DSC204A: Scaleable Data Systems Winter 2025

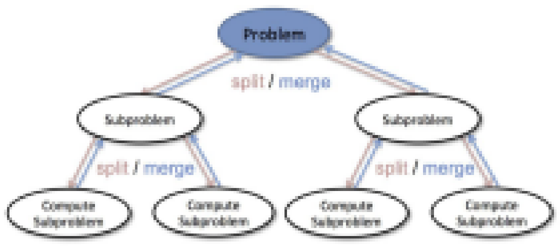
13: Task Parallelism

Instructor: Umesh Bellur Scribe: John Driscoll, Qiyu Li, Jason Wang

1 Introduction to Parallel Computing

1.1 Central Issue

When the work is too much for one processor, we encounter an issue that the it takes too long to process the workload. So we come up with the idea that splitting up workload across processors and perhaps also across machines/workers (aka “Divide and Conquer”)



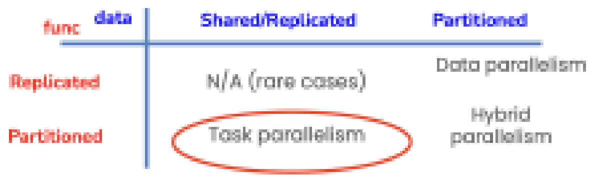
1.2 Key Parallelism Types

There are several key paradigms of parallelism have emerged in data processing systems that incorporate coordination mechanisms. These approaches differ in how they distribute computational work and data across available resources:

Task parallelism refers to executing different operations simultaneously, potentially on the same or different data. This approach is particularly effective when a workflow consists of multiple independent operations that can be performed concurrently.

Data parallelism involves applying the same operation across different portions of data simultaneously. This paradigm shines when processing large datasets where the same computation must be applied to each data element independently.

Hybrid parallelism combines both task and data parallel approaches to maximize computational efficiency. Modern high-performance computing systems often implement hybrid strategies to address complex compu tational challenges that benefit from both forms of parallelism.



func \ data	Shared/Replicated	Partitioned
Replicated	N/A (rare cases)	Data parallelism
Partitioned	Task parallelism	Hybrid parallelism

1.3 Terminology Across Computing Domains

The field of parallel computing spans multiple domains, each with its own terminology to describe similar concepts. This diversity of language reflects the various perspectives and historical developments within different computing communities:

1.3.1 Architecture and Parallel Computing

In computer architecture and traditional parallel computing, terminology focuses on hardware organiza tion and instruction execution patterns. Single-node multi-core systems execute parallel workloads within a shared memory environment. SIMD (Single Instruction, Multiple Data) architectures apply the same oper ation to multiple data elements simultaneously. MIMD (Multiple Instruction, Multiple Data) systems allow different operations on different data concurrently. SIMT (Single Instruction, Multiple Threads) is a model used particularly in GPU computing where multiple threads execute the same instruction but maintain independent program counters.

1.3.2 Distributed Systems

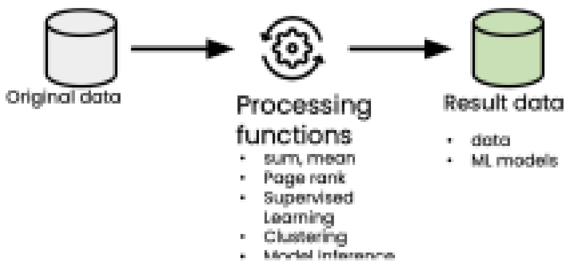
The distributed systems community emphasizes the organization of computation across multiple connected machines. Multiple-node multi-core architectures distribute work across networked computers, each with multiple processing cores. SPMD (Single Program, Multiple Data) is a common paradigm where the same program executes on multiple processors but operates on different portions of the data. MPMD (Multiple Program, Multiple Data) allows different nodes to execute different programs on different data, providing greater flexibility for heterogeneous workloads.

1.3.3 Machine Learning Community

In machine learning, parallelism terminology focuses on model structure and training methodology. The distinction between data parallelism and model parallelism reflects different strategies for distributing neu ral network training across compute resources. Inter-operator parallelism distributes different operations in a computational graph across processors, while intra-operator parallelism parallelizes the execution of individual operations internally.

1.4 Data Processing: Abstraction

At its core, data processing follows a generalized pipeline that transforms original data through various processing functions to produce result data. This abstraction provides a framework for understanding how parallel computing can accelerate data processing workflows.



2 Task Graph

2.1 Relational Task Graph

Operations that occur in database query processing can be represented as a task graph. The expression $\pi(\sigma(R) \cup S \bowtie T)$ can be broken apart into its input data, intermediate data, and operators. This is also called a logical query plan in the database systems world.

2.2 Machine Learning Task Graph

Transformations that occur in a machine learning task, such as an activation function, can be represented as a task graph. The function $ReLU(W X + b)$ can be broken apart into its input data, intermediate data, and operators. This is also called a neural network computational graph.

2.3 ChatGPT Task Graph

Asking ChatGPT to describe its own task graph will give the same system we've come to expect, a divide and conquer method behind the scenes. Different functions are applied within the task from input to a resulting output.

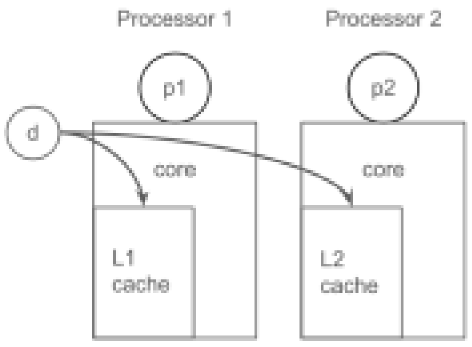
2.4 Raising the Level of Abstraction

Task graphs can then be applied to lines of code, functions and operators (in machine learning), entire functions. For example, null value detection, number of null values in a dataset, value interpolation, and more. No matter the level of abstraction we can apply the same concept.

3 Task Parallelism

Task Parallelism distributes tasks across workers. If there is a shared dataset, data replication may occur across workers or caches.

In a single node file system with multiple cores, if two workers are simply reading the same file, replication of the file isn't necessary. This data would be cached and made available to workers. However, simultaneous writing would require duplication. At the processor code level, if there are processes p_1 and p_2 requiring that data both will have a copy of the data in their cache.



Running parallel tasks on different machines will require replication, because a file system can only reside in one space and remote reads are expensive. The cost of this increase in speed is the time and space used for replication.

Aside: Threading

Both the abstraction and entity existing at runtime are a process within the operating system, because anytime you run a program it's a process. In order to achieve parallelism, the program must make an expensive fork call to create another process.

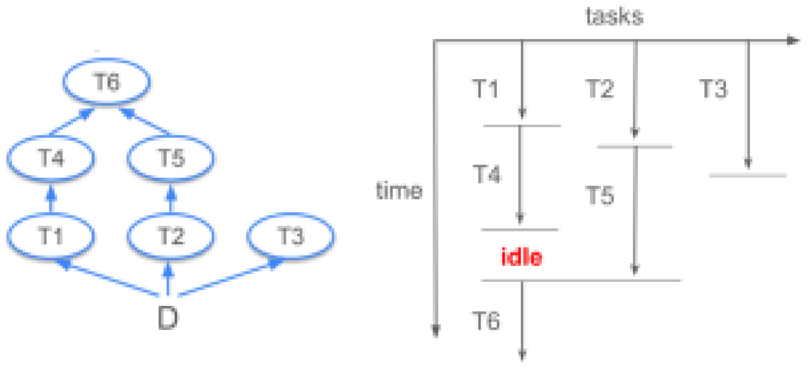
Threading addresses this cost with a lightweight, within-process method. A process that is not multi-threaded has a single thread of control. Each thread is associated with a program counter which runs instructions. With multiple threads, both exist and share the same address space, memory, and variables. (Separate processes do not share address spaces). Threading is lightweight because there is minimal duplication.

Threading can be supported by the OS or a programming language. When the OS is scheduling, it can schedule at the process level or thread level (user choice). OS thread bookeeping requires slightly more overhead.

Python, an interpreted language, as opposed to C a compiled language, has an interpreter that cannot by itself multi-thread (non-reentrant) as it can only run a single copy of itself. Python can't truly do parallel processing, but it can schedule threads once a working thread has dropped. For example, if there are 4 cores and 2 threads in Python, only 1 core will be occupied because of this constraint. This is not the case for C. Rust supports threading as a language level first class entity construct.

In a task graph, separate functions don't need to 'synchronize', however there must be an external entity / manager to keep track of tasks.

We can visualize the run of the parallelized program in terms of time:



Degree of Parallelism is the maximum amount of concurrency in a program at a time. This does not describe the benefit of parallelism.

4 Quantifying Benefit of Parallelism: Speedup

4.1 Definition of Speedup

$$\text{Speedup} = \frac{\text{completion time with only 1 worker}}{\text{completion time with } n (> 1) \text{workers}}$$

We use speedup to quantify the benefit of parallelism execution. Speedup is the ratio of the amount of time taken on a sequential execution of the program to the amount of time taken on a parallel execution.

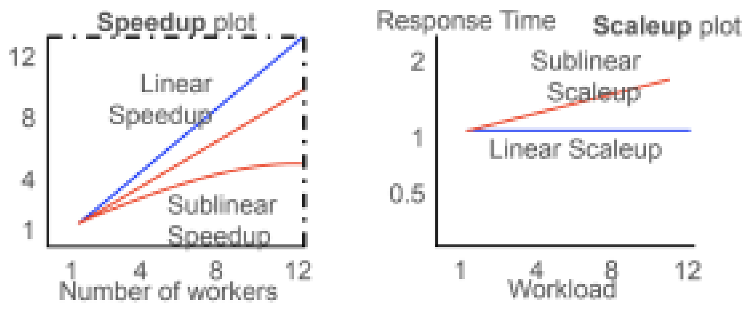
Given *N* workers, ideally we can get a speedup of *N* (i.e., linear speedup). However, many factors may affect the actual speedup, such as degree of parallelism, task dependency graph structure, and intermediate data sizes.

Scalability refers to the ability of the system to retain the same performance as the workload goes up. So, ideally we want linear scaleup: as the number of resources keep increasing, the ratio of tasks per resources remains effective.

However, most systems only achieve the sublinear speedup due to overheads that exist within the system. Similarly, they exhibit sublinear scaleup, where the response time increases as the load goes up.

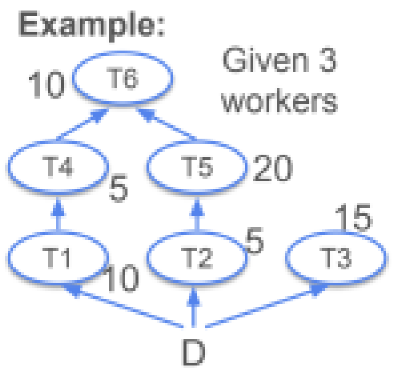
4.2 Calculating Task Parallelism Speedup

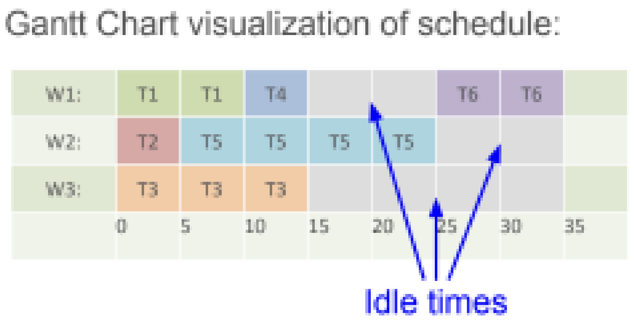
The overall workload's completion time on task-parallel setup is always lower bounded by the longest path in the task graph, which represents the sequential part execution of the flow. For example, after task T4 completes in 5 time units, it must wait for task T5 to finish before proceeding to the subsequent task (T6), resulting in idle time. Similarly, Task T4 cannot be parallelized with Task T1.



6 *7: Task Parallelism*

Completion time with one worker: 10 + 5 + 15 + 5 + 20 + 10 = 65
Parallel completion time: 35
Speedup: 65/35 = 1.9x, while the linear speedup is 3x.





4.3 Amdahl's Law

Amdahl's Law upper bounds the possible speedup, since only the non-sequential part of the program can benefit from multi-core parallelism.

Formula: Given *n* resources,

$$\text{Speedup} = \frac{1}{S + 1/N(1 - S)}$$

where *S* refers to the fraction of the program that is sequential.