CS 6240: Parallel Data Processing in MapReduce: Module 2, Lecture 1

Mirek Riedewald

Consider a few simple examples to get a feeling for obvious and more subtle challenges when parallelizing an algorithm.

Sum Of Integers

- Compute sum of a large set of integers
- Sequential: simple for-loop (scan)
- Parallel: assign data chunk to each processor to compute local sum, then add them together
- Algorithmically easy, but…
  - Transmitting a chunk to another machine might take longer than locally computing the sum
    - No problem if data is distributed already
  - Many-core: what if data transfer to the cores is the bottleneck of the computation?

Word Count

- Number of occurrences of each word in a large document collection
- Sequential: for each document, update counter for each word found
  - Use single data structure, e.g., hash map, to keep track of counts
- Parallel program 1: each processor does this for a subset of documents using a local "copy" of the data structure
  - Good: perfectly parallel counting if each processor already has its own data chunk
  - Bad: many "copies" of the hash maps, which need to be moved around for final aggregation step
- Parallel program 2: use single shared data structure for counts
  - Good: no "replication", no need for final aggregation step
  - Bad: need to coordinate access to shared data structure (e.g., locking), not a good fit for shared-nothing architecture

Before exploring parallel algorithms in more depth, how do we know if our parallel algorithm or implementation actually does well or not?

Performance Metrics

- Total execution time
- Total resources consumed
- Total amount of money paid
- Total energy consumed

- Optimize some combination of the above
  - E.g., minimize total execution time, subject to a money budget constraint

Classic Measures Of Success For Parallelization

- If sequential version takes time t, then parallel version on n processors should take time t/n
  - Speedup = sequentialTime / parallelTime
  - Note: job, i.e., work to be done, is fixed
- Response time should stay constant if number of processors increases at same rate as “amount of work”
  - Scaleup = workDoneParallel / workDoneSequential
  - Note: time to work on job is fixed

Scalability Through Load Balancing

- Avoid overloading one processor while another is idle
  - Careful: if better balancing increases total load, it might not be worth it
  - Careful: optimizes for response time, but not necessarily other metrics like $ paid
- Static load balancing
  - Need cost analyzer like in DBMS
- Dynamic load balancing
  - Easy: Web search
  - Hard: join

Amdahl's Law

- Consider job taking sequential time 1 and consisting of two sequential tasks taking time t_1 and $1-t_1$, respectively
- Assume we can perfectly parallelize the first task on n processors
  - Parallel time: $t_1/n + (1 – t_1)$
- Speedup = $1 / (1 – t_1(n-1)/n)$
  - $t_1=0.9$, $n=2$: speedup = 1.81
  - $t_1=0.9$, $n=10$: speedup = 5.3
  - $t_1=0.9$, $n=100$: speedup = 9.2
  - Max. possible speedup for $t_1=0.9$ is $1/(1-0.9) = 10$

Implications of Amdahl’s Law

- Parallelize the tasks that take the longest
- Sequential steps limit maximum possible speedup
  – Communication between tasks, e.g., to transmit intermediate results, can inherently limit speedup, no matter how well the tasks themselves can be parallelized
- If fraction x of the job is inherently sequential, speedup can never exceed 1/x
  – No point running this on too many processors

Course Content in a Nutshell

- In big-data processing, usually the same computation needs to be applied to a lot of data.
- We want to divide the work between multiple processors.
- When dividing work, we often need to combine intermediate results from multiple processors.
- We want an environment that simplifies writing such programs and executing them on many processors.

Why This Is Not So Easy

- How can the work be partitioned without communicating too much intermediate data?
- How do we start up and manage 1000s of tasks for a job?
- How do we get large data sets to processors or move processing to the data?
- How do we deal with slow responses and failures?

Technical Problems

- Shared resources limit scalability due to the cost of managing concurrent access, e.g., through locking.
- Shared-nothing architectures still need communication for processes to share data and coordinate with each other.
- Whenever multiple concurrent processes interact, there is a potential for deadlocks and race conditions.
- It is difficult to reason about the behavior and correctness of concurrent processes, especially when failures are part of the model.
- There is an inherent tradeoff between consistency, availability, and partition tolerance. We will discuss this in a future unit.

What Can We Do?

- As a programmer, work at the right level of abstraction.
  - If the approach is too low-level, it becomes difficult to write programs.
    - Manage locks on shared data structures and manage communication between machines in the application code; handle failures
  - If the approach is too high-level, it could suffer from poor performance if control for crucial bottleneck is "abstracted away".
- Possible solution: declarative style of programming
  - Specify WHAT needs to be computed, not HOW this is done
  - Success story: SQL for relational databases
    - SQL query specifies what the user is looking for
    - Database optimizer automatically chooses an efficient implementation (More on this in a future unit.)

The MapReduce Way

- Use hardware that can scale out, not just up.
  - MapReduce was initially designed for WSCs. Doubling the number of commodity servers in a cluster is easy, but buying a double-sized SMP machine is not.
- Have the data located near the processors.
  - For Big Data, moving too much data around tends to result in poor performance. MapReduce therefore tries to assign tasks to machines that already have the data.
- Avoid centralized resources that are likely bottlenecks.
- Read and write data sequentially in large chunks to amortize latency.