Parallelism beyond MapReduce: Distributed Data Processing

Kai Shen

Parallel Data Processing Models

- Single-machine shared memory
  - Data must fit in one machine
  - Multiple threads, each can access the entire memory space (shared memory actually makes programming easy)
  - Synchronization to protect shared data and enforce dependency
- Weaknesses:
  - Limited scalability
  - Limited I/O capacity (even if the computing capability is sufficient)

Parallel Data Processing Models

- Distributed data processing
  - Data is partitioned and distributed
  - Tasks run on different machines on different data partitions; collaboration through network communication
- Weaknesses:
  - Inconvenience for not being able to access all the data
  - Slower data communication

Programming for Distributed Data Processing

- MapReduce: simple programming (load balancing, data movement, fault tolerance is automated) but restrictive in semantics
- MPI (Message Passing Interface)
- General distributed data processing
- Make use of idle resources (where you can find them)
- Not required assignment on any beyond MapReduce

Message Passing Interface

- De facto standard programming interface for message passing-based parallel programs
  - think of threads for shared memory parallel programming
- You write a single program, multiple copies of which will run on multiple machines
  - Assumption: all processes do mostly similar things
  - Different parts distinguish through process ID
- Communications
  - Point-to-point: send/receive
  - Group communications: broadcast, gather, scatter, reduce, barrier
- It has a variety of implementations that we won't go into

MPI Send/Receive

- Matching send/receive:
  - Process x sends a message to process y
  - Process y receives a message from process x
- Nonblocking send
- Synchronous send
- Nonblocking receive
- … … … …

MPI Group Communications

- Barrier
  - All processes wait until all have arrived
- Broadcast
  - One process (root) sends a message to be received by others
- Reduce (just like reduce in MapReduce)
  - A function (MAX, SUM, …) is applied to data supplied by all processes; result is returned at one process (root)
  - Function is evaluated following process rank order
    - Reduce(R_1, R_2) = R_{12}, Reduce(R_{12}, R_3) = R_{123}, …
    - Can be optimized if associative and/or commutative
- … … … …

MPI Applications

- Word counting
  - Divide the documents into partitions
  - Each MPI task counts words in its own partition
  - Reduce at the end
- K-means
  - Divide the samples into subsets
  - In each iteration, an MPI task assigns samples in its partition
  - Barrier between iterations (re-computation of cluster centers is a bit tricky)

MPI Applications

- PageRank / matrix-vector multiplication
  - Divide the matrix into blocks/rows, all nodes have a copy of the vector
  - Each MPI task computes matrix-vector multiplication for its own data
  - All-to-all broadcast between iterations, new pagerank vector is distributed to all
- Gaussian Elimination
  - Divide the matrix into blocks/rows
  - … can be done, but somewhat complex

MPI vs. MapReduce

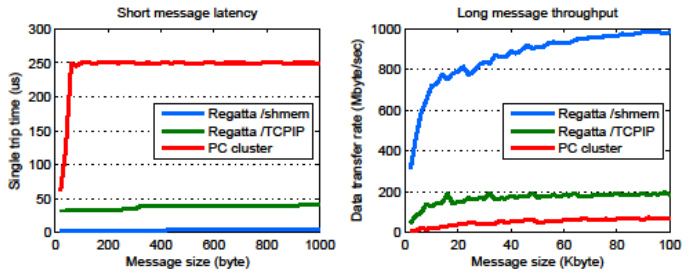
- Ease of programming
  - complexity of interface specification
- Automatic system support
  - for load balancing, data movement, and fault tolerance
- Flexibility
  - in supporting complex application semantics
  - in custom data distribution and transfer
- MPI is still restrictive
  - in communication modes
  - in custom performance optimization

General Distributed Data Processing

- As usual, we first divide the data into partitions
- Individual per-partition data processing runs on each machine
- Tasks communicate through sockets (TCP/IP)
  - Send/receive
  - Asynchronous Send/receive
- Implement everything else on top of the above
  - MPI synchronous send?
  - Reduce, data aggregation?
  - … …
- Most flexible, and most efficient (if done right), but requiring most programming work

Communication Performance

- Direct data accesses on a shared memory machine
- TCP/IP on a shared memory machine
- TCP/IP over a cluster



Communication Performance

- Long communication latency (vs. high bandwidth)
  - our Ethernet cluster: 250us latency, 80MB/sec bandwidth
    if 1KB per synchronization, effective bandwidth is 4MB/sec
  - ⇨ synchronize/wait as few times as possible
- Performance issues with TCP/IP
  - Connection establishment
  - Congestion control
  - ⇨ UDP or raw IP with some error detection management

Custom Fast Communications

- Fast local area network (Myrinet, infiniband, …)
  - No need to support Internet communications (TCP/IP)
- Large multiprocessors from Cray, IBM, …
  - Each processor (or processor group) has some local memory
  - Fast access to remote memory through fast system bus

Load Balancing

- With most flexibility, you also must take care of all performance optimization and fault/anomaly management
- Dynamic load balancing
  - Implement master (in MapReduce) or TaskTracker (in Hadoop)
  - Maintain more tasks than machines; assign tasks to machines
    - Reactive assignment: assign one more task to a machine that just informed me it has completed its current assignment
    - Proactive assignment: poll the load situation at machines and assign more to those with low load (don't have to poll all frequently)
  - Observe data locality as much as possible

Parallel Data Aggregation

- How to implement reduce()?
- All data sent to the one node; reduced at that node
- Tree-ordered parallel reduction (if reduction op is associative)
- Adaptive order based on progress at each node (if reduction op is commutative)

Performance Outliers

- Performance of your application is bounded by the slowest task
- Many reasons for a particularly slow task (even if load appears to be balanced):
  - awful data locality
  - long network switch distance
  - OS daemons run at unfortunate time
  - disk/SSD remapped data layouts (due to wear) hinder I/O speed
  - …
- Monitor the progress of tasks, and re-launch a redundant task (at a different machine) if necessary

Deadlocks

- Why does my distributed program get stuck?
  - The MapReduce system support typically ensures progress is always made
- Possible reasons
  - Receive without a matching send (or a matching send cannot be reached)
  - Group communications are not called by all in the group
  - Send blocked by insufficient buffer space
  - … … … …
- Debugging
  - Find out where each process is blocked at
  - How does it conflict with design? What's wrong with implementation?

Fault Tolerance

- Checkpointing and restart
  - Checkpointed state (a distributed state) is a consistent state
  - A consistent state is one that can be reached (after freezing the execution of all nodes at once) in some real execution
- Easy to do for MapReduce
  - Since each map or reduce task does not send data or interact with others til completion, wiping out the partial work of one task still reaches a consistent state (as if the task execution has been extremely slow, actually made no progress since start)
- Challenging for general distributed applications
  - If A is checkpointed before sending out a message (dest. B), then B should be checkpointed before receiving the message

Utilizing Idle Resources

- Motivation: lots of machines are mostly idle in a University lab or across the Internet
- We distribute work to those machines (screen saver download) and have them done when the machine is idle
- SETI@home
  - Distributed task: a data partition to process
- Brute-force password cracking
  - Distributed task: passwords to try
- Realization:
  - Data/work must be easily partitionable without interdependencies
  - Must tolerate potentially long network delays
  - Must deal with unpredictable response time of tasks
  - Must be un-annoying
- Is it worth doing?