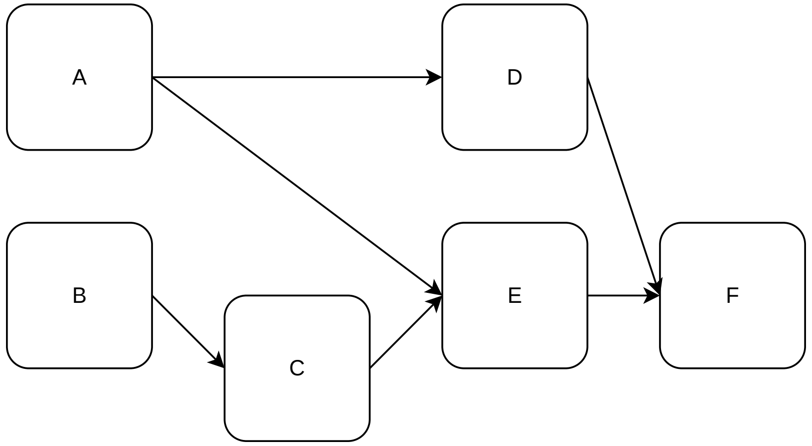
Task Parallelism

Task Graphs



Multiple Instruction Multiple Data (MIMD) parallelism.

Execution time of tasks might be vastly different.

Graph might be static or dynamic.

When to use it?

Whenever you have heterogenous computation on heterogenous data.

Examples:

- compiler pipelines
- computer games
- unbalanced search trees
- ...

Only when regular and nested data parallelism do not apply.

Only when speedup through parallelism exceeds coordination overhead.

Java with Project Loom

https://docs.oracle.com/en/java/javase/25/core/virtual-threads.html

```java
Thread.Builder builder = Thread.ofVirtual().name("MyThread");
Runnable task = () -> {
    System.out.println("Running thread");
};
Thread thread = builder.start(task);
System.out.println("Thread t name: " + thread.getName());
thread.join();
```

Project Loom lead by Ron Pressler, many contributers, started 2017, merged 2022.

Data Parallelism in Java

https://docs.oracle.com/javase/tutorial/collections/streams/parallelism.html

> To create a parallel stream, invoke the operation Collection.parallelStream.

```java
double average = roster
    .parallelStream()
    .filter(p -> p.getGender() == Person.Sex.MALE)
    .mapToInt(Person::getAge)
    .average()
    .getAsDouble();
```

> Note that parallelism is not automatically faster than performing operations serially, although it can be if you have enough data and processor cores. [...], it is still your responsibility to determine if your application is suitable for parallelism.

demo/Sparse.java

Promises and Futures

> The evaluator does this by creating and returning for each subexpression a *future*, which is a promise to deliver the value of that subexpression at some later time, if it has a value. Each future can evaluate its subexpression independently and concurrently with other futures because it is created with its own evaluator *process*, which is dedicated to evaluating its subexpression.

Henry C. Baker and Carl Hewitt. 1977. The incremental garbage collection of processes.

Promise: Must be written to (fulfilled, resolved, completed) exactly once.

Future: Can be read from (awaited) more than once.

In practice: a promise and a future are the same mutable reference cell.

Task Parallelism in Java

```java
ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor();

Future<Integer> A = executor.submit(() -> 1 + 1);

Future<Integer> B = executor.submit(() -> 2 + 2);

Future<Integer> C = executor.submit(() -> A.get() + B.get());

System.out.println(C.get());
```

https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/concurrent/ExecutorService.html

https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/concurrent/Future.html

demo/Basics.java

No Cycles when using Futures

Reading from a future suspends work until the promise is fulfilled.

Each task can only depend on futures created earlier.

```
Future<Integer> A = executor.submit(() -> B.get() + 1); // B is not in scope

Future<Integer> B = executor.submit(() -> A.get() + 1);
```

No cycles in task graph and therefore no deadlocks are possible.

Granularity

Tasks are the unit of parallelism.

Coarse-grained tasks: low parallelism.

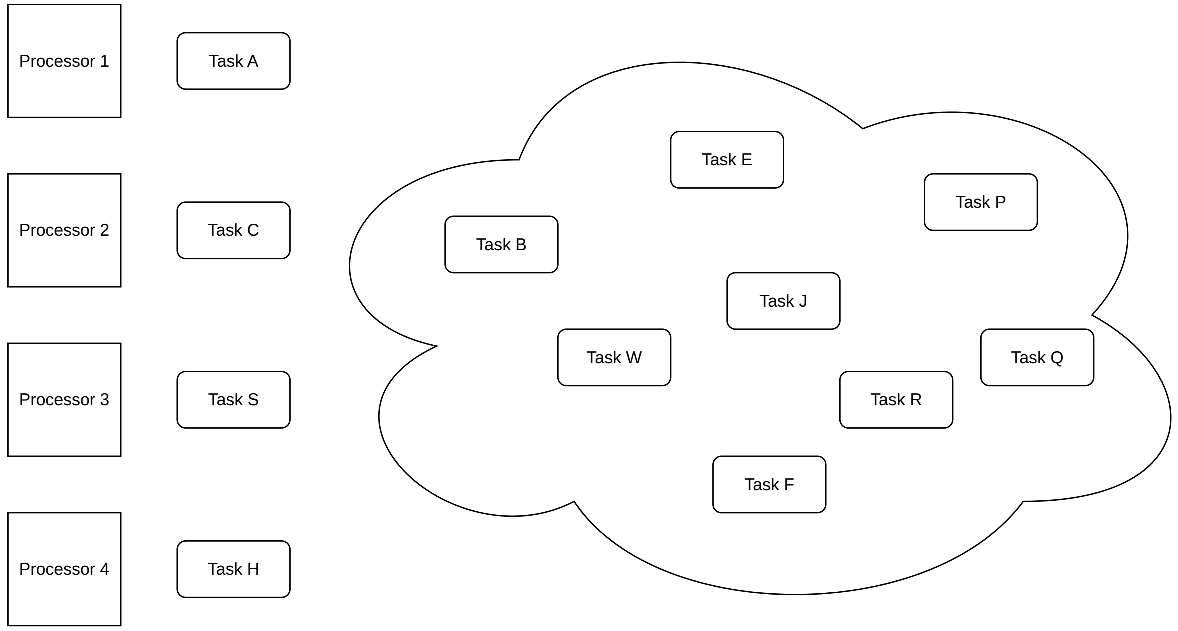
Fine-grained tasks: high overhead.

We need to balance both.

demo/Mergesort.java

Work Stealing

Work stealing attempts to make sure all processors are fully utilized.

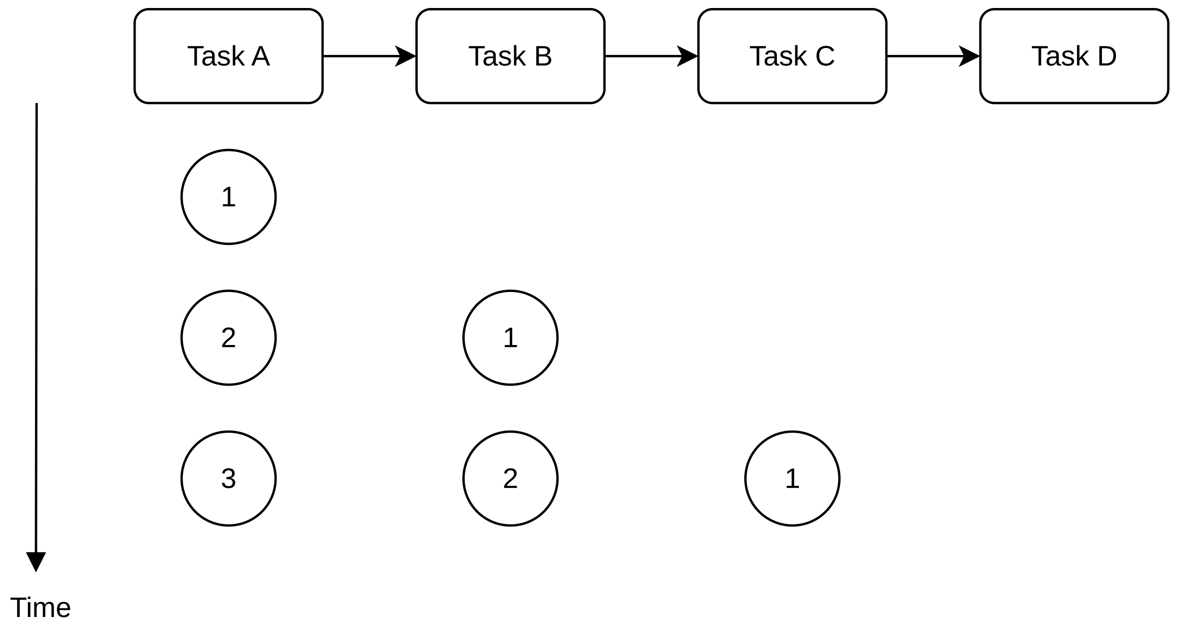


Tasks can be created dynamically and their execution time varies.

demo/SocialNetwork.java

Pipelining

When processing a stream of items, divide computation into stages.



Each stage processes items in parallel to other stages.

Blocking Queues

First-in-first-out (FIFO) data and control structure.

Used for communication and synchronization between stages.

Can in theory be unbounded, should in practice be bounded.

Consumer blocks when empty, producer blocks when full.

Blocking Queues in Java

```java
BlockingQueue<String> queue = new ArrayBlockingQueue<>(10);

queue.put("Hello");
queue.put("World");

String hello = queue.take();
String world = queue.take();
```

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ArrayBlockingQueue.html

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/LinkedBlockingQueue.html

Pipelining in Java

```
BlockingQueue<Integer> queue = new ArrayBlockingQueue<>(1);
executor.execute(() -> producer(queue));
executor.execute(() -> consumer(queue));
```

```
static void producer(BlockingQueue<Integer> queue) {
    try {
        for (int i = 0; i < 5; i++) queue.put(i);
    } catch (InterruptedException e) {}
}

static void consumer(BlockingQueue<Integer> queue) {
    try {
        for (int i = 0; i < 5; i++) System.out.println(queue.take() * 2);
    } catch (InterruptedException e) {}
}
```

"Execution in the Kingdom of Nouns"

demo/Pipeline.java

Cycles when using Queues

```
BlockingQueue<Integer> queue1 = new ArrayBlockingQueue<>(1);

BlockingQueue<Integer> queue2 = new ArrayBlockingQueue<>(1);

executor.execute(() -> { queue2.put(queue1.take() * 2); });

executor.execute(() -> { queue1.put(queue2.take() + 1); });
```

Cyclic dependencies cause deadlocks.

demo/Deadlock.java

Poison Pill Pattern

Mark the end of the stream with a sentinel value.

Signal from producer to consumer that it should stop.

One producer, multiple consumers: send one poison pill for each consumer.

Multiple producers, one consumer: count number of poison pills.

How can the consumer signal to the producer to stop?

Poison Pill in Java

```java
// producer
for (int i = 0; i < 5; i++) {
    queue.put(Optional.of(i));
}
queue.put(Optional.empty());
```

```java
// consumer
while (true) {
    Optional<Integer> value = queue.take();
    if (value.isEmpty()) break;
    System.out.println(value.get());
}
```

demo/PoisonPill.java

Batching

Accumulate items, process entire batches.

Processing a batch might be more efficient: work sharing, data parallelism, ...

Pay in latency, gain in throughput.

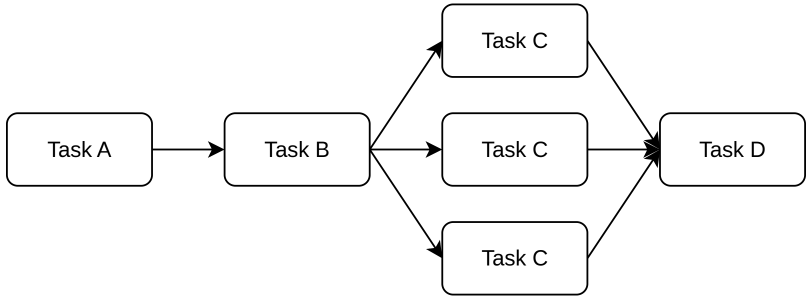
demo/Batching.java

Balancing

In a pipeline, each stage should take roughly the same amount of time.

Blocking queues naturally apply backpressure when consumer cannot keep up.

We can also assign multiple tasks to the same stage for balancing.



The order of items is not preserved.

demo/Balancing.java

Input and Output

Computation on other devices and machines can happen in parallel.

Example: fetch data from a database.

```
Future<Integer> A = executor.submit(() -> fetch(8001));
Future<Integer> B = executor.submit(() -> fetch(8002));
System.out.println(A.get() + B.get());
```

demo/Client.java

Summary and Outlook

Task parallelism is flexible.

Next week: Asynchronous communication.