Parallel Data-Processing Basics

Mirek Riedewald

Key Learning Goals

- What are speedup and scaleup, and how do we measure them for a given application?
  - What is the theoretically best speedup possible?
- Is load balance a measure of good parallelization?
- Will adding more processors always speed up computation? Why or why not?
- What does Amdahl's Law say? Give an example to illustrate it.

Key Learning Goals

- Why is distributed programming with shared data structures generally not considered *scalable*?
- Show an example of a “concurrency bug” where concurrent access to a shared data structure causes inconsistency.
- Why is it beneficial to have data already pre-partitioned across many machines *before* the computation starts?

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

Sum Of Integers

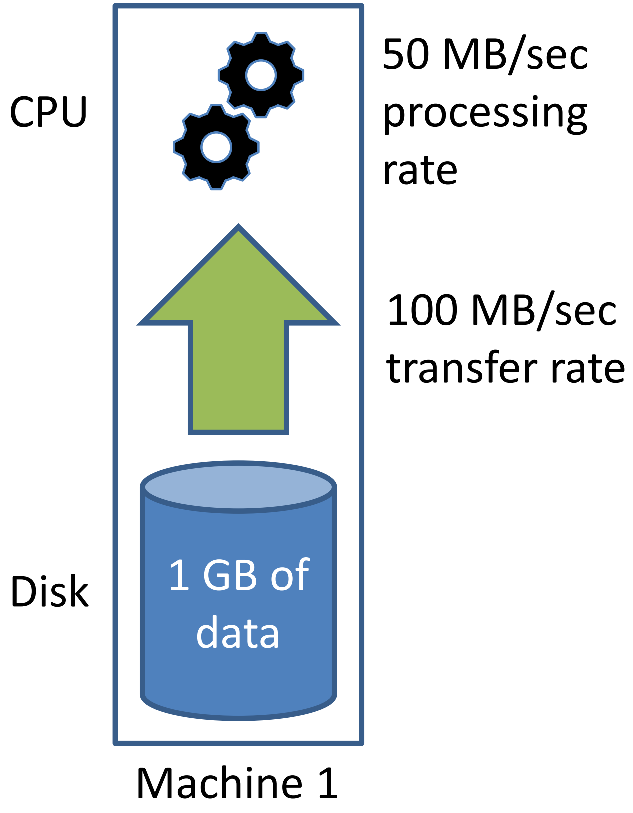
- Consider a problem that is conceptually easy to parallelize: Given a large file of integers, compute their sum. The sequential program is straightforward:
  - Open the file for reading, initialize SUM = 0.
  - While there are more numbers in the file, read the next number and add it to SUM.
  - Return SUM.
- Parallel computation is also trivial: Assign a data chunk to each processor to compute a local sum, then add up all intermediate results.
- While algorithmically simple, this parallel program might take longer to finish than the sequential version. Why?

Sum Of Integers

- Consider a scenario with 2 machines M1 and M2, where the data is stored on M1. Assume M1's CPU processes data at a rate of 50 MB/sec, while the disk can only transfer 25 MB/sec. Then M1 alone can process 1 GB of data in 40 sec.
- The disk is the bottleneck, i.e., it is 100% busy during the entire time, while the CPU is only 50% utilized. Transferring data to another machine M2 requires splitting M1's disk transfer capacity between the two machines, e.g., each CPU receives 0.5 GB of data at 12.5 MB/sec from M1's disk. Then each machine still takes 0.5 GB/12.5 MB/sec = 40 sec to receive the data, while both CPUs are only 25% utilized.
- If we take into account that communication also adds latency, and that an additional summation step is needed to add up the values from M1 and M2, then total time will exceed 40 sec.
- What if the disk was faster? Let's take a look.

Sum of Integers Demo

Let's look at the computation with one machine. Assume the CPU is slow compared to the disk.



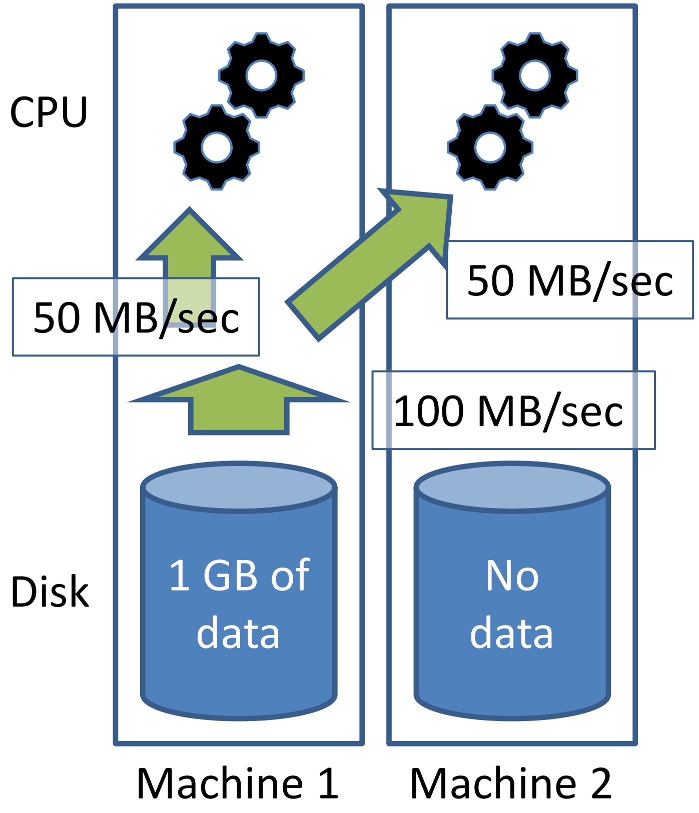
Time to sum up all numbers:
20 sec

Time to transfer all data to CPU:
10 sec

Time to complete the job:
20 sec

Sum of Integers Demo

Let's try to speed this up by using two machines. To balance the load, we send half of the data to CPU 2. As we hoped, the job completes in half the time.



2*50 MB/sec processing rate

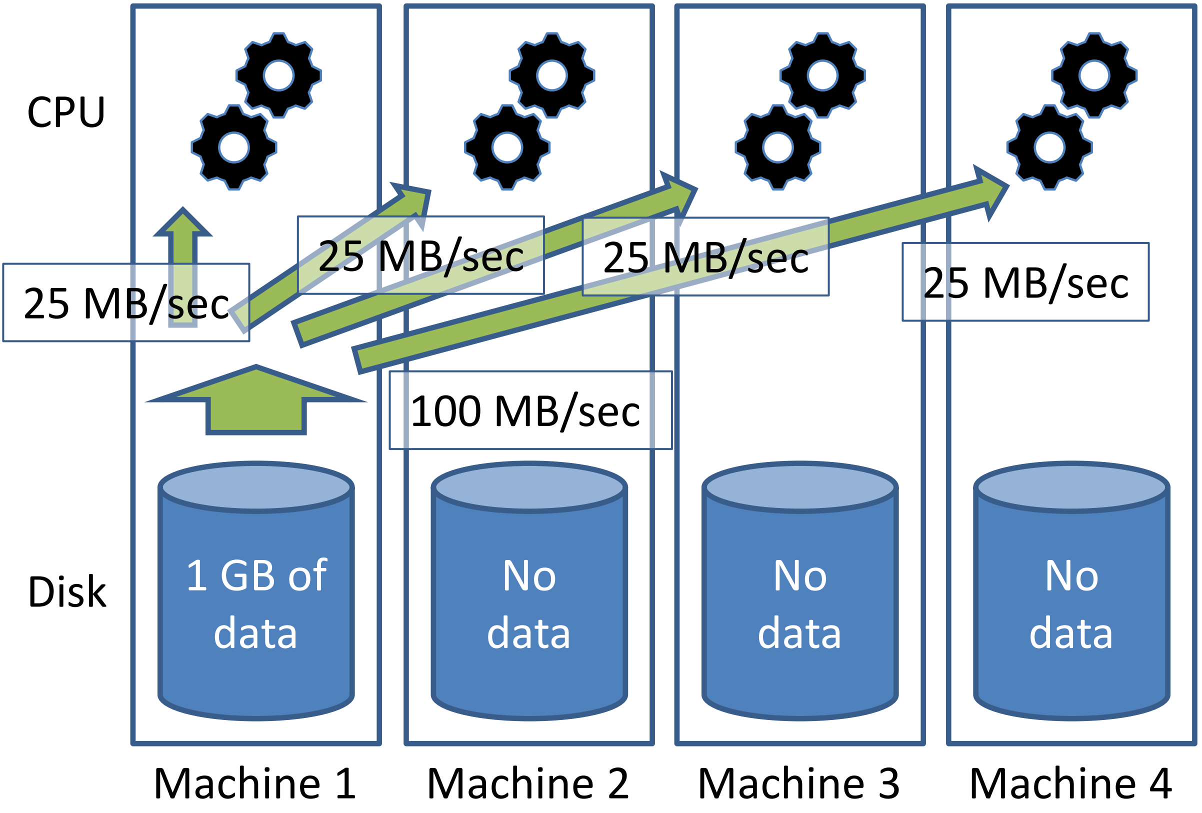
Time to sum up all numbers: **10** sec

Time to transfer all data to CPU: 10 sec

Time to complete the job: **10** sec

Sum of Integers Demo

However, as we are adding more machines, processing time does not improve any more. Even though all CPUs together could process the data in 5 sec, they must wait for the disk. As the disk became the bottleneck, adding more CPU capacity did not reduce job completion time.



4*50 MB/sec processing rate

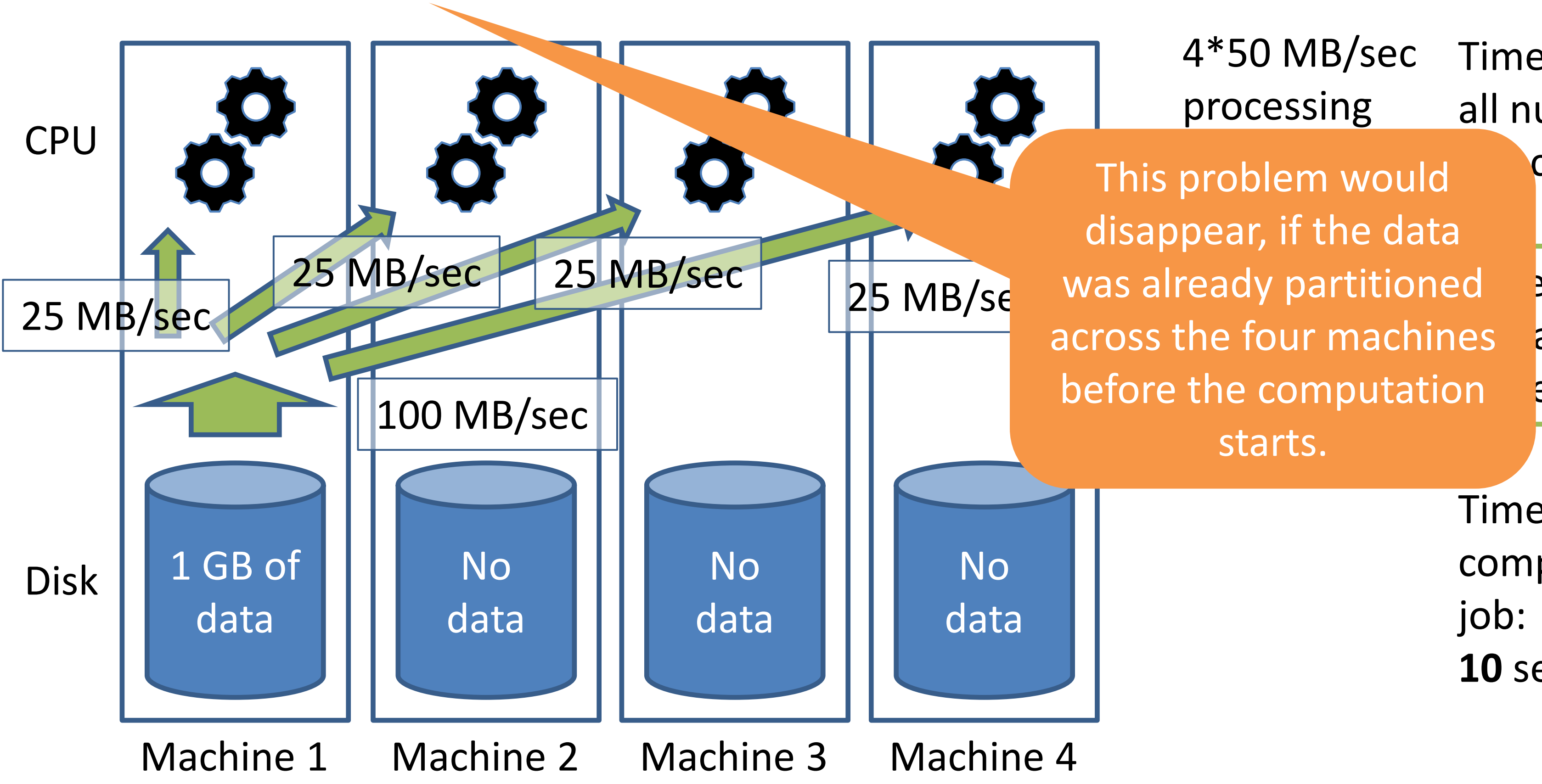
Time to sum up all numbers: 5 sec

Time to transfer all data to CPU: 10 sec

Time to complete the job: **10** sec

Sum of Integers Demo

However, as we are adding more machines, processing time does not improve any more. Even though all CPUs together could process the data in 5 sec, they must wait for the disk. As the disk became the bottleneck, adding more CPU capacity did not reduce job completion time.


CPU
Disk
25 MB/sec
25 MB/sec
25 MB/sec
25 MB/se
100 MB/sec
1 GB of data
No data
No data
No data
Machine 1
Machine 2
Machine 3
Machine 4
This problem would disappear, if the data was already partitioned across the four machines before the computation starts.

4*50 MB/sec processing

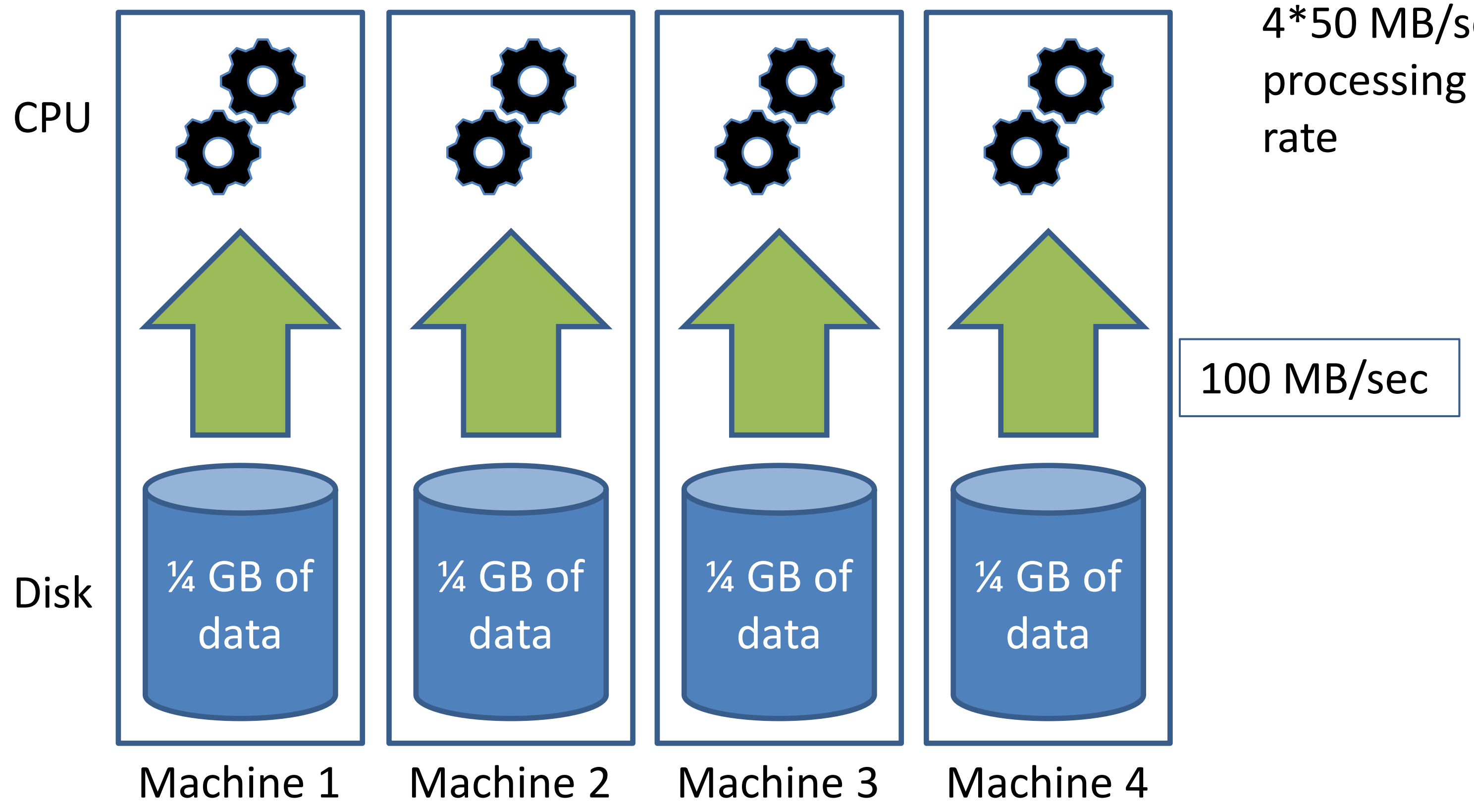
Time to sum up all numbers:
c

e to transfer
ata to CPU:
ec

Time to complete the job:
10 sec

Sum of Integers with Pre-Partitioned Data

If each disk contains ¼ of the input, then each CPU can read locally at full speed. Now the slow CPU determines the processing rate, like in the single-machine scenario. However, each machine has only a fraction of the work. This is the ideal case, assuming the final aggregation step is fast.



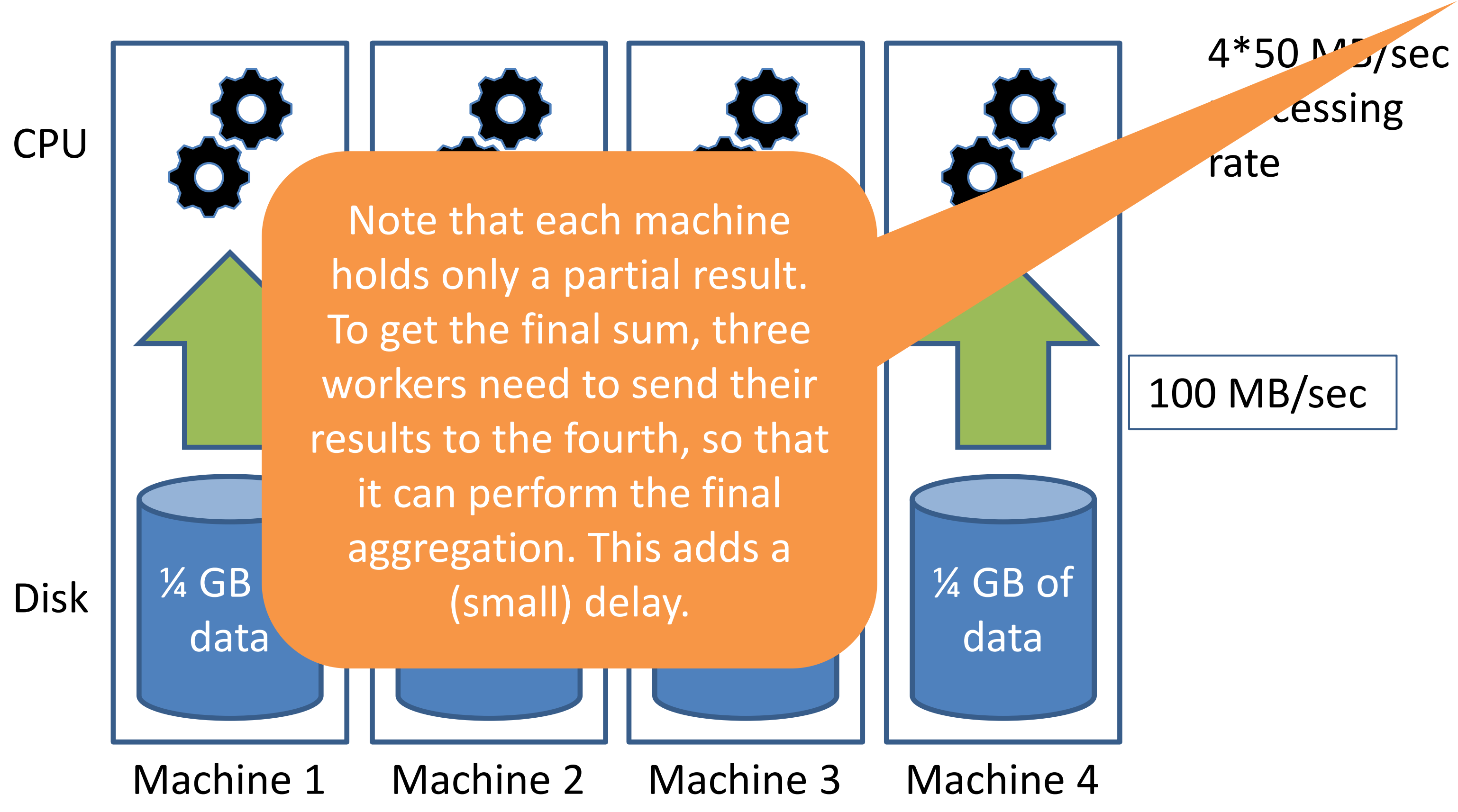
Time to sum up all numbers: **5** sec

Time to transfer all data to CPU: **2.5** sec

Time to complete the job: **5** sec

Sum of Integers with Pre-Partitioned Data

If each disk contains ¼ of the input, then each CPU can read locally at full speed. Now the slow CPU determines the processing rate, like in the single-machine scenario. However, each machine has only a fraction of the work. This is the ideal case, assuming the final aggregation step is fast.



CPU
Disk
¼ GB data
¼ GB of data
Machine 1
Machine 2
Machine 3
Machine 4
4*50 MB/sec processing rate
100 MB/sec
Note that each machine holds only a partial result. To get the final sum, three workers need to send their results to the fourth, so that it can perform the final aggregation. This adds a (small) delay.

Time to sum up all numbers: **5** sec

Time to transfer all data to CPU: **2.5** sec

Time to complete the job: **5** sec

Lessons Learned

- Adding more compute power does not speed up computation when data transfer to the processors is the bottleneck.
  - How can we see the issue in practice? During program execution, the CPUs have low utilization, because they wait for data.
- How can the problem be addressed?
  - Store data partitions on the machines that will perform the computation. The distributed file system supports this transparently.
  - If the data is transferred on-the-fly from a separate storage environment, e.g., Amazon S3, to a compute environment, e.g., Amazon EC2, then more bandwidth between storage and compute nodes should be provided for bigger data and larger compute clusters. Cloud providers usually do this automatically.

Can we avoid the problem by using a shared memory approach?

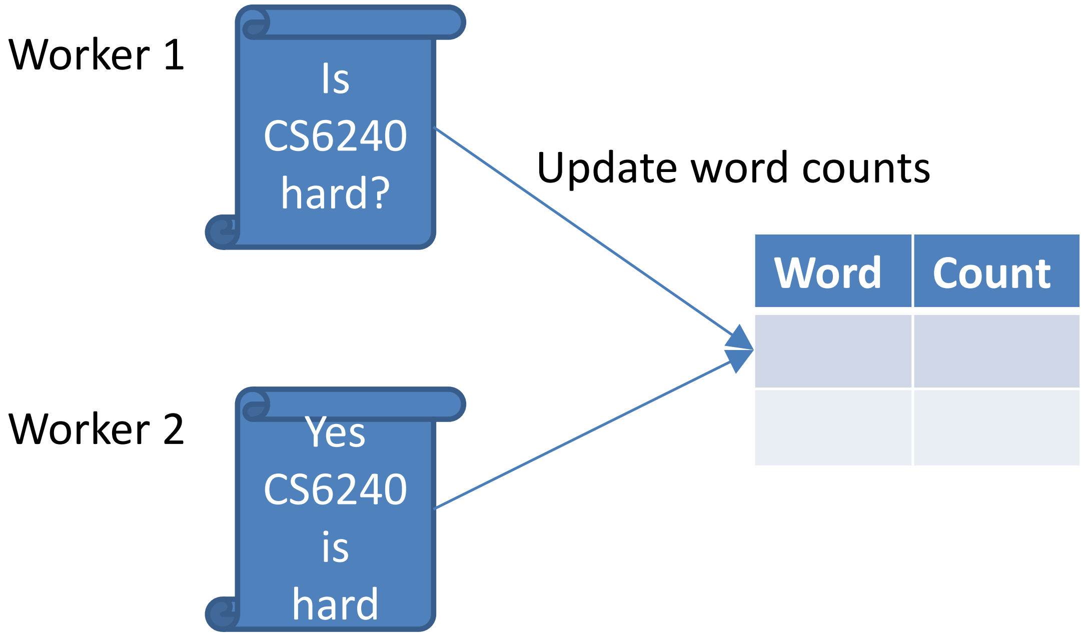
Unfortunately, that has other downsides.

Word Count

- Word Count takes the SUM example a step further and highlights the problem of shared data structures. We are given a large collection of text documents. For each word in the collection, determine how many times it occurs in total.
- Sequential program
  - For each document, update a counter for each word found in the document.
  - Note: We use a single data structure, e.g., a hash map, to keep track of the count for each word.
- We can parallelize the counting process easily, but instead of a single SUM, we need to keep track of a count for each word.

Word Count: Shared-Memory Version

- There is a single shared data structure—a hash map that maps a word to its count—that can be updated by all workers. As each worker processes a document, it attempts to update the data structure.



Worker 1

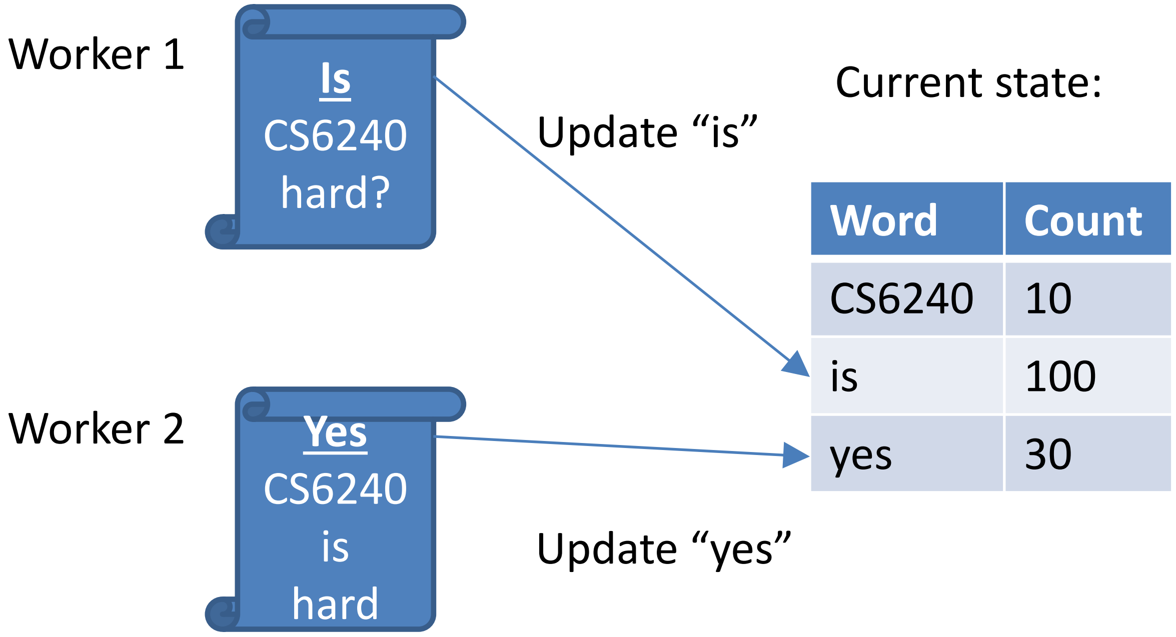
Is
CS6240
hard?

Worker 2

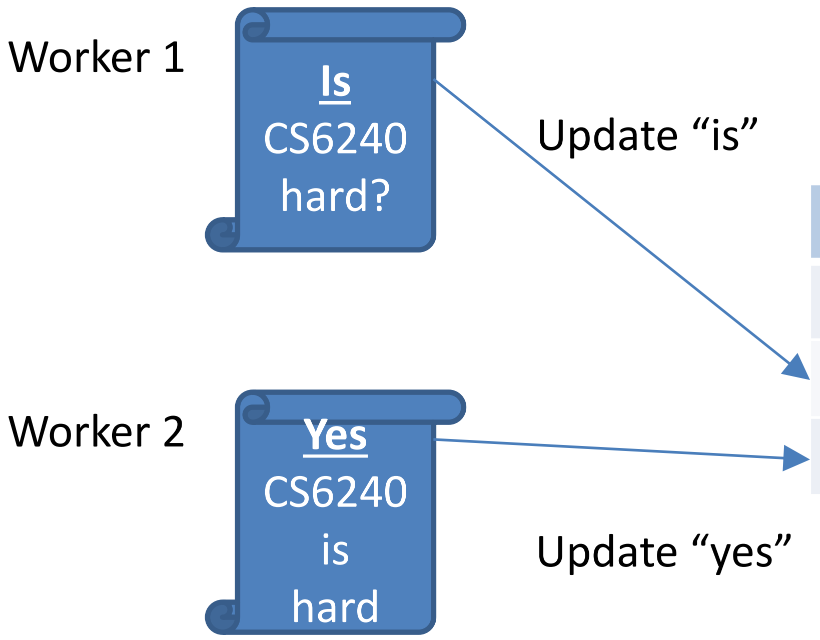
Yes
CS6240
is
hard

Current state (before the 2 new documents are being processed):

Word	Count
CS6240	10
is	100
yes	30



Word	Count
CS6240	10
is	100
yes	30

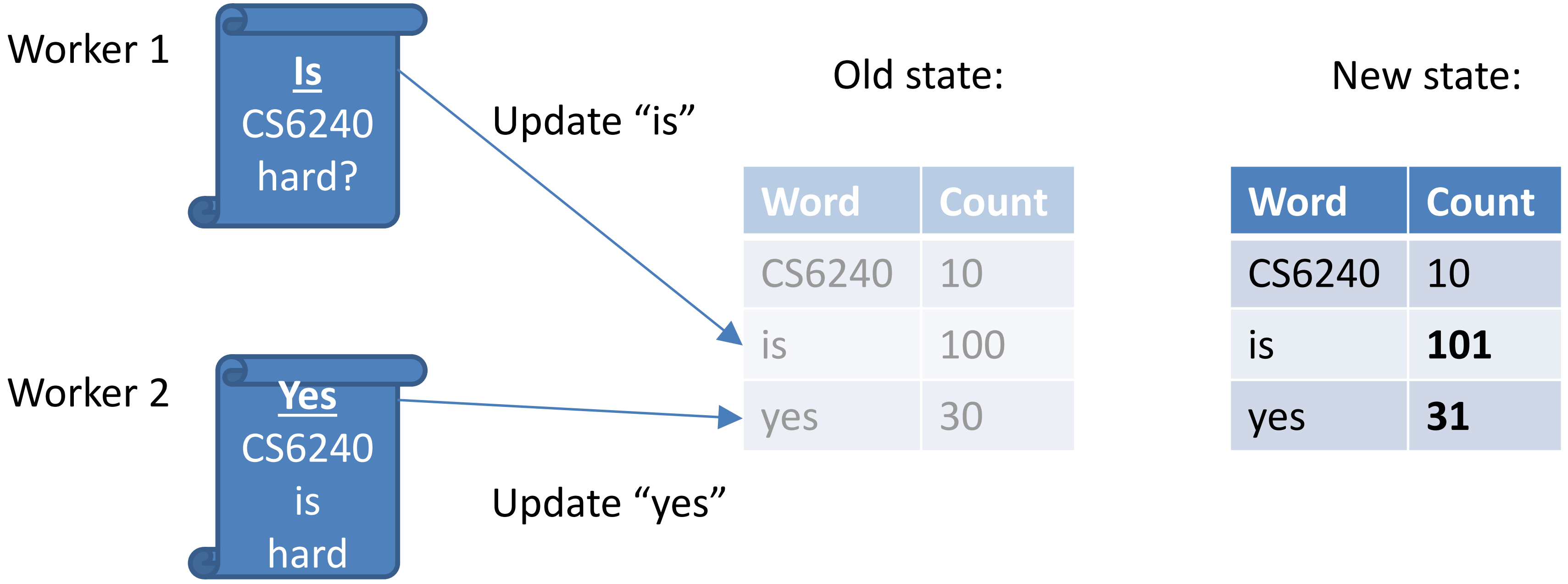


Old state:

Word	Count
CS6240	10
is	100
yes	30

New state:

Word	Count
CS6240	10
is	**101**
yes	**31**

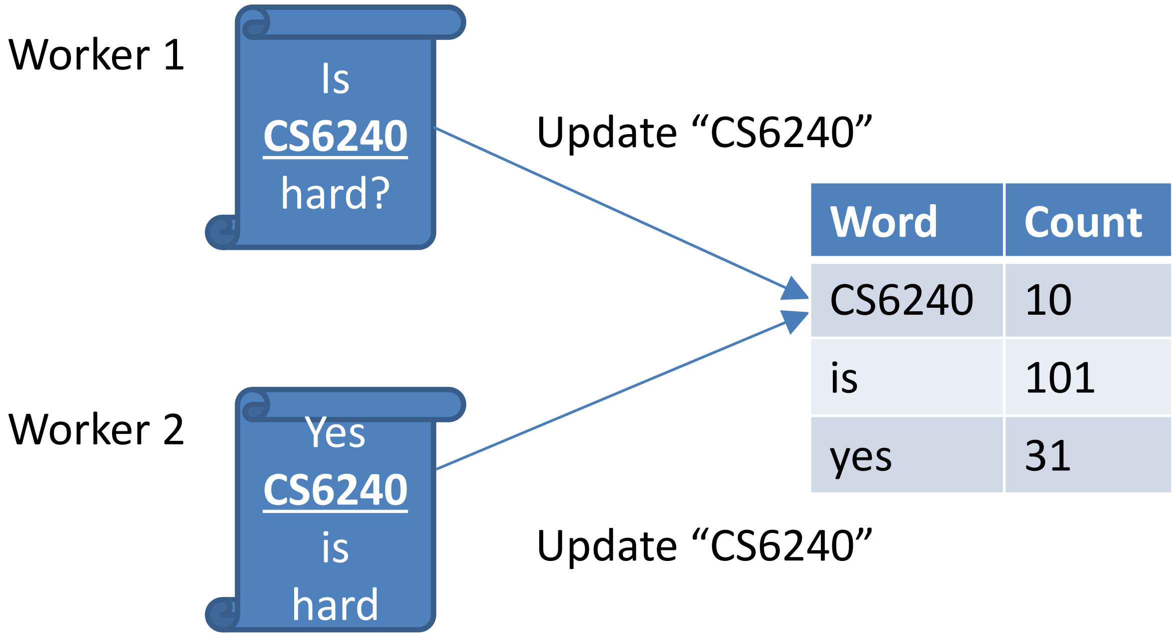


Old state:

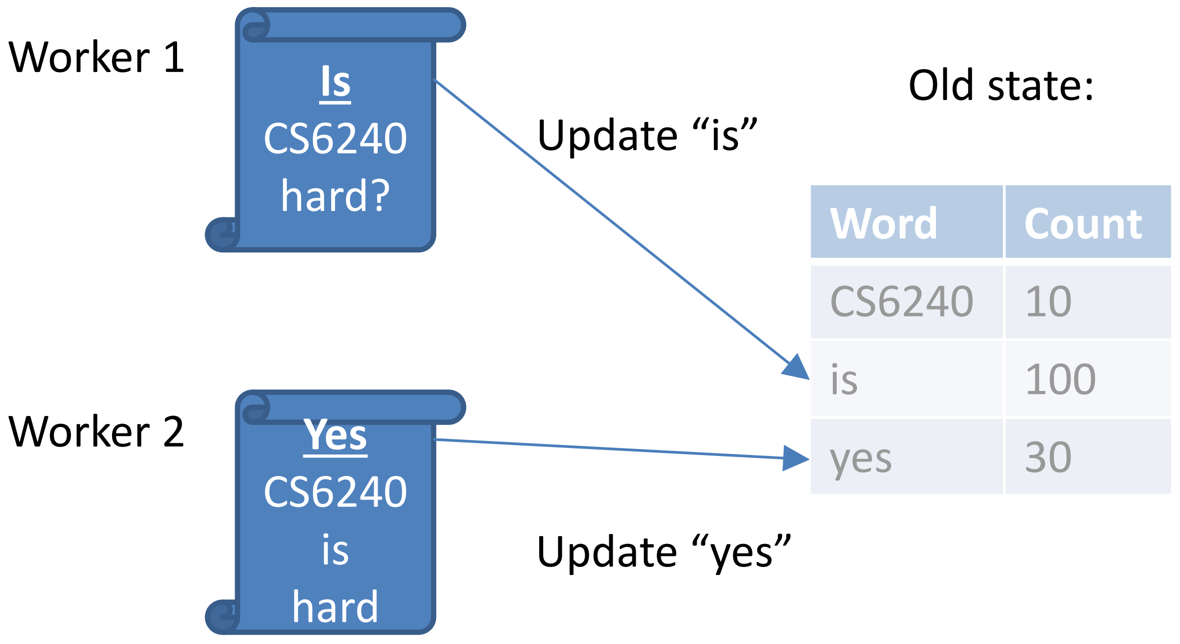
Word	Count
CS6240	10
is	100
yes	30

New state:

Word	Count
CS6240	10
is	**101**
yes	**31**



Word	Count
CS6240	10
is	101
yes	31

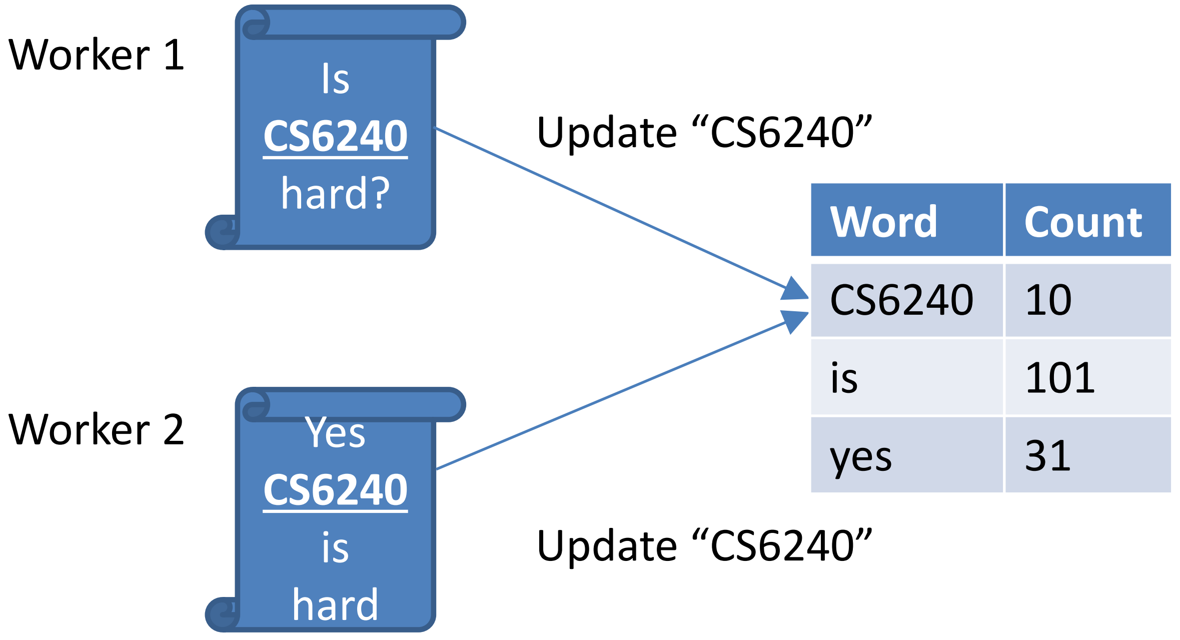


Old state:

Word	Count
CS6240	10
is	100
yes	30

New state:

Word	Count
CS6240	10
is	**101**
yes	**31**



Word	Count
CS6240	10
is	101
yes	31

What could go wrong as both workers try to update the counter for “CS6240” concurrently?

Concurrent Updates and Synchronization

- Updates to a simple hash map H are not atomic, i.e., they consist of multiple low-level operations.
  – The documentation of the data structure would have to explicitly state any atomicity properties a more advanced hash-map implementation may guarantee.
- For simplicity, assume hash-map update H[word]++ consists of the following operations:
  – Read current value of H[word]
  – Add 1 to the value
  – Write the updated value back to H[word]
- What could happen in our example?

Problematic Concurrent Execution

- In this execution, shown advancing from top to bottom, one of the updates is lost. Depending on the system, there might also be an exception.
- This issue is difficult to debug, because the problem may or may not occur, depending on the timing of the operations.

Action by Worker 1		Action by Worker 2	
Read H[CS6240]:	val = 10		
		Read H[CS6240]:	val = 10
		Add 1 to val:	val = 11
Add 1 to val:	val = 11		
		Write val back	H[CS6240] = 11
Write val back	H[CS6240] = 11		

Preventing Concurrency Bugs

- To prevent the problem, the update operation must acquire a lock on the key.
  - The lock ensures exclusive access, i.e., only the worker with the lock can proceed with the update. The other one must wait until the lock owner is done.

```
Function increment (word w) {

   acquire lock on H[w]
   H[w]++
   release lock on H[w]
}
```

Action by Worker 1		Action by Worker 2	
Acquire lock on	H[CS6240]		
Read H[CS6240]:	val = 10		
		Wait for lock on	H[CS6240]
Add 1 to val:	val = 11		
Write val back	H[CS6240] = 11		
Release lock on	H[CS6240]		
		Acquire lock on	H[CS6240]
		Read H[CS6240]	val = 11

Things Are More Complicated

- The first encounter of a word creates a new entry for this key. That operation also must be protected by locking. Unfortunately, we cannot lock the key if it is not in the hash map yet. Hence, we must lock the *entire hash map*.
  - In general, we must lock the object that is accessed. Looking up and inserting a key are operations on the hash-map object.
- Not only updates, but even read-only access must be protected by locking. Otherwise, the value might be overwritten in the middle of the read operation.
  - Some systems distinguish between read locks and write locks. There can be more than one concurrent reader, but as soon as a worker holds a write lock, no concurrent read or write is allowed.
- Locking reduces parallelism by forcing sequential execution of workers competing for the same lock. Hence one should lock as little as possible for as little time as possible.
  - A lock on individual keys allows the update on "is" and on "yes" to proceed in parallel. If the update locks the entire hash map, then these two updates will happen sequentially!

Lessons Learned

- Concurrent access to shared data structures requires careful synchronization, e.g., through locking.
- Locking reduces parallelism and increases computation cost for lock management.
- Ironically, the more workers we use for increased parallelism, the higher the probability of locking conflicts. For this reason, the shared-memory programming approach is generally not considered highly scalable.

Word Count: “Shared-Nothing” Version

- Each worker creates its own local copy of the hash map. It has exclusive access to this copy and hence there are no concurrency issues.
- After all workers are done, they communicate their local counts in order to compute the final counts. This step requires a barrier primitive to ensure that all workers are done. (The barrier is also needed for the shared-memory version!)
- Communication cost may be high, depending on the sizes of the local hash maps. The extra post-processing step can significantly delay job completion. (The shared-memory approach did not need this step!)

Performance Metrics

- Before exploring parallel algorithms in more depth, how do we know if our parallel algorithm or implementation does well?
- There are many measures of program quality in general:
  - Total execution time
  - Total resources consumed
  - Total amount of money paid
  - Total energy consumed
- In practice, one could try to optimize some combination of the above, e.g., minimize total execution time, subject to a monetary cost constraint.
- All these measures matter for both sequential and parallel programs. For parallel programs, there are additional measures of success that determine the effectiveness of the parallelization: speedup and scaleup.

Speedup

- Ideally, if the sequential execution of a program takes time t, then the parallel execution on n processors should take time t/n. We measure how close we come to this ideal scenario by using speedup:
  - Speedup = sequentialTime / parallelTime
- Note: to determine speedup, the work to be done is fixed!
- On n identical machines, the theoretically best possible speedup is n.
  - To see this, note that a sequential program can execute each of the n parallel tasks, one after the other. This takes at most n times as much as the longest of these n tasks.

Scaleup

- Another goal for successful parallelization is to keep the job's response time constant if the number of processors increases at same rate as the "amount of work." We measure how close we come to this ideal scenario by using scaleup:
  - Scaleup = workDoneParallel / workDoneSequential
- Note: to determine scaleup, the time to work on the job is fixed!
- On n identical machines, the theoretically best possible scaleup is n. (The argument is similar to the one for speedup.)
- In practice, it is not easy to measure the "work done." For simplicity, it is often set equal to input size. However, this can lead to confusion when algorithm complexity is not linear in input size.
  - Consider a quadratic algorithm. Doubling input size creates *four* times the work for the algorithm. Hence it is not realistic to expect that doubling the number of machines will ensure the same job completion time on twice the input.
  - For more realistic scaleup, one should take algorithm complexity into account. In practice, we can estimate the work done for a given input by measuring processing time on a single machine.

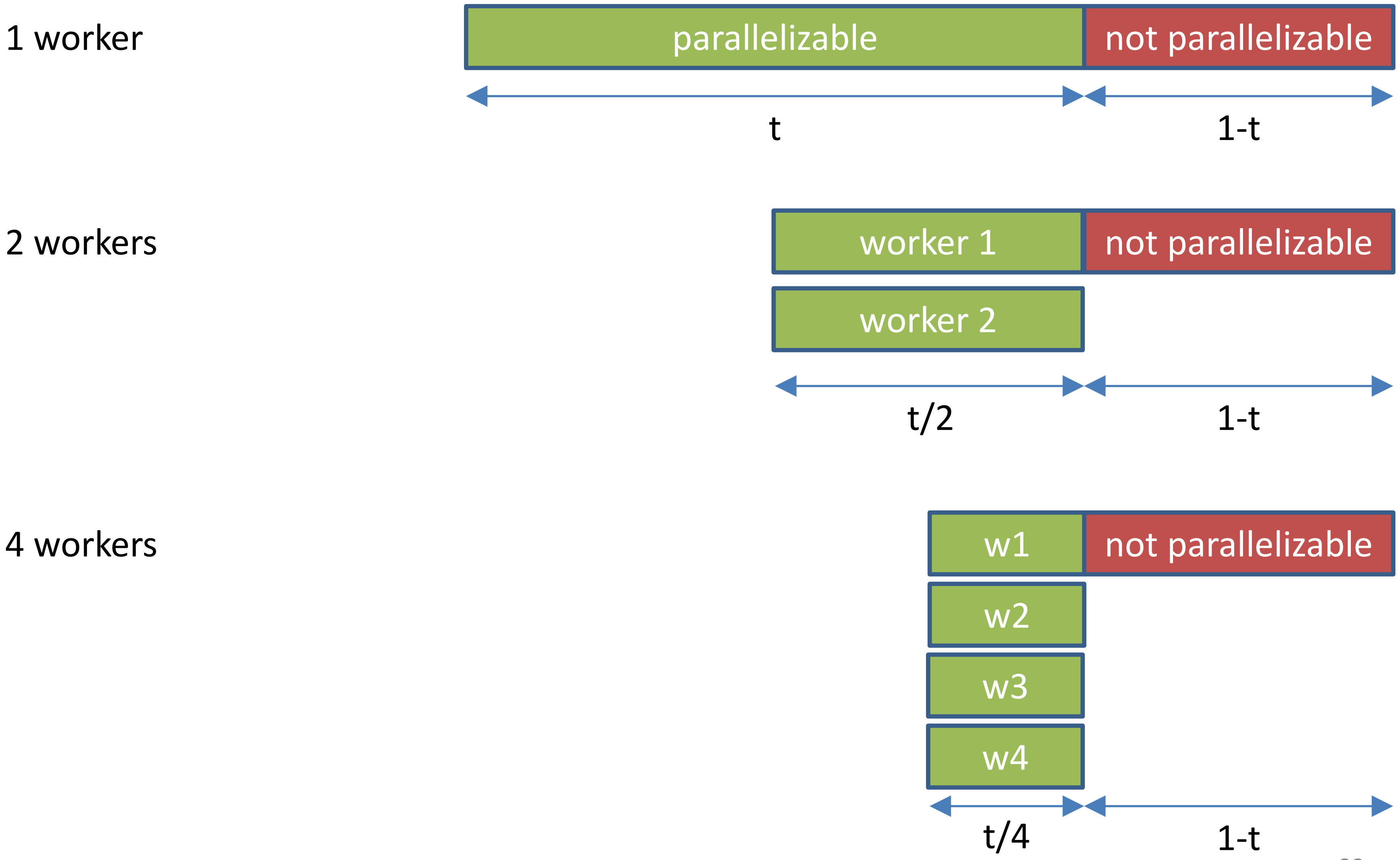
Scalability Through Load Balancing

- Load balancing is **not** a measure of program quality. It is a means of achieving higher speedup and scaleup by distributing work evenly over the processors. This avoids overloading one processor while another is idle.
- To see why balanced load is not a good goal in itself, note that we can “parallelize” a program by sending all data to all processors and having each processor execute the entire program sequentially. This perfectly balances load but results in poor speedup and scaleup.
- Also note that load balancing optimizes for *response time*, but not necessarily for other metrics such as throughput or energy consumption.
- There are two types of load balancing:
  - Static load balancing happens before program execution at compile time. An optimizer analyzes program and data to determine how to partition the work into multiple tasks. It cannot react to runtime events, e.g., an unexpectedly slow machine.
  - Dynamic load balancing happens at runtime, i.e., during program execution. It can react to changes in system state, but incurs additional overhead for monitoring and reacting, e.g., the cost of transferring data and code from one machine to another. Dynamic load balancing is easy for tasks with a simple structure like Web search, but difficult for more complex problems such as joins.

Amdahl’s Law

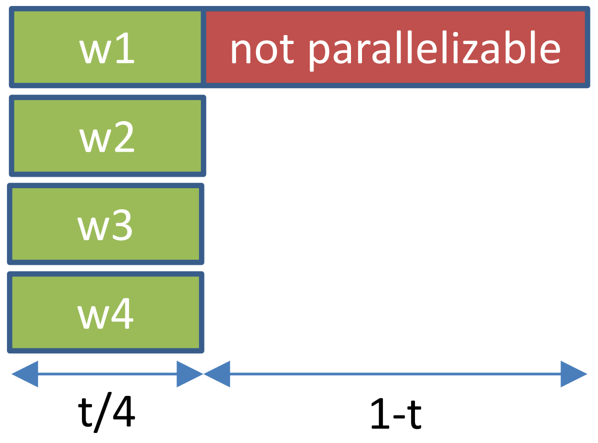
- How do we know if our program’s speedup is “good”? And how do we determine how many machines to use for a task? Amdahl’s Law can help answer both questions.
- Consider a job taking sequential time 1 and consisting of two sequential tasks taking time t and 1-t, respectively. The first task must be completed before the second can start.
- Assume we can perfectly parallelize the first task on n processors, but the second is not parallelizable. Then the best possible parallel running time we can hope for is t/n + (1 – t).
- Hence, we obtain a speedup of 1 / (1 – t(n-1)/n). To understand this formula better, consider an example where the first task makes up 90% of the total work, i.e., only 10% of the job cannot be parallelized. Now use the formula to determine the speedup for different numbers of machines:
  - t=0.9, n=2: speedup = 1.81
  - t=0.9, n=10: speedup = 5.3
  - t=0.9, n=100: speedup = 9.2
  - As n approaches infinity, we obtain the maximal possible speedup for t=0.9 as 1/(1-0.9) = 10

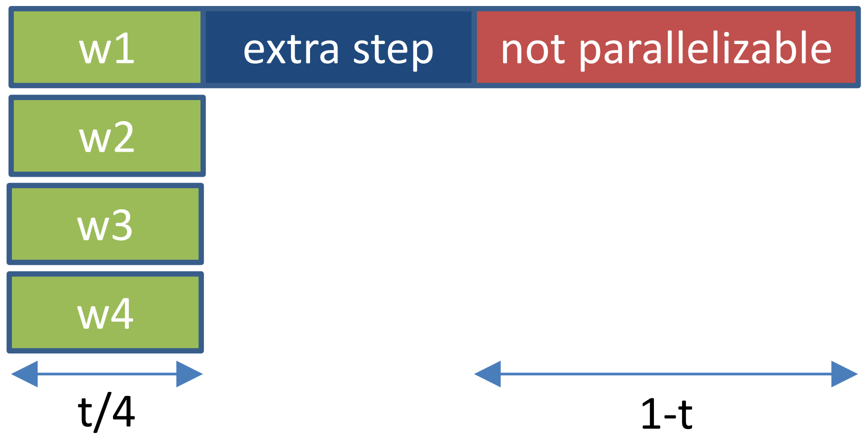
Intuition Behind Amdahl's Law



A More Realistic View

- Parallelization may introduce additional steps, e.g., to communicate and combine the partial results from different workers. For some problems this can negate any positive effect of parallelization.





Implications of Amdahl's Law

- Parallelize the tasks that take the longest. They have the greatest effect on speedup.
- Sequential steps inherently limit the maximum possible speedup. If fraction x of the job is inherently sequential, speedup can never exceed 1/x. Hence there is no point running the job on an excessive number of processors.
  - For x = 0.1, going from 10 to 100 machines improves speedup by less than a factor of 2. Going from 100 to 1000 has no significant effect on speedup.

- This kind of analysis matters in practice. When parallelizing work, we usually introduce additional communication between tasks, e.g., to transmit intermediate results. Such communication can inherently limit speedup, no matter how well the tasks themselves can be parallelized.

Let’s summarize what we have learned so far.

Scalable Data Processing 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.

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 and shared-disk 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 module.

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. For instance, just imagine dealing with locks on shared data structures and managing communication between machines in application code, especially when having to handle failures.
  - If the approach is too high-level, it could suffer from poor performance if control for a crucial bottleneck is “abstracted away.”
- One solution to this dilemma is to use a declarative style of programming. A declarative program specifies WHAT needs to be computed, not HOW this is done at the low level. A well-known success story for declarative programming is SQL for relational databases (RDBMS). An SQL query specifies what the user is looking for and the database optimizer automatically chooses an efficient implementation to compute the desired result.

The MapReduce/Spark 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.
- Place the data near the processors.
  - Moving too much data around tends to result in poor performance. MapReduce therefore tries to assign tasks to machines that already have the data on local disk. Spark takes this even further, by considering data in memory.
- Avoid centralized resources that are likely bottlenecks.
  - Ise them only for “lighter” tasks, e.g., job management.
- Read and write data sequentially in large chunks to amortize latency. (More on this in another module.)