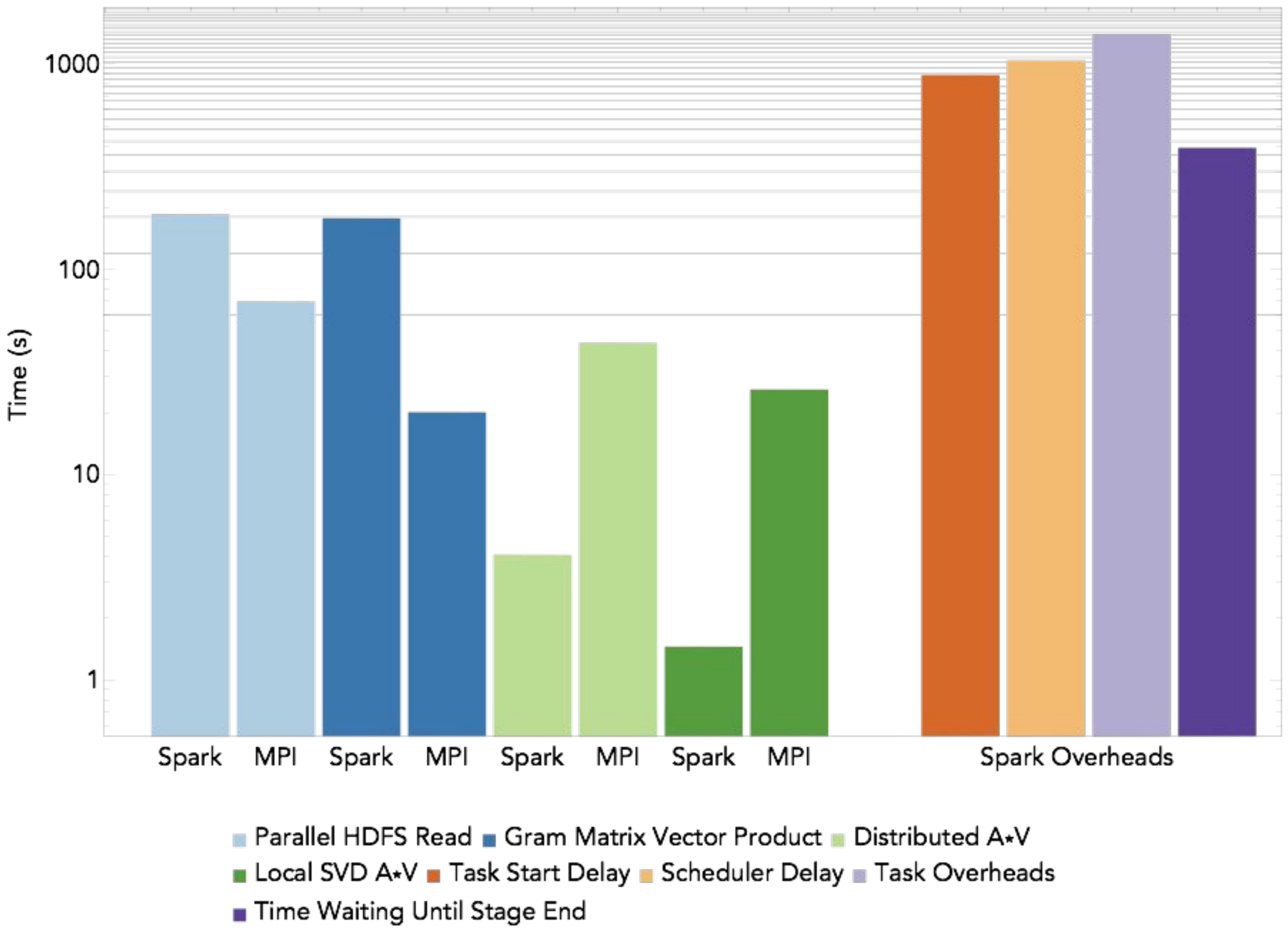
2.2TB ocean

temperature data



Case Study: Spark vs. MPI

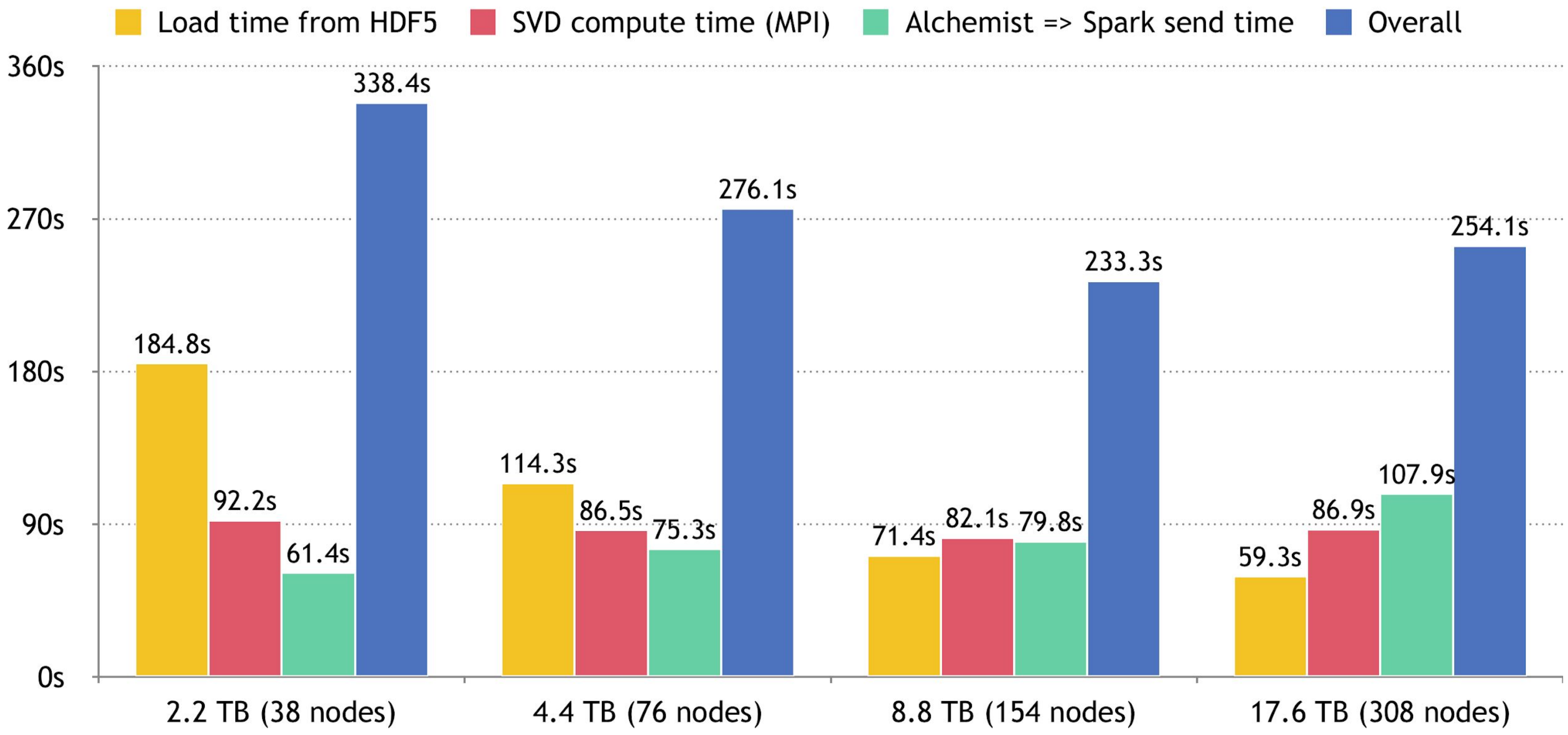
Rank 20 SVD of 16TB atmospheric data using 48K+ cores



Example: Truncated SVD

Experiment Setup

- 2.2TB (6,177,583-by-46,752) ocean temperature data read in from HDF5 file
- Data replicated column-wise



Upcoming Features

- **PySpark, SparkR ⇔ MPI Interface**
  - Interface for Python => PySpark support
  - Future work: Interface for R
- **More Functionality**
  - Support for sparse matrices
  - Support for MPI-based libraries built on ScaLAPACK
- **Alchemist and Containers**
  - Alchemist running in Docker and Kubernetes
  - Will enable Alchemist on clusters and the cloud

Limitations and Constraints

- **Two copies of data in memory**, more during a relayout during computation
- **Data transfer overhead** between Spark and Alchemist when data on different nodes
  - Subject to network disruptions and overload
- **MPI is not fault tolerant or elastic**
- **Lack of MPI-based libraries for machine learning**
  - No equivalent to MLlib currently available, closest is MaTEx
- On Cori, **need to run Alchemist and Spark on separate nodes** -> more data transfer over interconnects -> larger overheads

Future Work

- **Apache Spark ⇔ X Interface**
  - Interest in connecting Spark with other libraries for distributed computing (e.g. Cray Chapel, Apache REEF)
- **Reduce communication costs**
  - Exploit locality
  - Reduce number of messages
  - Use communication protocols designed for underlying network infrastructure
- **Run as network service**
- **MPI computations with (basic) fault tolerance and elasticity**

Thanks to Cray Inc., DARPA and NSF for financial support

Kai Rothauge
kai.rothauge@berkeley.edu
github.com/kai-rothauge/alchemist

github.com/project-alchemist/

References

- A. Gittens, K. Rothauge, M. W. Mahoney, *et al.*, “Alchemist: Accelerating Large-Scale Data Analysis by offloading to High-Performance Computing Libraries”, 2018, *Proceedings of the 24th ACM SIGKDD International Conference,* Aug 2018, to appear
- A. Gittens, K. Rothauge, M. W. Mahoney, *et al.*, “Alchemist: An Apache Spark ⇔ MPI Interface”, 2018, to appear in *CCPE Special Issue on Cray User Group Conference 2018*

riselab UC Berkeley