Parallel Nested Loops

• For each tuple $s_i$ in $S$
  – For each tuple $t_j$ in $T$
    • If $s_i = t_j$, then add $(s_i, t_j)$ to output
• Create partitions $S_1$, $S_2$, $T_1$, and $T_2$
• Have processors work on $(S_1, T_1)$, $(S_1, T_2)$, $(S_2, T_1)$, and $(S_2, T_2)$
  – Can build appropriate local index on chunk if desired
• Nice and easy, but…
  – How to choose chunk sizes for given $S$, $T$, and #processors?
  – There is data duplication, possibly a lot of it
    • Especially undesirable for highly selective joins with small result

Parallel Partition-Based

• Create $n$ partitions of $S$ by hashing each $S$-tuple $s$, e.g., to bucket number $(s \mod n)$
• Create $n$ partitions of $T$ in the same way
• Run join algorithm on each pair of corresponding partitions

• Can create partitions of $S$ and $T$ in parallel
• Choose $n$ = number of processors
• Each processor locally can choose favorite join algorithm
• No data replication, but…
  – Does not work well for skewed data
  – Limited parallelism if range of values is small
More Join Thoughts

• What about non-equi join?
  – Find pairs \((s_i, t_j)\) that satisfy a predicate like inequality, band, or similarity (e.g., when \(s\) and \(t\) are documents)
• Hash-partitioning will not work any more
• Now things are becoming really tricky...
• We will discuss these issues in a future lecture.

Median

• Find the median of a set of integers
• Holistic aggregate function
  – Chunk assigned to a processor might contain mostly smaller or mostly larger values, and the processor does not know this without communicating extensively with the others
• Parallel implementation might not do much better than sequential one
• Efficient approximation algorithms exist
Parallel Office Tools

• Parallelize Word, Excel, email client?
• Impossible without rewriting them as multi-threaded applications
  – Seem to naturally have low degree of parallelism
• Leverage economies of scale: $n$ processors (or cores) support $n$ desktop users by hosting the service in the Cloud
  – E.g., Google docs

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

• 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

Things to Consider: 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 = \[ 1 / (1 - t_1(n-1)/n) \]
  – \( t_1=0.9, n=2: \text{speedup} = 1.81 \)
  – \( t_1=0.9, n=10: \text{speedup} = 5.3 \)
  – \( t_1=0.9, n=100: \text{speedup} = 9.2 \)
  – 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 an excessive number of processors

Performance Metrics

• Total execution time
  – Part of both speedup and scaleup
• Total resources (maybe only of type X) 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
Popular Strategies

- Load balancing
  - Avoid overloading one processor while other 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

Let’s see how MapReduce works.
MapReduce

• Proposed by Google in research paper
• MapReduce implementations like Hadoop differ in details, but main principles are the same

Overview

• MapReduce = programming model and associated implementation for processing large data sets
• Programmer essentially just specifies two (sequential) functions: map and reduce
• Program execution is automatically parallelized on large clusters of commodity PCs
  – MapReduce could be implemented on different architectures, but Google proposed it for clusters
Overview

• Clever abstraction that is a good fit for many real-world problems
• Programmer focuses on algorithm itself
• Runtime system takes care of all messy details
  – Partitioning of input data
  – Scheduling program execution
  – Handling machine failures
  – Managing inter-machine communication

Programming Model

• Transforms set of input key-value pairs to set of output values (notice small modification compared to paper)
• Map: \((k1, v1) \rightarrow \text{list } (k2, v2)\)
• MapReduce library groups all intermediate pairs with same key together
• Reduce: \((k2, \text{list } (v2)) \rightarrow \text{list } (k3, v3)\)
  – Usually zero or one output value per group
  – Intermediate values supplied via iterator (to handle lists that do not fit in memory)
Example: Word Count

- Insight: can count each document in parallel, then aggregate counts
- Final aggregation has to happen in Reduce
  - Need count per word, hence use word itself as intermediate key (k2)
  - Intermediate counts are the intermediate values (v2)
- Parallel counting can happen in Map
  - For each document, output set of pairs, each being a word in the document and its frequency of occurrence in the document
  - Alternative: output (word, “1”) for each word encountered

Word Count in MapReduce

Count number of occurrences of each word in a document collection:

```java
map(String key, String value):
// key: document name
// value: document contents
for each word w in value:
    EmitIntermediate(w, "1");
```

```java
reduce(String key, Iterator values):
// key: a word
// values: a list of counts
int result = 0;
for each v in values:
    result += ParseInt(v);
Emit(AsString(result));
```

Almost all the coding needed
(need also MapReduce specification object with names of input and output files, and optional tuning parameters)
Execution Overview

• Data is stored in files
  – Files are partitioned into smaller splits, typically 64MB
  – Splits are stored (usually also replicated) on different cluster machines

• **Master** node controls program execution and keeps track of progress
  – Does not participate in data processing

• Some workers will execute the Map function, let’s call them **mappers**
• Some workers will execute the Reduce function, let’s call them **reducers**

Execution Overview

• Master assigns map and reduce tasks to workers, taking data location into account
• Mapper reads an assigned file split and writes intermediate key-value pairs to local disk
• Mapper informs master about result locations, who in turn informs the reducers
• Reducers pull data from appropriate mapper disk location
• After map phase is completed, reducers sort their data by key
• For each key, Reduce function is executed and output is appended to final output file
• When all reduce tasks are completed, master wakes up user program
Master Data Structures

• Master keeps track of status of each map and reduce task and who is working on it
  – Idle, in-progress, or completed
• Master stores location and size of output of each completed map task
  – Pushes information incrementally to workers with in-progress reduce tasks
**Example: Equi-Join**

- Given two data sets $S=(s_1, s_2, \ldots)$ and $T=(t_1, t_2, \ldots)$ of integers, find all pairs $(s_i, t_j)$ where $s_i.A = t_j.A$
- Can only combine the $s_i$ and $t_j$ in Reduce
  - To ensure that the right tuples end up in the same Reduce invocation, use join attribute $A$ as intermediate key ($k_2$)
  - Intermediate value is actual tuple to be joined
- Map needs to output $(s.A, s)$ for each $S$-tuple $s$ (similar for $T$-tuples)

**Equi-Join in MapReduce**

- Join condition: $S.A = T.A$
- $Map(s) = (s.A, s)$; $Map(t) = (t.A, t)$
- Reduce computes Cartesian product of set of $S$-tuples and set of $T$-tuples with same key

![Diagram of DFS nodes, Mappers, Reducers, and DFS nodes connected by arrows representing data flow.](image)
Comments

- Programming model might appear very limited
- But, map and reduce can do anything with their input
  - Could implement a Turing machine inside...
  - ...which could compute anything, but...
  - ...would not result in a good parallel implementation.
- Challenge: find **best** MapReduce implementation for a given problem

Basic MapReduce Program Design

- Tasks that can be performed independently on a data object, large number of them: **Map**
- Tasks that require combining of multiple data objects: **Reduce**
- Sometimes it is easier to start program design with Map, sometimes with Reduce
- Select keys and values such that the right objects end up together in the same Reduce invocation
- Might have to partition a complex task into multiple MapReduce sub-tasks