Today's content

- Resilient Distributed Datasets(RDDs) ------ Spark and its data model (supplemental)
- Parallel Algorithms

Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing -- Spark

By Matei Zaharia, etc.

Presented by 齐琦
海南大学

Abstract

- Resilient Distributed Datasets (RDDs)
  - A distributed memory abstraction; perform in-memory computations on large clusters in a fault-tolerant manner.
- Motivation
  - Current computing frameworks handle inefficiently: iterative algorithms and interactive data mining tools
- Expressive enough to apply to wide range of computations
- Spark, implemented RDDs.

Introduction

- Cluster computing frameworks
  - MapReduce, Dryad, etc. , widely adopted for large-scale data analytics
  - High-level operators for parallel computations, hide work distribution and fault tolerance
- Lack abstractions leveraging distributed memory
  - Emerging applications: reuse intermediate results across multiple computations

Introduction, contd.

- Data reuse
  - Iterative machine learning, graph algorithms
  - PageRank, K-means clusters, logistic regression
- Interactive data mining
  - Multiple ad-hoc queries on same data subset

challenge

- Define a programming interface providing fault tolerance efficiently
- Systems (existing abstractions for in-memory storage on clusters) before
  - Distributed shared memory
  - Key-value stores
  - Databases
  - Piccolo
  - Its interface based on fine-grained updates to mutable state(table cells)
  - Only way providing fault tolerance – replicate data across machines; log updates across machines
    - Expensive for data-intensive workloads
    - Copy large data over cluster network, bandwidth far lower than RAM; substantial storage overhead

Evaluation

- Through both micro-benchmarks and measurements of user applications
- Up to 20X faster than Hadoop for iterative applications
- A real-world data analytics speed up by 40X
- Interactively scan a 1TB dataset , 5-7s latency
- Show its generality
  - Implemented Pregel and HaLoop on top of Spark (200 lines of code each)

Aspect	RDDs	Distr. Shared Mem.
Reads	Coarse- or fine-grained	Fine-grained
Writes	Coarse-grained	Fine-grained
Consistency	Trivial (immutable)	Up to app / runtime
Fault recovery	Fine-grained and low-overhead using lineage	Requires checkpoints and program rollback
Straggler mitigation	Possible using backup tasks	Difficult
Work placement	Automatic based on data locality	Up to app (runtimes aim for transparency)
Behavior if not enough RAM	Similar to existing data flow systems	Poor performance (swapping?)

Table 1: Comparison of RDDs with distributed shared memory.

Contd.

- Immutable nature mitigates slow nodes by running backup copies of slow tasks
  - Hard on DSM, two copies access same memory locations, interfere with each other's updates
- For bulk operations on RDDs, tasks can be scheduled based on data locality to improve performance
- Partitions not fit in RAM can be stored on disk, and provide similar performance to current data-parallel systems

RDDs limits

- Best suited for batch applications (batch analytics)
  - Apply same operation to all elements of a dataset
  - Each transformation as one step in a lineage graph; recover lost partitions without log large amount data
- Less suitable for applications that make asynchronous fine-grained updates to shared state
  - Storage system for a web application, or incremental web crawler
  - For these, more efficient to use traditional update logging and data check-pointing, e.g. database

Example applications: Logistic regression

- Machine learning algorithms
  - Many are iterative in nature (iterative optimization procedures)

```
val points = spark.textFile(...)
                  .map(parsePoint).persist()
var w = // random initial vector
for (i <- 1 to ITERATIONS) {
  val gradient = points.map{ p =>
    p.x * (1/(1+exp(-p.y*(w dot p.x)))-1)*p.y
  }.reduce((a,b) => a+b)
  w -= gradient
}
```

Overview

- Implemented Spark in 14,000 lines of Scala
- Runs over Mesos cluster manager, allowing it share resources with Hadoop, MPI and other applications
  - Each Spark program as separate Mesos app, with its own driver and workers
  - resource sharing between apps handled by Mesos
- Spark can read data from any Hadoop input source(e.g. HDFS or HBbase)
  - By Hadoop's existing input APIs

Job Scheduling

- When runs an action(e.g. count or save) on an RDD, scheduler examines RDD's lineage graph to build a DAG of stages to execute
  - Each stage contains as many pipelined transformations with narrow dependencies as possible
  - Stage boundaries are shuffle ops required for wide dependencies
  - Scheduler launches tasks to compute missing partitions from each stage, until computed target RDD

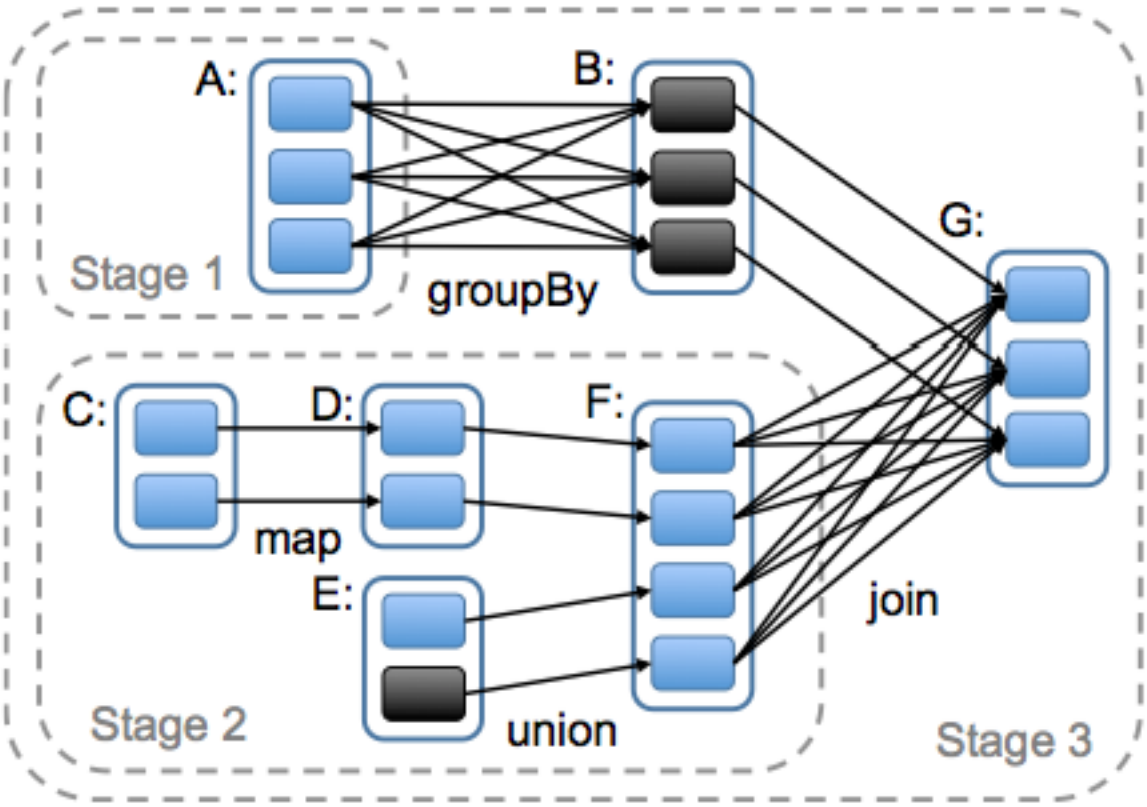


Figure 5: Example of how Spark computes job stages. Boxes with solid outlines are RDDs. Partitions are shaded rectangles, in black if they are already in memory. To run an action on RDD G, we build build stages at wide dependencies and pipeline narrow transformations inside each stage. In this case, stage 1's output RDD is already in RAM, so we run stage 2 and then 3.

- Scheduler assigns tasks to machines based on data locality using delay scheduling
- If a task fails, re-run it on another node as long as its stage's parents are still available
  - If stages unavailable, resubmit tasks to compute missing partitions in parallel
- Not yet tolerate scheduler failures

Interpreter Integration

- How Scala interpreter works
  - Compile a class for each line typed by user, loading it into JVM, invoke a function on it
  - Types var x = 6 ; then println(x)
    - Define a class called Line1 containing x
    - Second line compiled to println(Line1.getInstance().x)

Changes to interpreter in Spark

- Class shipping
  - Worker nodes fetch bytecode for classes on each line, and served over HTTP
- Modified code generation
  - To reference instance of each line object directly

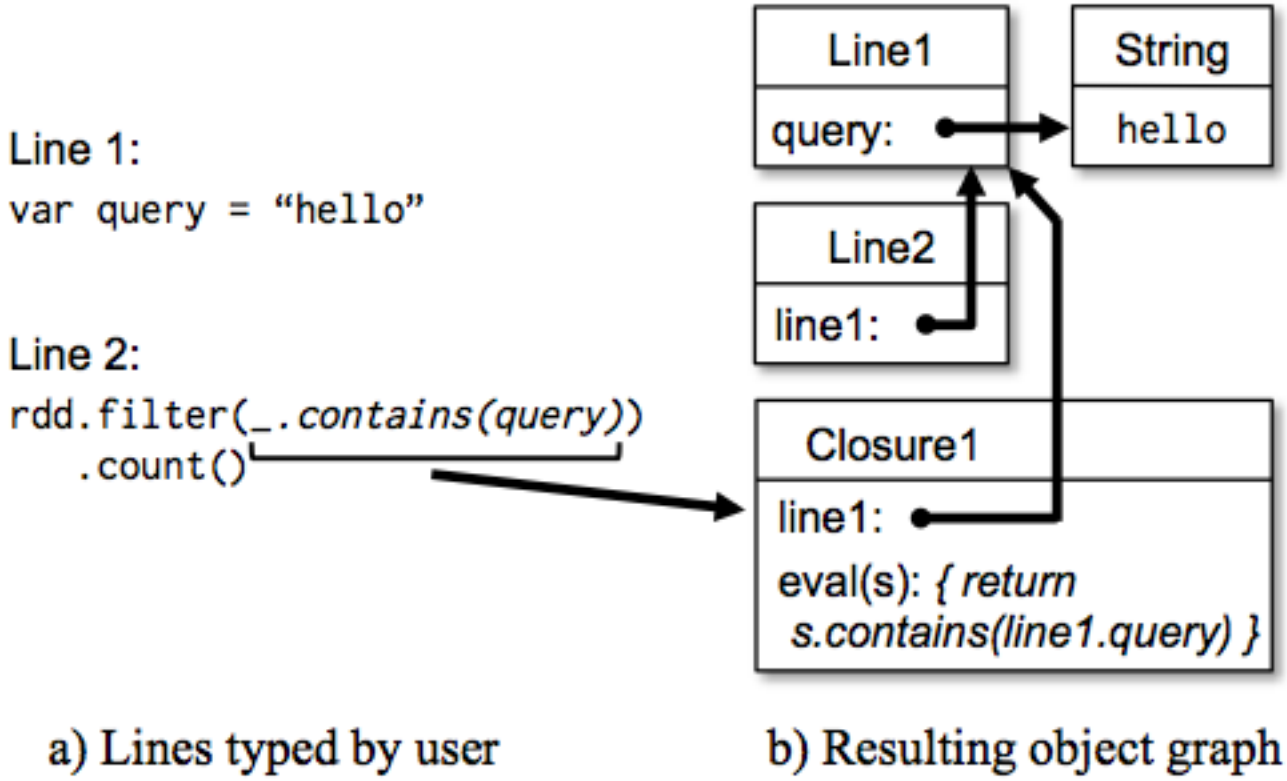


Figure 6: Example showing how the Spark interpreter translates two lines entered by the user into Java objects.

Support for Checkpointing

- Lineage time-consuming for RDDs with long lineage chains, so helpful to checkpoint some RDDs to stable storage
- Long lineage graphs containing wide dependencies
  - A node failure may cause loss of some slice of data from each parent RDD, requiring a full recomputation
- Spark provides API for checkpointing, leaves decision of which data to checkpoint to users

Evaluation

Overview

- Run on Amazon EC2
- Spark outperforms Hadoop by up to 20x in iterative machine learning and graph applications
  - Through avoiding I/O and deserialization costs by storing data in memory as Java objects
  - Applications written by users perform and scale well
  - When nodes fail, Spark can recover quickly by rebuilding only lost RDD partitions
  - Spark can query a 1 TB dataset interactively with latencies of 5—7 seconds

Settings

- Used m1.xlarge EC2 nodes with 4 cores, 15GB RAM
- Used HDFS for storage, with 256MB blocks
- Before each test, cleared OS buffer caches to measure IO costs accurately

Iterative Machine learning applications

- Two applications: logistic regression, k-means
- Systems to compare
  - Hadoop (0.20.2 stable release)
  - HadoopBinMem
    - Converts input data into low-overhead binary format, stores in an in-memory HDFS instance
  - Spark: implementation of RDDs

- Ran both algorithms for 10 iterations on 100 GB datasets using 25—100 machines
- Iteration time of k-means dominated by computation; logistic regression less compute-intensive and more sensible to time spent in deserialization and I/O

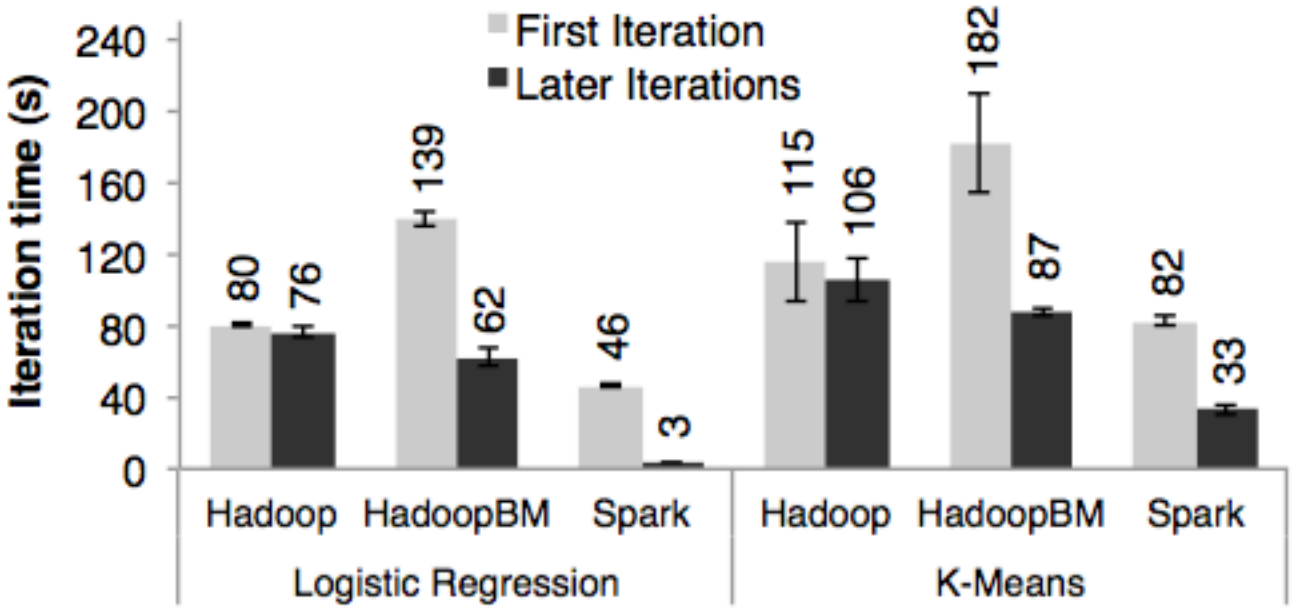


Figure 7: Duration of the first and later iterations in Hadoop, HadoopBinMem and Spark for logistic regression and k-means using 100 GB of data on a 100-node cluster.

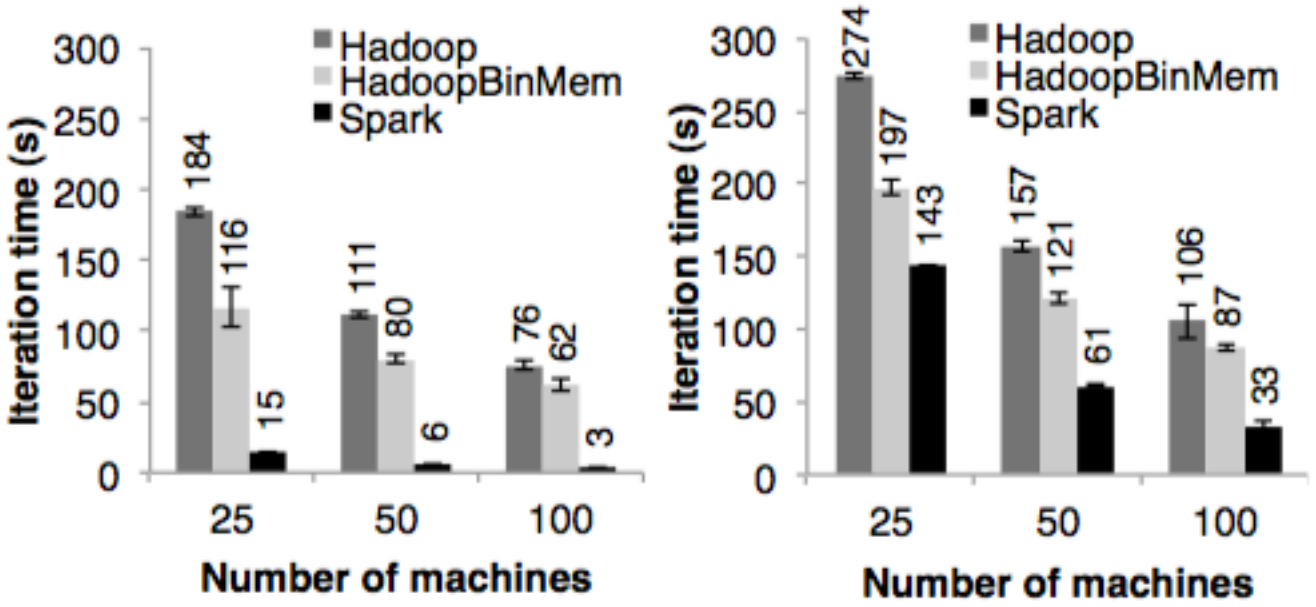


Figure 8: Running times for iterations after the first in Hadoop, HadoopBinMem, and Spark. The jobs all processed 100 GB.

- Spark outperformed even Hadoop with in-memory storage of binary data by a 20x margin
- Storing RDD elements as Java objects in memory, Spark avoids these overheads
  - Reading through HDFS introduced a 2-second overhead
  - Parsing text overhead was 7 seconds
  - Reading from in-memory file, converting pre-parsed binary data into Java objects took 3 seconds

PageRank experiment

- Dataset: 54GB wikipedia dump
- 10 iterations to process a link graph of 4 million articles
- In-memory storage alone provided Spark with a 2.4x speedup over Hadoop on 30 nodes
- Results also scaled nearly linearly to 60 nodes

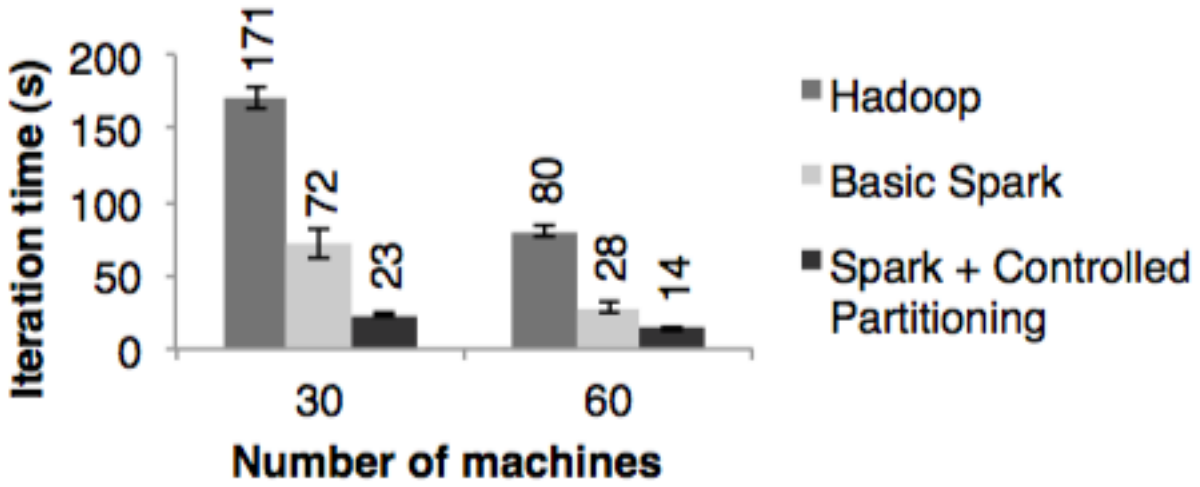


Figure 10: Performance of PageRank on Hadoop and Spark.

Fault Recovery

- Evaluated cost of reconstructing RDD partitions using lineage after a node failure in k-means application

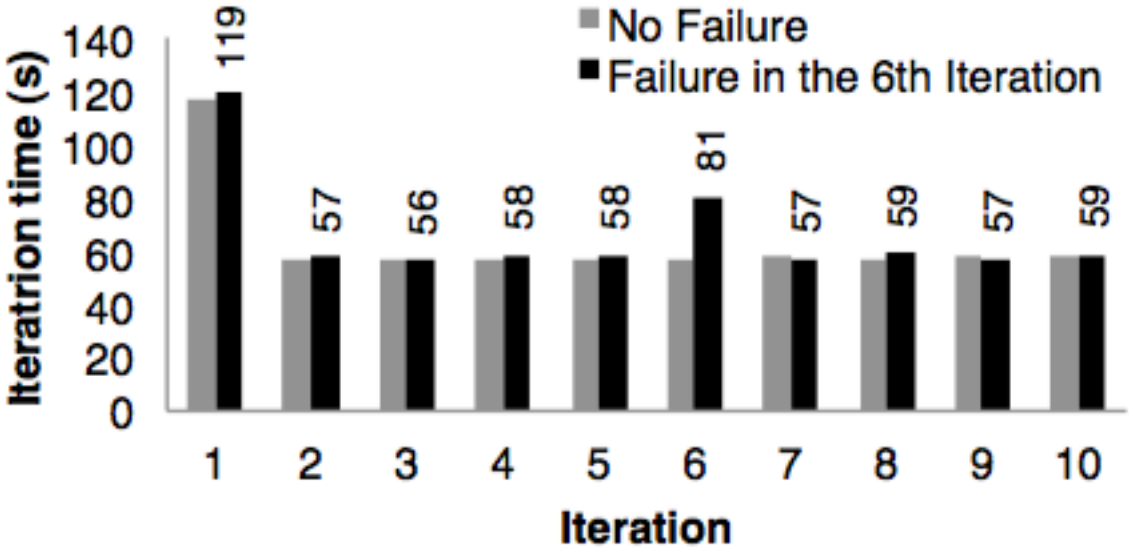


Figure 11: Iteration times for k-means in presence of a failure. One machine was killed at the start of the 6th iteration, resulting in partial reconstruction of an RDD using lineage.

Behavior with insufficient memory

- What if there's not enough memory to store a job's data
- Performance degrades gracefully with less space

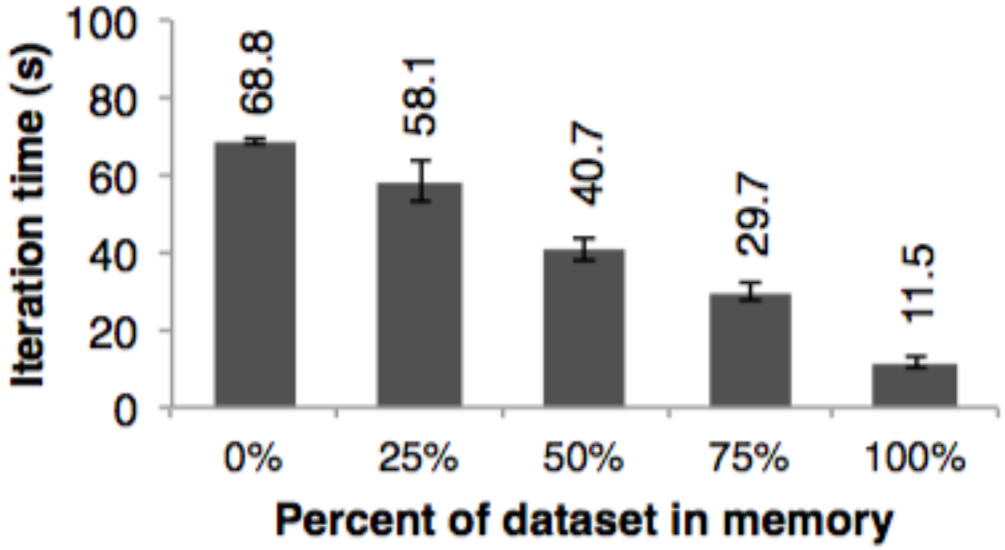
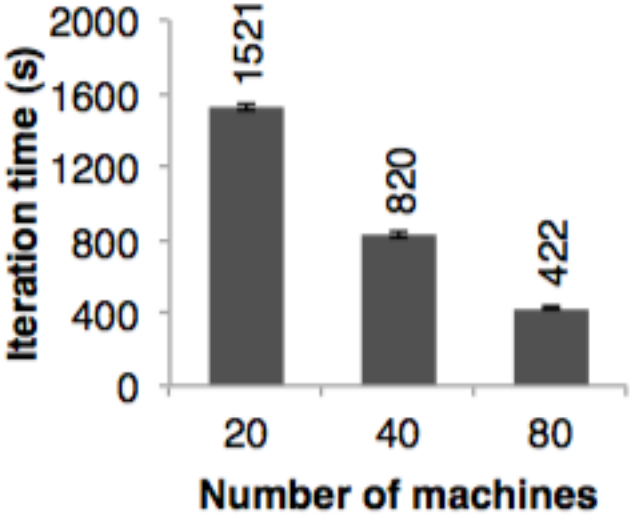


Figure 12: Performance of logistic regression using 100 GB data on 25 machines with varying amounts of data in memory.

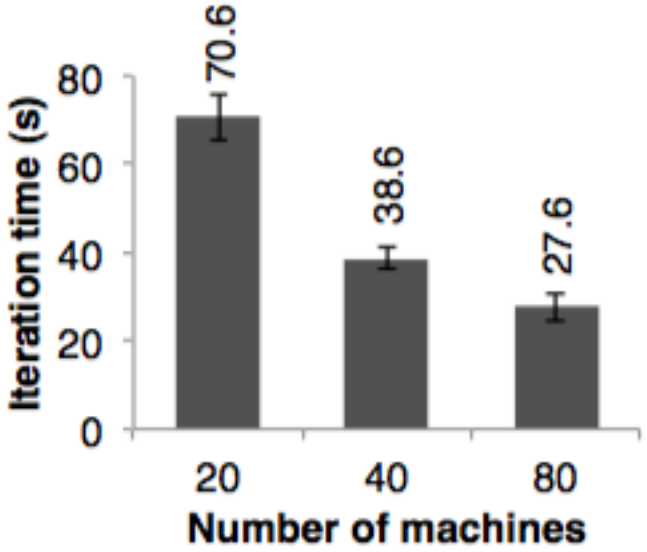
User applications built with Spark

- In-memory analytics
  - Conviva Inc, video distribution company, used Spark accelerate data analytics reports previously ran over Hadoop
  - Speed up report by 40x
  - A report on 200GB compressed data took 20 hours on Hadoop cluster, now runs in 30 minutes using only two Spark machines
  - Spark program only required 96 GB RAM, because only stored rows and columns matching user's filter in a RDD, not whole decompressed file

- Traffic Modeling
  - Mobile Millennium project at Berkeley, parallelized learning algorithm for inferring road traffic congestion from sporadic automobile GPS measurements
  - Source data, 10,000 link road network for a metropolitan area, 600,000 samples of point-to-point trip times
  - Traffic model estimates time taking to travel across road links
  - EM algorithm trained model, repeats two map and reduceByKey steps iteratively
  - Scales nearly linearly from 20 to 80 nodes with 4 cores each



(a) Traffic modeling



(b) Spam classification

Figure 13: Per-iteration running time of two user applications implemented with Spark. Error bars show standard deviations.

- Twitter Spam Classification
  - Monarch project at Berkeley used Spark to identify link spam in Twitter messages
  - Implemented a logistic regression classifier on top of Spark
  - Training a classifier over 50GB subset of the data: 250,000 URLs and 10to7 features/dimensions
  - Scaling not as close to linear due to a higher fixed communication cost per iteration

Interactive Data Mining

- Analyze 1TB Wikipedia page view logs(2 years of data)
- Used 100 m2.4xlarge EC2 instances with 8 cores and 68GB of RAM each
- Queries
  - All pages
  - Pages with titles matching a given word
  - Pages with titles partially matching a word
  - Each query scanned the entire input data
- Took Spark 5—7 seconds
  - 170s from disk

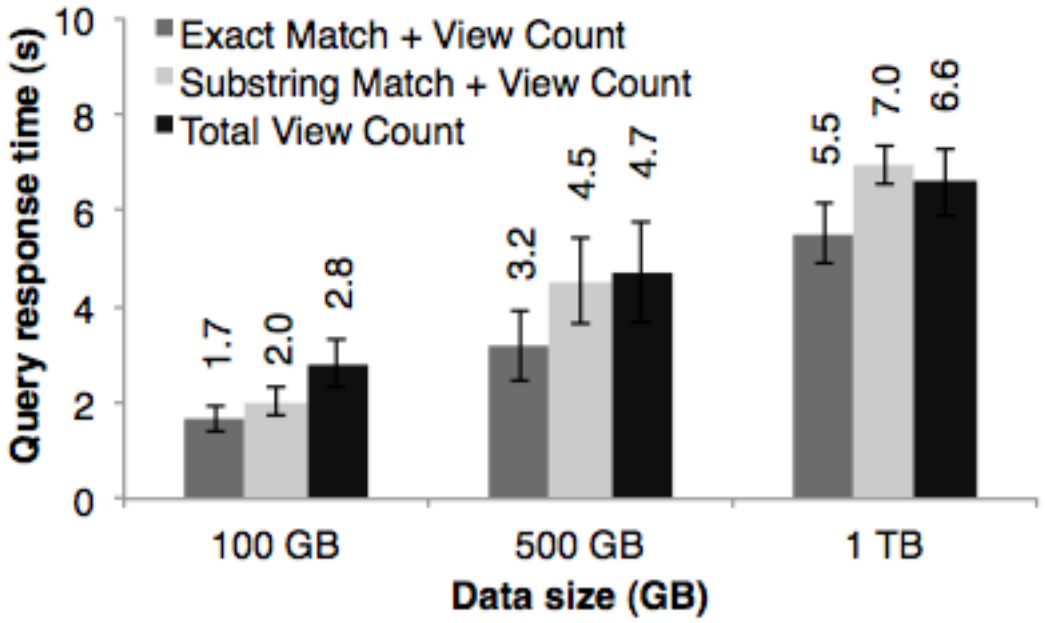


Figure 14: Response times for interactive queries on Spark, scanning increasingly larger input datasets on 100 machines.

Discussion

- RDDs immutable nature and coarse-grained transformations, suitable for a wide class of applications
- Can express many cluster programming models that so far been proposed as separate frameworks, allowing users tom compose these models in one program, and share data between them.

Expressing existing programming models

- RDDs can efficiently express a number of cluster programming models that have been proposed independently
- Also capture Optimizations
  - Keep specific data in memory
  - Partitioning to minimize communication
  - Recover from failures efficiently

Expressible models by RDDs

- MapReduce
  - By flatMap, groupByKey, or reduceByKey
- DryadLINQ
  - All bulk operators that correspond directly to RDD transformations (map, groupByKey, join, etc.)
- SQL
  - Like DryadLINQ expressions, SQL queries perform data-parallel operations on sets of records

Contd.

- Pregel
  - Google's specialized model for iterative graph applications
  - A program runs series coordinated "supersteps"; on each one, each vertex runs a user function that update state associated with vertex, change graph topology, and send messages to other vertices for use in the next superstep
  - It can express many graph algorithms, shortest paths, bipartite matching, PageRank

Contd.

- Pregel
  - Pregel applies same user function to all vertices on each iteration
  - Thus, can store vertex states for each iteration in an RDD, perform a bulk transformation (flatMap) to apply function and generate an RDD of messages
  - Then join this RDD with vertex states to perform message exchange
  - Have implemented Pregel as 200-line library on top of Spark

- Iterative MapReduce
  - Such as HaLoop, Twister, provide iterative MapReduce model, user gives the system a series of MapReduce jobs to loop
  - HaLoop implemented as a 200-line library using Spark

- Batched Stream Processing
  - Incremental processing systems for applications that periodically update a result with new data
  - E.g. , statistics about ad clicks every 15 minutes should combine intermediate state from previous 15-minute window with data from new logs
  - They perform bulk operations, but store application state in distributed file systems
  - Placing intermediate state in RDDs would speed up their processing

Why RDDs able to express these diverse programming models?

- Restrictions on RDDs have little impact in many parallel applications (though RDDs can only be created through bulk transformations)
- Many parallel programs naturally apply the same operation to many records – make them easy to express
- Immutability of RDDs not an obstacle, because on can create multiple RDDs to represent versions of the same dataset
- Many today's MapReduce applications run over file systems that do not allow updates to files, such as HDFS

Why previous frameworks have not offered the same level of generality?

- Because those systems explored specific problems that MapReduce and Dryad do not handle well, such as iteration, without observing that the common cause of these problems was a lack of data sharing abstractions

Leveraging RDDs for Debugging

- Initially designed RDDs to be deterministically recomputable for fault tolerance; it also facilitates debugging
- By logging lineage of RDDs created during a job
  - Reconstruct these RDDs later and let user query them interactively
  - Re-run any task from the job in a single-process debugger, by recomputing RDD partitions it depends on

- Traditional replay debuggers for general distributed systems
  - Must capture or infer the order of events across multiple nodes
- Adds virtually zero recording overhead because only the RDD lineage graph needs to be logged

Comparison with related work

- Data flow models like MapReduce, Dryad, and Ciel support operators for processing data but share it through stable storage systems
- RDDs represent a more efficient data sharing abstraction than stable storage because they avoid cost of data replication, I/O and serialization

Comparing contd.

- Previous high-level programming interfaces for data flow systems, like DryadLINQ, FlumeJava, provide language-integrated APIs, user manipulates "parallel collections" through ops like map and join
  - They cannot share data efficiently across queries
- Spark's API on parallel collection model, not claim novelty for this interface, but by providing RDDs as storage abstraction behind this interface

Contd.

- Pregel supports iterative graph applications, Twister and HaLoop are iterative MapReduce runtimes
  - They perform data sharing implicitly for pattern of computation they support, not provide a general abstraction that user can employ to share data of their choice among ops of their choice
- RDDs provide a distributed storage abstraction explicitly and can thus support applications those specialized systems do not capture, e.g. interactive data mining

- Some systems expose shared mutable state to allow user to perform in-memory computation
  - Piccolo – run parallel functions that read and update cells in a distributed hash table
  - Distributed shared memory(DSM) systems and key-value stores like RAMCloud
- RDDs provide a higher-level programming interface based on operators like map, sort, join, more complicated functions than those
- Those systems implement recovery through checkpoints and rollback, more expensive than lineage-based strategy of RDDs

Lineage

- Long been a research topic in scientific computing and databases
- RDDs provide a parallel programming model, where fine-grained lineage is inexpensive to capture, so can be used for failure recovery
- RDDs apply lineage to persist in-memory data efficiently across computations, with cost of replication and disk I/O

Relational databases

- RDDs are conceptually similar to views in a database, and persistent RDDs resemble materialized views
- Databases typically allow fine-grained read-write access to all records, requiring logging of operations and data for fault tolerance, additional overhead to maintain consistency
- Coarse-grained transformation model of RDDs not required with these overheads.

Sponsorships

Nightingale, and our reviewers for their feedback. This research was supported in part by Berkeley AMP Lab sponsors Google, SAP, Amazon Web Services, Cloudera, Huawei, IBM, Intel, Microsoft, NEC, NetApp and VMWare, by DARPA (contract #FA8650-11-C-7136), by a Google PhD Fellowship, and by the Natural Sciences and Engineering Research Council of Canada.

Parallel Algorithms

Citation

- Blelloch, G. E. and B. M. Maggs (2010). Parallel algorithms. Algorithms and theory of computation handbook. J. A. Mikhail and B. Marina, Chapman & Hall/CRC: 25-25.

- Most today's algorithms are sequential
  - Perform operations in a sequential fashion
  - Speed of sequential computers improved at exponential rate many years; it's getting greater and greater cost
- Cost-effective improvements by building "parallel" computers --- perform multiple operations in a single step
- To solve problem efficiently on parallel computer, necessary design parallel algorithm

Sum of sequence of n numbers, example

- Pairing and summing step to repeat
- Parallelism in an algorithm can yield improved performance on many different kinds of computers
- Make distinction between parallelism in an algorithm and ability of any particular computer to perform multiple operations in parallel
- It's more difficult to build a general-purpose parallel computer than a general-purpose sequential computer

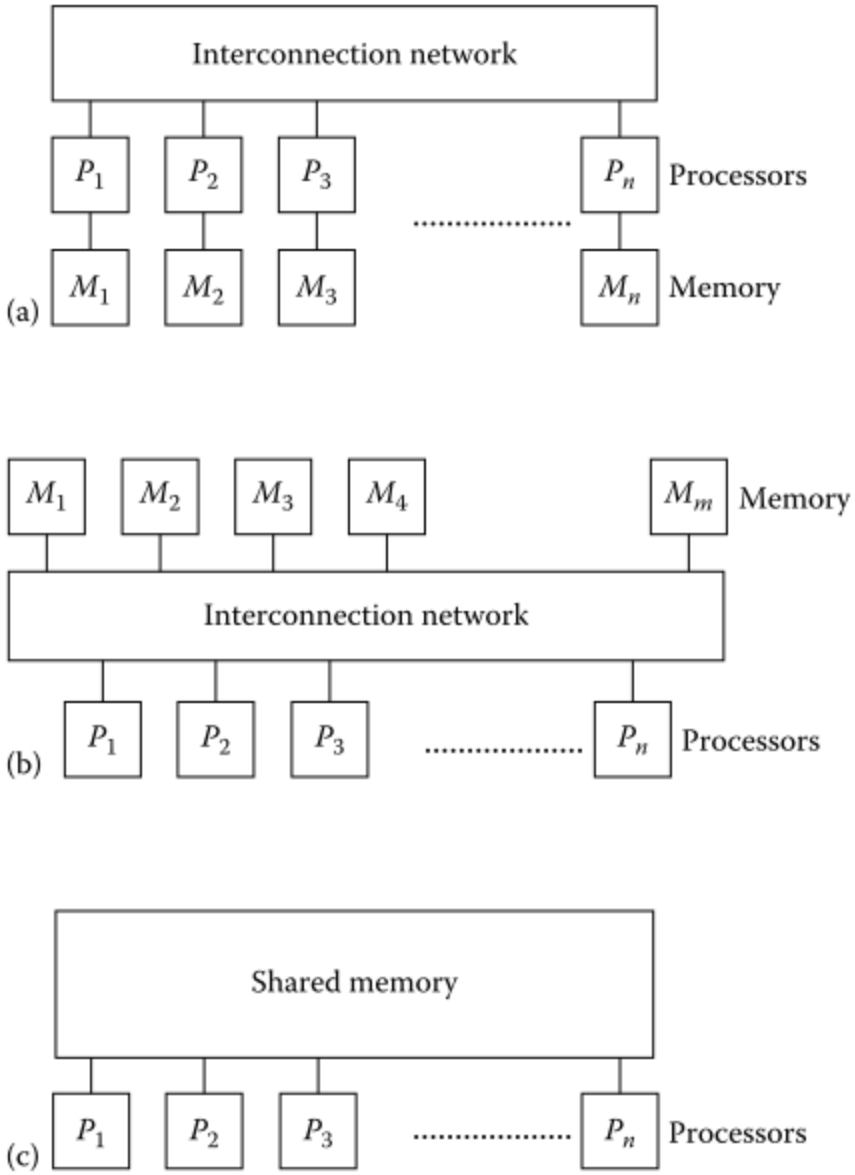
Modeling Parallel Computations

Parallel computation models

- Multiprocessor models
- Work-depth models

Multiprocessor Models

- Generalization of sequential RAM model where there is more processors
- Three basic types
  - Local-memory machine models
  - Modular memory machine models
  - Parallel random-access machine(PRAM) models



ıree types of multiprocessor machine models: (a) a local-memory machine model; (b) a modular

- They differ in way that memory can be accessed
  - Local-memory model --- time taken to access another process's memory, depend on communication network and memory access pattern by other processors
  - Modular memory model --- arraged, any processor to access any memory module is roughly uniform

- In a PRAM model
  - A processor can access any word of memory in single step
  - These accesses can occur in parallel
  - Controversial because no real machine lives up to ideal of unit-time access to shared memory
- Ultimate purpose of abstract model is not directly model a real machine, but help designer produce efficient algorithms
  - If an algorithm designed for a Prim model(or others), can be translated to runs efficiently on a real computer, then it succeeded.

Network Topology

- A network is a collection of switches connected by communication channels
- network topology: pattern of interconnection of the switches
- Bus; Two-dimensional (rectangular); three-dimensional; multistage network; fat-tree, etc.

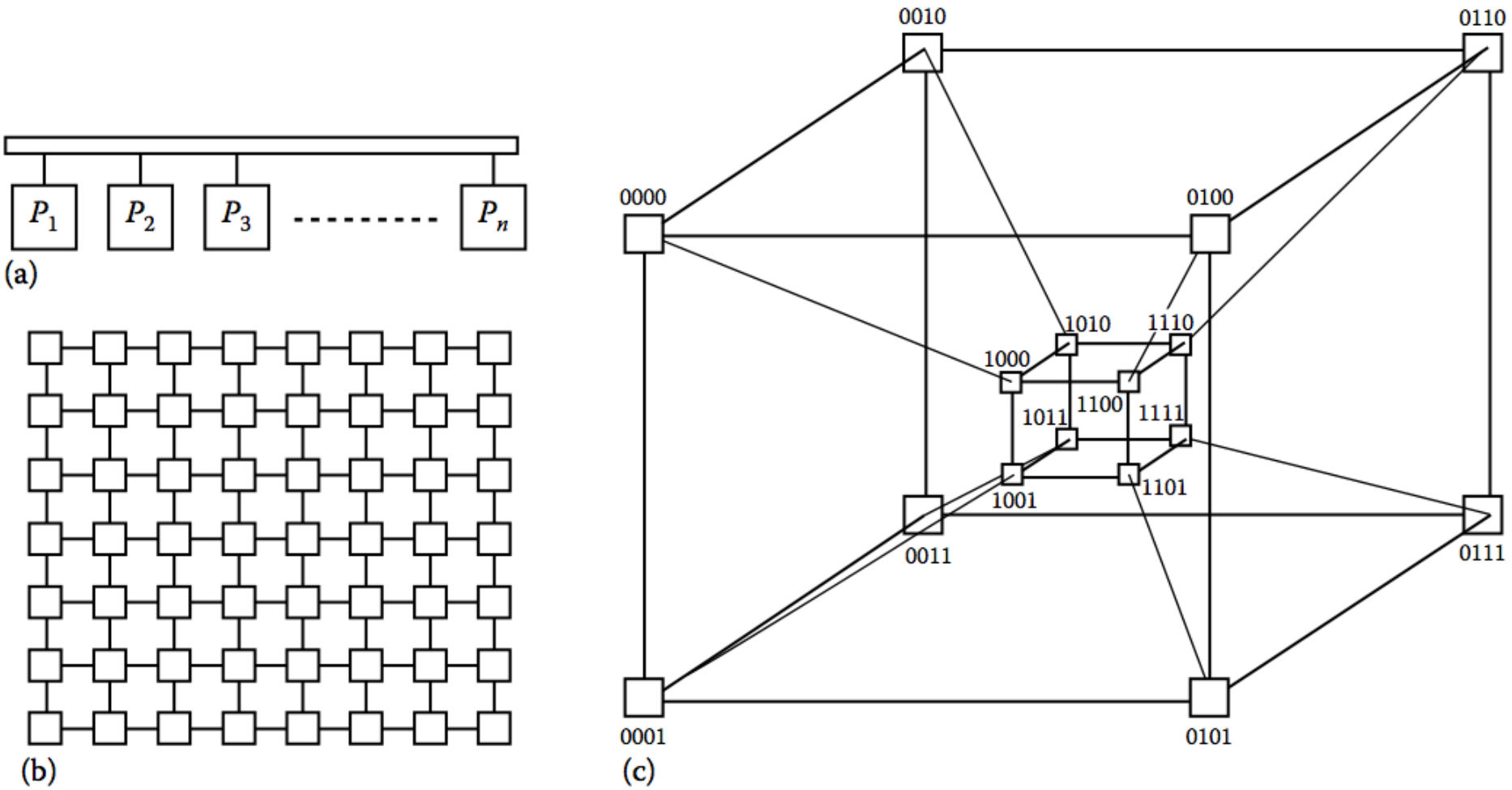
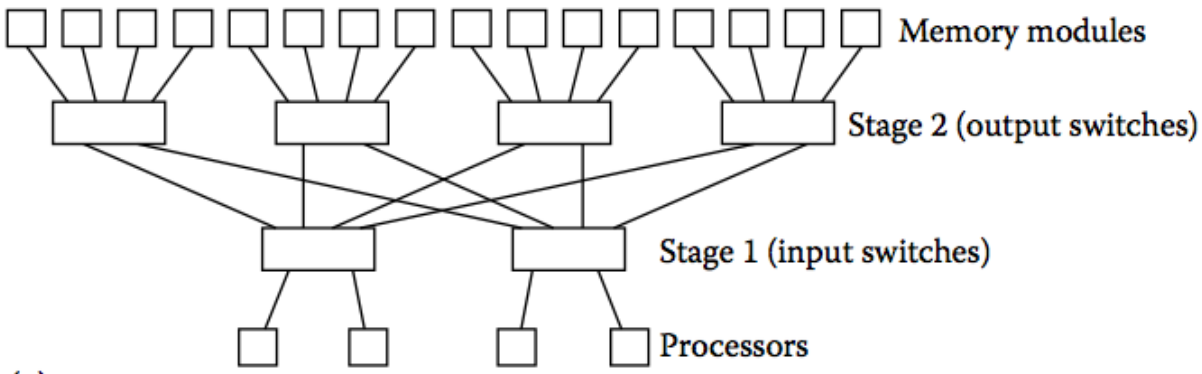


FIGURE 25.2 (a) Bus, (b) two-dimensional mesh, and (c) hypercube network topologies.



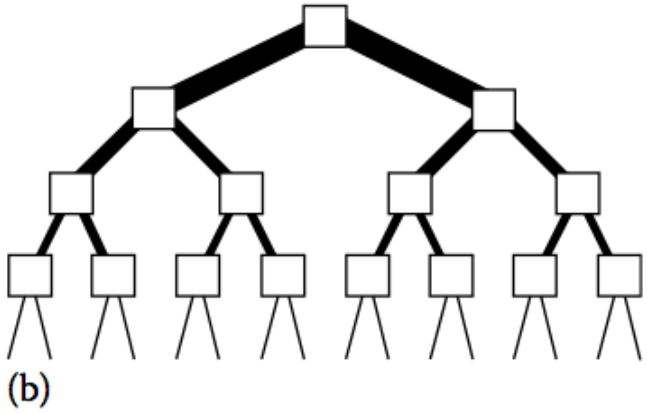


FIGURE 25.3 (a) 2-level multistage network and (b) fat-tree network topologies.

- For algorithms designed to run efficiently on particular network topologies
  - It leads to very fine-tuned algorithms, it has some disadvantages
    - May not perform well on other networks
    - Algorithms uses a particular network tend to be more complicated than algorithms designed for more abstract models

Alternative to modeling topology of a network

- To summarize its routing capabilities in two parameters
  - Latency --- of a network, time takes for message to traverse network
    - Often modeled by considering worst-case time
  - Bandwidth --- at each port, the rate at which a processor can inject data into network
    - Modeled as maximum rate at which processors can inject messages into network without causing congestion

Primitive operations

- Types of operations that processors and network are permitted to perform
- Restrictions might make an algorithm impossible to execute on a particular model, expensive
- Important to understand what instructions supported before designing a parallel algorithm

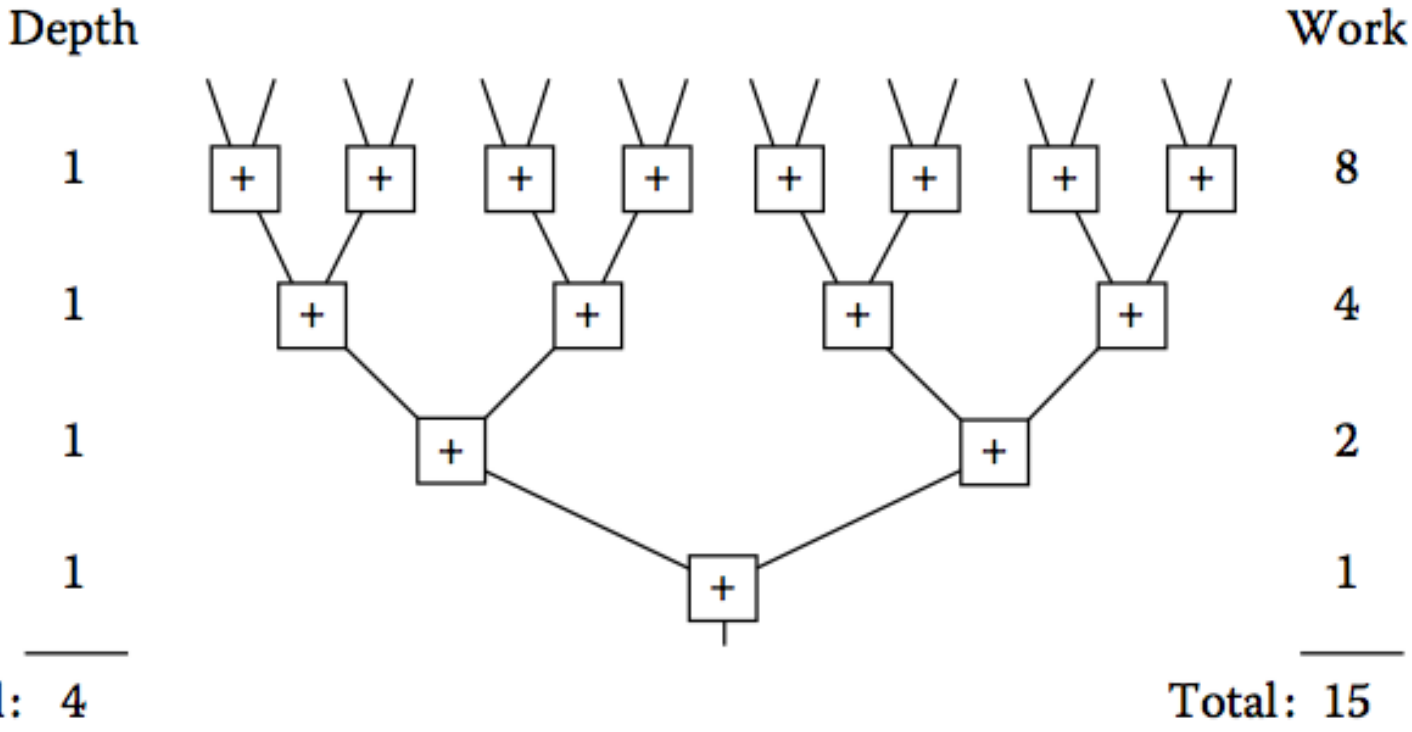
- Three classes of instructions
  - For Concurrent accesses to same shared memory location
    - Exclusive access to resource
    - Allow concurrent access to resource
  - For synchronization
  - Global operations on data

Work-Depth Models

- Alternative to focusing on machine, is to focus on the algorithm
- Work-depth model
  - Algorithm cost determined by # of operations, and dependencies among them
  - Work (W) --- total # of operations
  - Depth (D) --- longest chain of dependencies
  - Ratio P = W/D --- parallelism of the algorithm

- Algorithms efficient in work-depth models can often be translated to algorithms that efficient in multiprocessor models, and to real parallel computers
- Advantage is that no machine-dependent details to complicate design and analysis of algorithms

- Three classes of work-depth models
  - Circuit models
  - Vector machine models
  - Language-based models



16 numbers on a tree. The total depth (longest chain of dependencies) is 4 and the total work
s 15.

- For circuit models
  - Work --- total number of nodes
  - Depth --- number of nodes on longest directed path
- In vector model
  - Each step performs operation on a vector, produces a vector result
  - Work --- length of input(or output) vector, sum of work of each step
  - Depth --- number of vector steps
- In a language model
  - Cost associated with each programming language construct

Assign costs to algorithms

- Work-depth models, cost of algorithm determined by Work and depth
- These notions can be defined for multiprocessor models
  - Work W --- number of processors multiplied by time required for algorithm to complete
  - Depth D --- total time required to execute algorithm

Measure algorithm cost

- Depth of an algorithm is important
  - For those that time to perform a computation is crucial
- Most important measure is the work
  - Total cost of performing a computation roughly proportional to number of processors multiplied by amount of time
- A parallel algorithm is work-efficient
  - If asymptotically (as problem size grows) it requires at most a constant factor more work than best sequential algorithm.

Emulations among Models

- Often automatic and efficient techniques translating algorithms designed for one model into algorithms
- Translations are work-preserving
  - Work performed by both algorithms is same, to within a constant factor
- Algorithm designed for work-depth model can be translated in work-preserving fashion to a multiprocessor model
- Work-preserving translations are also between different multiprocessor models

Why choose work-depth language model

- Seems most clearly show basic ideas behind algorithms
- Model defined in terms of language constructs, and rules for assigning costs to constructs
- Uses two parallel constructs
  - Parallel apply-to-each
  - Parallel-do

- Apply-to-each construct
  - Apply expression over a sequence of values in parallel

$$\{a * a : a \in [3, -4, -9, 5]\}$$

- Parallel-do construct
  - Evaluate multiple statements in parallel
  - Keywords, in parallel do

in parallel do

$A :=$ FUNCTION1(X)

$B :=$ FUNCTION2(Y)

How work and depth assigned to the language constructs

- Apply to each construct
  - Work and depth of a scalar primitive operation is one
  - Work --- sum of work for each of individual applications of the function
  - Depth --- maximum of depths of individual applications of the function
- For parallel-do construct
  - Work --- sum of work for each of statements
  - Depth --- maximum depth of its statements

Additional built-in functions on sequences(each can be implemented in parallel)

- Distribute
  - Create a sequence of identical elements
- ++(append)
  - Appends two sequences
- Flatten
  - Converts a nested sequence to a flat sequence
- ← (write)
  - Write multiple elements into a sequence in parallel

$$[0,0,0,0,0,0,0,0] \leftarrow [(4,-2),(2,5),(5,9)]$$

Parallel Algorithmic Techniques

Divide-and-Conquer

- First splits problem into sub-problems that easier to solve, then solves sub-problems, often recursively
- Typically sub-problems can be solved independently
- Finally, merge solutions of sub-problems, to construct a solution to original problem

- Divide-and-conquer paradigm improves program modularity, often leads to simple and efficient algorithms
- Sub-problems typically independent, can be solved in parallel
- To yield a highly parallel algorithm
  - To parallelize the divide step and the merge step
  - To divide original problem into as many as sub-problems as possible

Example: mergesort algorithm

- Sequential running time specified by recurrence

$$T(n) = \begin{cases} 2T(n/2) + O(n) & n > 1 \\ O(1) & n = 1 \end{cases}$$

$$T(n) = O(n \log n).$$

- Parallel calls expressed as:

```
ALGORITHM: MERGESORT(A)
1    if (|A| = 1) then return A
2    else
3        in parallel do
4            L := MERGESORT(A[0..|A|/2))
5            R := MERGESORT(A[|A|/2..|A|))
6        return MERGE(L, R)
```

- Assume merging remains sequential
  - Work and depth to merge two sorted sequences of n/2 is O(n)

$$W(n) = 2W(n/2) + O(n)$$

$$D(n) = \max(D(n/2), D(n/2)) + O(n)$$

$$= D(n/2) + O(n)$$

$W(n) = O(n \log n)$ $D(n) = O(n)$,

parallelism of this algorithm is $O(\log n)$

parallelism of this algorithm is $O(\log n)$

- Not very much; problem is that merge step remains sequential, is the bottleneck
- Parallelism = W/D
  - The larger the better
- Depth, the smaller the better

- Uses a parallel merge[52]
  - Two sorted sequences n/2, can be merged with work O(n) and depth O(log log n)
  - Depth becomes:

$$D(n) = D(n/2) + O(\log \log n)$$

$$D(n) = O(\log n \log \log n)$$

- Uses a pipelined divide-and-conquer [26]
  - Depth be reduced to O(log n)
  - To start merge at top level before recursive calls complete

Randomization

- Sampling
  - Solving problem on that sample, then using solution for that sample to guide solution for original set
- Symmetry breaking
  - Select large independent set of vertices in a graph in parallel
  - Difficult to make simultaneously for each vertex if local structure at each vertex is same
- Load balancing
  - Randomly assign each element to a subset, to quickly partition a large number of data items into a collection of approximately evenly sized subsets

Parallel Pointer Techniques

- Traditional sequential techniques manipulating lists, trees, and graphs not translate easily into parallel techniques
  - Tree traversing, depth-first traversal
- Pointer Jumping
  - each pointer jumping step, each node in parallel replaces its pointer with that of its successor (or parent).
  - For example, one way to label each node of an n-node list (or tree) with the label of the last node (or root) is to use pointer jumping. After at most [log n] steps, every node points to the same node, the end of the list (or root of the tree).

- Euler Tour Technique
  - An Euler tour of a directed graph is a path through the graph in which every edge is traversed exactly once
  - By keeping a linked structure that represents the Euler tour of a tree it is possible to compute many functions on the tree, such as the size of each subtree [83].
  - This technique uses linear work, and parallel depth that is independent of the depth of the tree. The Euler tour technique can often be used to replace a standard traversal of a tree, such as a depth-first traversal.

- Graph Contraction
  - Graph contraction is an operation in which a graph is reduced in size while maintaining some of its original structure
  - after performing a graph contraction operation, the problem is solved recursively on the contracted graph
  - For example, one way to partition a graph into its connected components is to first contract the graph by merging some of the vertices with neighboring vertices, then find the connected components of the contracted graph
  - Many problems can be solved by contracting trees [64,65], in which case the technique is called tree contraction.

- Ear Decomposition
  - An ear decomposition of a graph is a partition of its edges into an ordered collection of paths.
  - The first path is a cycle, and the others are called ears. The end-points of each ear are anchored on previous paths.
  - Once an ear decomposition of a graph is found, it is not difficult to determine if two edges lie on a common cycle. This information can be used in algorithms for determining biconnectivity, triconnectivity, 4-connectivity, and planarity
  - An ear decomposition can be found in parallel using linear work and logarithmic depth, independent of the structure of the graph
  - this technique can be used to replace the standard sequential technique for solving these problems, depth-first search

- Other techniques
  - Finding small graph separators is useful for partitioning data among processors to reduce communication
  - Hashing is useful for load balancing and mapping addresses to memory
  - Iterative techniques are useful as a replacement for direct methods for solving linear systems