Parallelizing computations

TDA384/DIT391

Principles of Concurrent Programming

Nir Piterman and Gerardo Schneider

Chalmers University of Technology | University of Gothenburg

Lesson’s menu

- Challenges to parallelization
- Fork/join parallelism
- Pools and work stealing

Lesson’s menu

- Challenges to parallelization
    - evaluate efficiency of different solution
- Fork/join parallelism
    - programming constructs
- Pools and work stealing
    - Programming constructs, efficiency

Learning outcomes

Knowledge and understanding:

- demonstrate knowledge of the issues and problems that arise in writing correct concurrent programs;
- identify the problems of synchronization typical of concurrent programs, such as race conditions and mutual exclusion

Skills and abilities:

- apply common patterns, such as lock, semaphores, and message-passing synchronization for solving concurrent program problems;
- apply practical knowledge of the programming constructs and techniques offered by modern concurrent programming languages;
- implement solutions using common patterns in modern programming languages

Judgment and approach:

- evaluate the correctness, clarity, and efficiency of different solutions to concurrent programming problems;
- judge whether a program, a library, or a data structure is safe for usage in a concurrent setting;
- pick the right language constructs for solving synchronization and communication problems between computational units.

Parallelization: risks and opportunities

Concurrent programming introduces:

\+ the potential for parallel execution (faster, better resource usage)

− the risk of race conditions (incorrect, unpredictable computations)

The main challenge of concurrent programming is thus introducing parallelism without affecting correctness

There is no panacea!

We show several (common?) paradigms where some difficulties can be mitigated.

Paradigms of parallelization

In this lesson, we explore several paradigms to parallelizing computations in multi-processor systems

A task (F, D) consists in computing the result $F(D)$ of applying function F to input data D

A parallelization of (F, D) is a collection $(F_1, D_1), (F_2, D_2), \ldots$ of tasks such that $F(D)$ equals the composition of $F_1(D_1), F_2(D_2), \ldots$.

We discuss how to parallelize such problems in the context of shared-memory models (such as Java threads).

We note that similar solutions are possible in Erlang using message-passing between processes.

Challenges to Parallelization

Challenges to parallelization

A strategy to parallelize a task (F, D) should be:

- correct: the overall result of the parallelization is $F(D)$
- efficient: the total resources (time and memory) used to compute the parallelization are less than those necessary to compute (F, D) sequentially

A number of factors challenge designing correct and efficient parallelizations:

- sequential dependencies
- synchronization costs
- spawning costs
- error proneness and composability

Sequential dependencies

- Some steps in a task computation depend on the result of other steps; this creates sequential dependencies where one task must wait for another task to run
- Sequential dependencies limit the amount of parallelism that can be achieved

For example, to compute the sum $1 + 2 + \cdots + 8$ we could split into:

a. computing $1 + 2$, $3 + 4$, $5 + 6$, $7 + 8$
b. computing $(1 + 2) + (3 + 4)$ and $(5 + 6) + (7 + 8)$
c. computing $((1 + 2) + (3 + 4)) + ((5 + 6) + (7 + 8))$

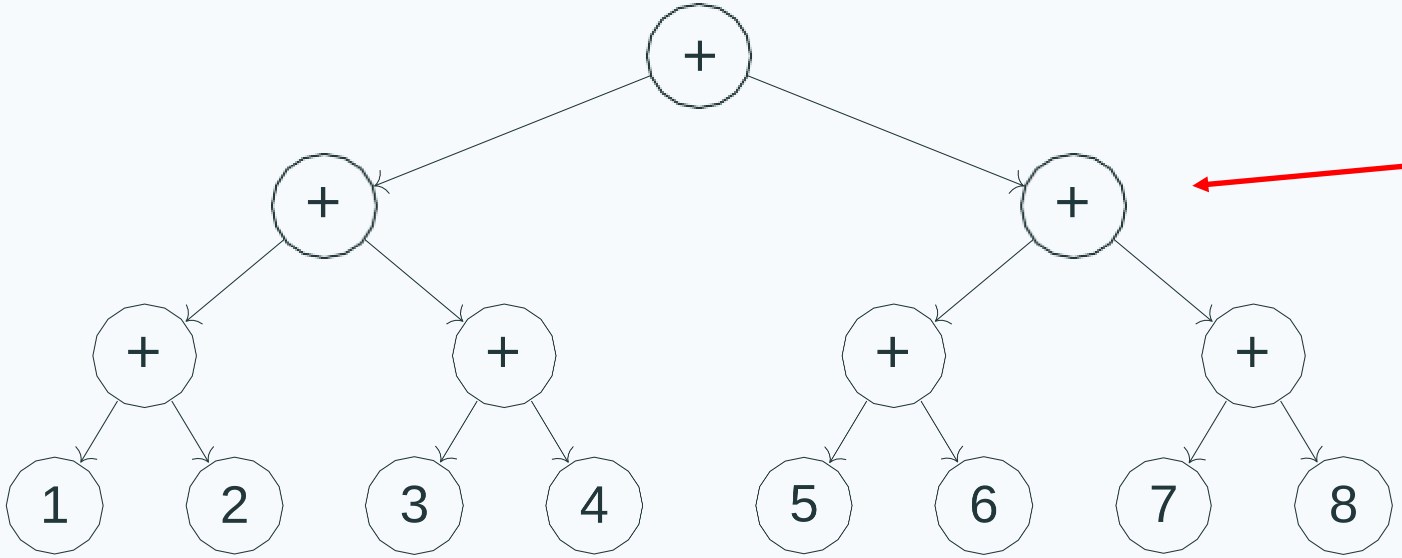
The computations in each group depend on the computations in the previous group, and hence the corresponding tasks must execute after the latter have completed

The synchronization problems (producer-consumer, dining philosophers, etc.) we discussed capture kinds of <u>sequential dependencies</u> that may occur when parallelizing

Dependency graph

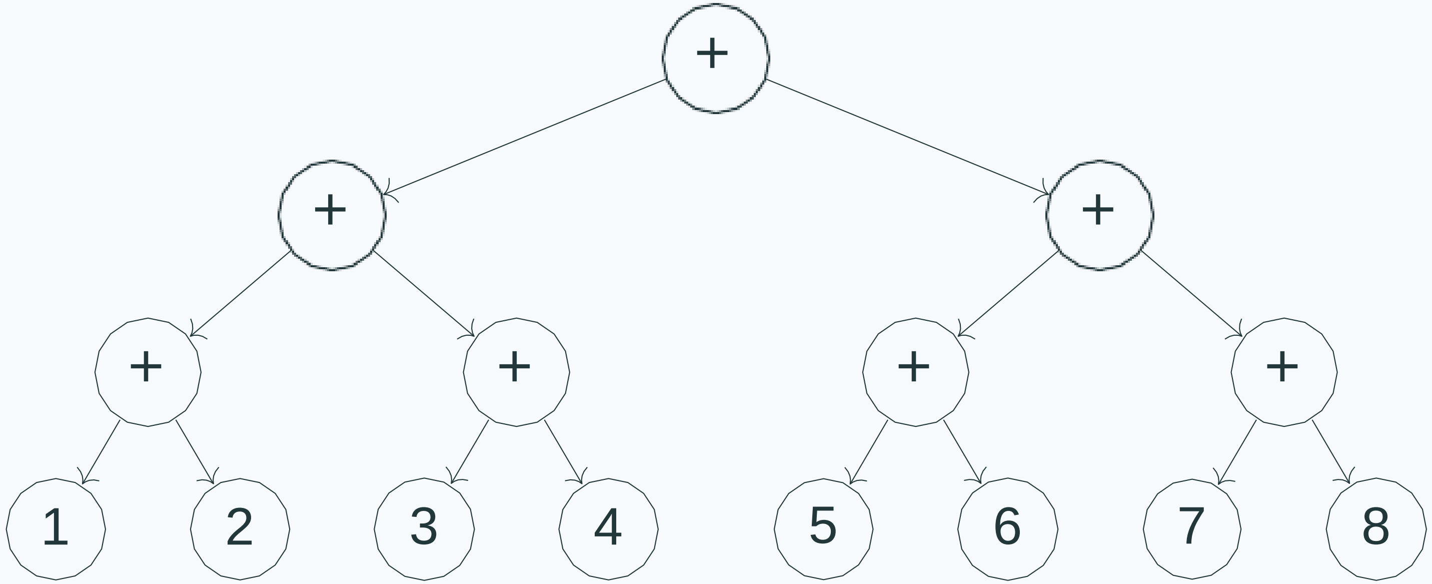
We represent tasks as the nodes in a graph, with arrows connecting a task to the ones it depends on

The graph must be acyclic for the decomposition to be executable



It works well here: there is symmetry given that ”+” is associative and commutative

Dependency graph



The time to compute a node is the maximum of the times to compute its children plus the time computing the node itself

Assuming all operations take a similar time, the longest path from the root to a leaf is proportional to the optimal running time with parallelization (ignoring overhead and assuming all processes can run in parallel)

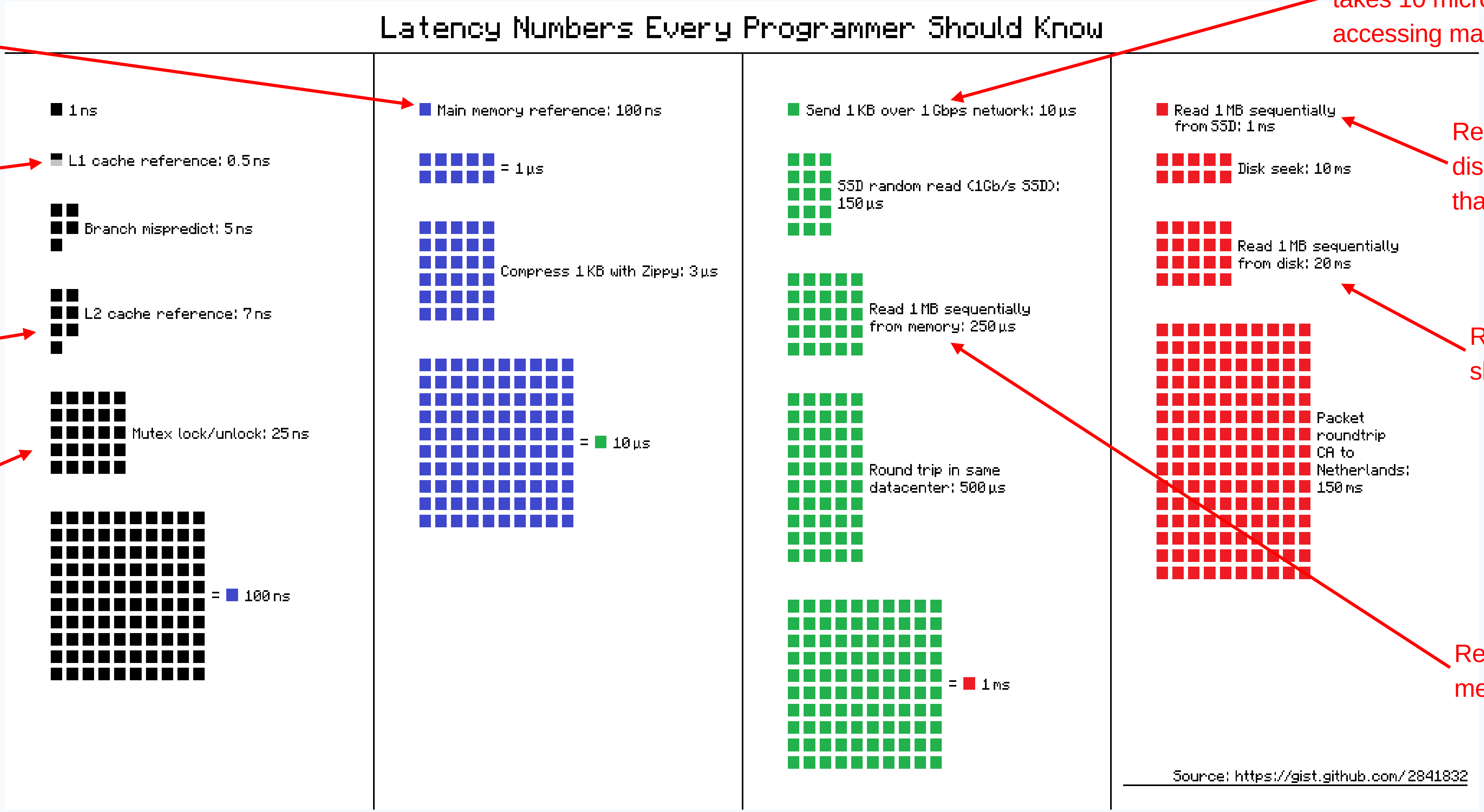
Digression: some latency numbers

Accessing main memory: 100 ns

Accessing an instruction in L1: 0.5 ns

If instruction in L2 (eg, different chip): 14 times slower

To lock/unlock a mutex: 25 ns



Send 1 Kb data on 1Gbps network takes 10 micro sec (100x more than accessing main memory)

Read 1 MB from an SSD disk: 1 ms (4 times slower than from main memory)

Read 1 MB from disk: 20x slower than from SSD disk

Read 1 MB from main memory: 250 micro sec

Chart by ayshen, based on Peter Norvig's "Teach Yourself Programming in Ten Years"

More numbers at https://gist.github.com/hellerbarde/2843375

Synchronization costs

Synchronization is required to preserve correctness, but it also introduces overhead that add to the overall cost of parallelization

In shared-memory concurrency:

- synchronization is based on locking
- locking synchronizes data from cache to main memory, which may involve a 100x overhead
- other costs associated with locking may include context switching (wait/signal) and system calls (mutual exclusion primitives)

In message-passing concurrency:

- synchronization is based on messages
- exchanging small messages is efficient, but sending around large data is quite expensive (still goes through main memory)
- other costs associated with message passing may include extra acknowledgment messages and mailbox management (removing unprocessed messages)

Spawning costs

Creating a new process is generally expensive compared to sequential function calls within the same process, since it involves:

- reserving memory
- registering the new process with runtime system
- setting up the process's local memory (stack and mailbox)

Even if process creation is increasingly optimized, the cost of spawning should be weighted against the speed up that can be obtained by additional parallelism

In particular, when the processes become way more than the available processors, there will be <u>diminishing returns</u> with more spawning

Error proneness and composability

Synchronization is prone to errors such as data races, deadlocks, and starvation

From the point of view of software construction, the lack of composability is a challenge that prevents us from developing parallelization strategies that are generally applicable

Error proneness and composability

Consider an `Account` class with methods `deposit` and `withdraw` that execute atomically

What happens if we combine the two methods to implement a `transfer` operation?

```
class Account {
  synchronized void
    deposit(int amount)
    { balance += amount; }
  synchronized void
    withdraw(int amount)
    { balance -= amount; }
}
```

execute uninterruptedly

```
class TransferAccount
    extends Account {
 // transfer from 'this' to 'other'
 void transfer(int amount, Account other)
  { this.withdraw(amount);
    other.deposit(amount); }
}
```

Method `transfer` does **not** execute uninterruptedly: other threads can execute between the call to `withdraw` and the call to `deposit`, possibly preventing the transfer from succeeding

(For example, Account `other` may be closed; or the total balance temporarily looks lower than it is!)

Composability

```
class Account {
  void // thread unsafe!
    deposit(int amount)
    { balance += amount; }
  void // thread unsafe!
    withdraw(int amount)
    { balance -= amount; }
}
```

```
class TransferAccount
    extends Account {
 // transfer from ‘this’ to ‘other’
 synchronized void
   transfer(int amount, Account other)
   { this.withdraw(amount);
     other.deposit(amount); }
}
```

None of the natural solutions to composing is fully satisfactory:

- let clients of `Account` do the locking where needed – error proneness, revealing implementation details, scalability
- recursive locking – risk of deadlock, performance overhead

With message passing, we encounter similar problems – synchronizing the effects of messaging two independent processes

Sequential dependencies and spawning costs

A number of factors challenge designing correct and efficient parallelizations:

- sequential dependencies
- synchronization costs
- spawning costs
- error proneness and composability

In the rest of this lesson, we present:

- fork/join parallelism – naturally capture sequential dependencies
- pools – curb spawning costs

In future lessons we will see other approaches to reduce synchronization costs and achieving composability

Fork/join parallelism

Mitigating Spawning Costs

In the rest of this lesson, we present:

- fork/join parallelism – naturally capture sequential dependencies
- pools – curb spawning costs

Apply to problems that look like this:

A task (F, D) consists in computing the result

$F(D)$ of applying function F to input data D

A parallelization of (F, D) is a collection $(F_1, D_1), (F_2, D_2), \ldots$ of tasks such that $F(D)$ equals the composition of $F_1(D_1), F_2(D_2), \ldots$.

What kind of problems look like this?

Recursion: merge sort

```java
// Allocate space and call recursive merge
// sort.
static void mergeSort(int[] arr, int size) {
    int[] space = new int[size];
    mergeSortRec(arr, 0, size, space);
}

// Recursive merge sort
static void mergeSortRec(int[] arr,
                         int low,
                         int high,
                         int[] space) {
    if (high - low <= 1) return;
    int mid = low + (high - low) / 2;
    mergeSortRec(arr, low, mid, space);
    mergeSortRec(arr, mid, high, space);
    merge(arr, low, mid, high, space);
}
```

```java
static void merge(int[] arr, int low,
                  int mid, int high,
                  int[] space) {
    int i = low; int j = mid; int k = low;
    while (i < mid && j < high)
    {
        if (arr[i] <= arr[j])
            space[k++] = arr[i++];
        else
            space[k++] = arr[j++];
    }
    while (i < mid)
        space[k++] = arr[i++];
    while (j < high)
        space[k++] = arr[j++];

    for (i = low; i < high; i++)
        arr[i] = space[i];
}
```

Parallel recursion

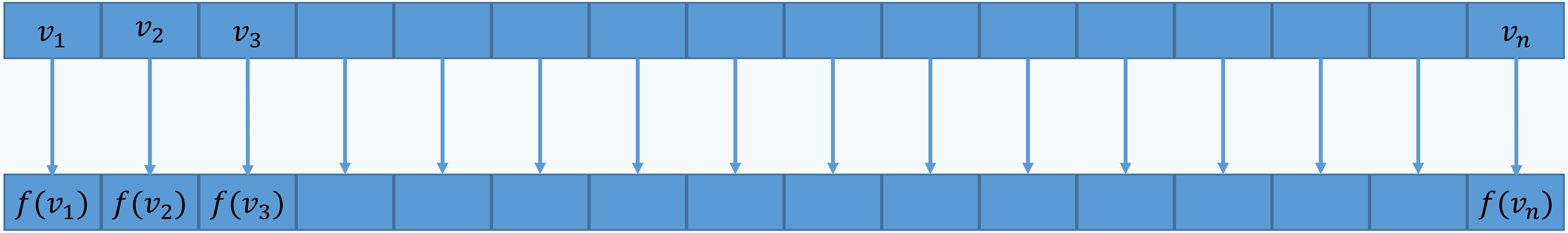
```
// Allocate space and call recursive merge
// sort.
static void mergeSort(int[] arr, int size) {
    int[] space = new int[size];
    mergeSortRec(arr, 0, size, space);
}

// Recursive merge sort
static void mergeSortRec(int[] arr,
                         int low,
                         int high,
                         int[] space) {
    if (high - low <= 1) return;
    int mid = low + (high - low) / 2;
    mergeSortRec(arr, low, mid, space);
    mergeSortRec(arr, mid, high, space);
    merge(arr, low, mid, high, space);
}
```

```
// This function calls recursive merge sort.
    public static void mergeSort(int[] arr, int size)
    {
        int[] space = new int[size];
        MergeSortParallel m = new
                MergeSortParallel(arr,0,size,space);
        m.run();
//      mergeSortRec(arr, 0, size, space);
    }
```

```
static class MergeSortParallel extends Thread {
        public void run() {
            if (high - low <= 1) return;
            int mid = low + (high - low) / 2;
            Thread l = new
               MergeSortParallel(arr,low,mid,space);
            Thread r = new
              MergeSortParallel(arr,mid,high,space);

            l.start(); r.start();

            try {
                l.join(); r.join();
                merge(arr, low, mid, high, space);
            } catch (InterruptedException e) {}
        }
    }
```

Map / forEach

- Apply a given function to all elements in a collection.
- Natural concept in functional programming.
- Introduced also in imperative programming languages (C++, Java, …).
- The approach to doing this in Java here is structured towards concurrency …



Parallel map

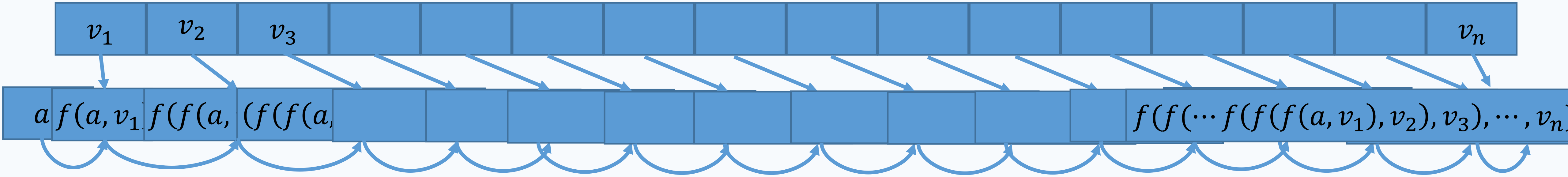
The lack of interference in `map` lends itself to parallelization

```
public class Map<X,Y> {

    protected ArrayList<X> source;
    protected Function<X,Y> func;
    protected ArrayList<Y> target;

    Map(ArrayList<X> source,
        Function<X,Y> func,
        ArrayList<Y> target) {
        …
    }

    public void map() {
        for (int i=0 ; i< source.size(); i++) {
            target.set(i,func.apply(source.get(i)));
        }
    }
}
```

```
public class ParallelMap<X,Y> extends Map<X,Y> {

    ParallelMap(ArrayList<X> source, … ) { … }

    public class Applicator implements Runnable {
        int loc;
        Applicator(int loc) { … }

        public void run() {
            target.set(loc,func.apply(source.get(loc)));
        }
    }

    public void map() {
        ArrayList<Thread> threads =
                  new ArrayList<Thread>(source.size());;

        for (int i=0 ; i<source.size() ; i++) {
            threads.set(i,new Thread(new Applicator(i)));
            threads.get(i).start();
        }
        try {
            for (int i=0 ; i<source.size() ; i++) {
                threads.get(i).join();
            }
        } catch (InterruptedException e) {}
    }
}
```

Reduce – summarize a collection

- Apply a given function starting from an initial value and accumulating the result applied to all elements in a collection.
- Natural concept in functional programming.
- Introduced also in imperative programming languages (C++, Java, …).
- The approach to doing this in Java here is structured towards concurrency …



Parallel reduce

The parallel version of `reduce` (aka `foldr`) uses a halving strategy similar to merge sort

```
import java.util.function.BinaryOperator;

public class Reduce<X> {

    protected ArrayList<X> source;
    protected BinaryOperator<X> func;
    protected X initial;
```

```
}
```

`Parallel reduce` equals `reduce` if:

- Function `F` is associative (parallel reduce does not apply `F` right-to-left)
- For every list element `E`:

  `F(E,init)= F(init,E) = E`

(The data is a monoid with `F` as the binary operation and `init` its identity element)

It works with e.g. addition but not division

```
import java.util.function.BinaryOperator;

public class ParallelReduce<X> extends Reduce<X> {

    ParallelReduce(…) { … }

    public class Applicator extends Thread {
        Applicator(int st, int end, X init) { … }

        public void run() {
            if (end - st > 1) {
                int mid = st + (end - st) / 2;
                Thread l = new Applicator(st,mid,init);
                Thread r = new Applicator(mid,end,init);
                l.start(); r.start();
                try {
                    l.join(); r.join();
                    source.set(start, func.apply(
                               source.get(start), source.get(mid)));
                } catch (InterruptedException e) {}
             else {
                source.set(start,func.apply(initial,
                                            source.get(start)));


          reduce() {
          cator a = new Applicator(0,size,initial);
          ();
        return source.get(0);
    }
}
```

MapReduce

MapReduce is a programming model based on parallel distributed variants of the primitive operations `map` and `reduce`

MapReduce is a somewhat more general model, since it may produce a list of values from a list of key/value pairs, but the underlying ideas are the same

MapReduce implementations typically work on very large, highly parallel, distributed databases or filesystems.

- The original MapReduce implementation was proprietary developed at Google
- Apache Hadoop offers a widely-used open-source Java implementation of MapReduce

Revisiting parallel merge sort

There are a number of things that should be improved in the parallel merge sort example:

granularity too small!

```
protected void run() {
    if (high - low <= 1) return;       // size <= 1: sorted already
    int mid = low + (high - low)/2;    // mid point
    // left and right halves:
    PMergeSort left = new PMergeSort(data, low, mid);
    PMergeSort right = new PMergeSort(data, mid, high);
    left.fork();     // fork thread working on left
    right.fork();    // fork thread working on right
    left.join();     // wait for sorted left half
    right.join();    // wait for sorted right half
    merge(mid);      // merge halves
}
```

the forking thread is idle!

Revisited parallel merge sort using fork/join

choose experimentally (at least 1000)

```
protected void run() {
  if (high - low <= THRESHOLD)
     sequential_sort(data, low, high);
  else {
     int mid = low + (high - low)/2;      // mid point
     // left and right halves
     PMergeSort left = new PMergeSort(data, low, mid);
     PMergeSort right = new PMergeSort(data, mid, high);
     left.fork();     // fork thread working on left
     right.run();     // continue work on right
     left.join();     // when done, wait for sorted left half
     merge(mid);              // merge halves
  }
```

before joining, do more work in current task

Fork/join good practices

In order to obtain good performance using fork/join parallelism:

- After forking children tasks, keep some work for the parent task before it joins the children
- For the same reason, use `invoke` and `invokeAll` only at the top level as a norm
- Perform small enough tasks sequentially in the parent task, and fork children tasks only when there is a substantial chunk of work left
  - Java's fork/join framework recommends that each task be assigned between 100 and 10'000 basic computational steps
- Make sure different tasks can proceed independently – minimize data dependencies

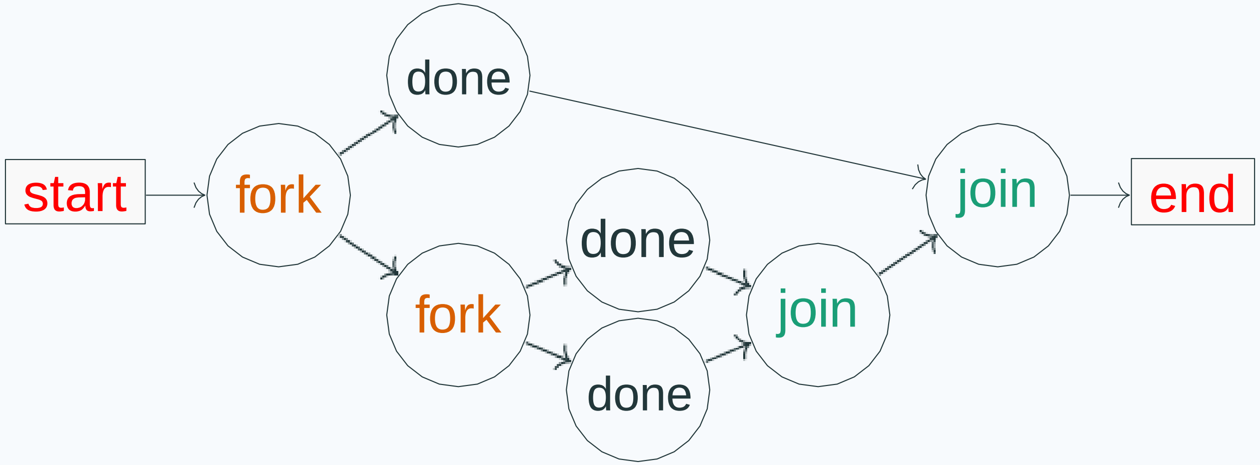
The advantages of parallelism may only be visible with several physical processors, and on very large inputs

(The Java runtime may need to warm up before it optimizes the parallel code more aggressively)

Fork/join parallelism

This recursive subdivision of a task that assigns new processes to smaller tasks is called fork/join parallelism:

- forking: spawning child processes and assigning them smaller tasks
- joining: waiting for the child processes to complete and combining their results



The order in which we wait at a join node for forked children does not affect the total waiting time: if we wait for a slower process first, we won't wait for the others later

What are the issues?

- The number of threads depends on the problem and may choke the computing power.
- How to return a value?

Two similar solutions

- Fork/Join pools
  - Recursively partition the task – what to do with fork and join?
  - Granularity of tasks decreases
  - Load is hard to evenly distribute
  - Terminates
- Executor services
  - Get a bank of tasks and run them in parallel
  - Meant for larger and more predictable coarse-grained tasks
  - Dependencies are simpler
  - Can be terminated

Pools and work stealing

How many processes is *lagom*?

Parallelizing by following the recursive structure of a task is simple and appealing

However, the potential performance gains should be weighted against the overhead of creating and running many processes

- Process creation in Erlang is lightweight: 1 GB of memory fits about 432’000 processes, so one million processes is quite feasible

How many processes is lagom?

Parallelizing by following the recursive structure of a task is simple and appealing

However, the potential performance gains should be weighted against the overhead of creating and running many processes

- There are still limits to how many processes fit in memory
- Besides, even if we have enough memory, more processes don't improve performance if their number greatly exceeds the number of available physical processors

Remember Amdahl's law

Workers and pools

Process pools are a technique to address the problem of using an appropriate number of processes

A pool creates a number of worker processes upon initialization

The number of workers is chosen according to the actual available resources to run them in parallel – a detail which pool users need **not** know about:

- As long as more work is available, the pool deals a work assignment to a worker that is available
- The pool collects the results of the workers' computations
- When all work is completed, the pool terminates and returns the overall result

This kind of pool is called a dealing pool: it actively deals work to workers

Workers

Workers are threads that run as long as the pool that created them does

A worker can be in one of two states:

- idle: waiting for work assignments from the pool
- busy: computing a work assignment

```
public class WorkThread
{ Queue [] queue; // queues of all worker threads
public void run() {
{ int me = ThreadID.get(); // my thread id
  while (true) {
    for (Task task: queue[me]) // run all tasks in my queue
      task.run();
    if (queue[me].empty()) queue[me].await();
  } } }
```

From dealing to stealing

Dealing pools work well if:

- the workload can be split in even chunks, and
- the workload does not change over time (for example if users send new tasks or cancel tasks dynamically)

Under these conditions, the workload is balanced evenly between workers, so as to maximize the amount of parallel computation

In realistic applications, however, these conditions are not met:

- it may be hard to predict reliably which tasks take more time to compute the workload is highly dynamic

Stealing pools use a different approach to allocating tasks to workers that better addresses these challenging conditions

Work stealing

A stealing pool associates a queue to every worker process

The pool distributes new tasks by adding them to the workers' queues

When a worker becomes idle:

- first, it gets the next task from its own queue
- if its queue is empty, it can directly steal tasks from the queue of another worker that is currently busy

With this approach, workers adjust dynamically to the current working conditions without requiring a supervisor that can reliably predict the workload required by each task

With stealing, the pool may even send all tasks to one default thread, letting other idle threads steal directly from it, simplifying the pool and reducing the synchronization costs it incurs

Work stealing algorithm

Outline of the algorithm for work stealing

It assumes the queue array `queue` can be accessed by concurrent threads without race conditions

```
public class WorkStealingThread
{ Queue [] queue; // queues of all worker threads
 public void run() {
 { int me = ThreadID.get(); // my thread id
   while (true) {
     for (Task task: queue[me]) // run all tasks in my queue
       task.run();
     // now my queue is empty: select another random thread
     int victim = random.nextInt(queue.length);
     // try to take a task out of the victim's queue
     Task stolen = queue[victim].pop();
     // if the victim's queue was not empty, run the stolen task
     if (stolen != null) stolen.run();
   } } }
```

Fork/join

Characteristics

- Dynamic forking and joining
- Granularity of tasks changing
- Load hard to even
- Termination of task

Fork/join parallelism in Java

Java package **java.util.concurrent** includes a library for fork/join parallelism

To implement a method T m() using fork/join parallelism:

If m is a procedure (T is **void**):

- create a class that inherits from RecursiveAction
- override **void** compute() with m's computation

If m is a function:

- create a class that inherits from RecursiveTask<T>
- override T compute() with m's computation

RecursiveAction and RecursiveTask<T> provide methods:

- fork(): schedule for asynchronous parallel execution
- T join(): waits for termination and returns result if T != **void**
- T invoke(): arranges synchronous parallel execution (fork and join) and returns result if T != **void**
- invokeAll(Collection<T> tasks): invoke all tasks in collection (fork all and join all), and return collection of results

Parallel merge sort using fork/join

```java
public class PMergeSort
  extends RecursiveAction {
  // values to be sorted:
  private Integer[] data;
  // to be sorted: data[low..high):
  private int low, high;
```

```java
@Override
  protected void compute() {
    if (high - low <= 1) return; // size<=1: sorted already
    int mid = low + (high - low)/2;  // mid point
    // left and right halves:
    PMergeSort left = new PMergeSort(data, low, mid);
    PMergeSort right = new PMergeSort(data, mid, high);
    left.fork();     // fork thread working on left
    right.fork();    // fork thread working on right
    left.join();     // wait for sorted left half
    right.join();    // wait for sorted right half
    merge(mid);      // merge halves
  }
```

Running a fork/join task

The top computation of a fork/join task is started by a pool object:

```
// to sort array 'numbers' using PMergeSort:
RecursiveAction sorter = new PMergeSort(numbers, 0, numbers.length);
// schedule 'sorter' for execution, and wait for computation to finish:
ForkJoinPool.commonPool().invoke(sorter);
// now 'numbers' is sorted
```

`ForkJoinPool` makes top invocation:

- it launchs a pool object, a synchronous parallel execution of all threads which will fork and join
- it terminates once all the threads join and terminate

The pool takes care of efficiently dispatching work to threads

The framework introduces a layer of abstraction between computational tasks and actual running threads that execute the tasks

This way, the fork/join model simplifies parallelizing computations, since we can focus on how to split data among tasks in a way that avoids race conditions

Fork/join good practices

To take advantage of the number of available cores (in Java):

"In Java, the fork/join framework provides support for parallel programming by splitting up a task into smaller tasks to process them using the available CPU cores.

When you execute ForkJoinPool() it creates an instance with a number of threads equal to the number returned by the method Runtime.getRuntime().availableProcessors(), using defaults for all the other parameters."

(Taken from https://www.pluralsight.com/guides/introduction-to-the-fork-join-framework)

Executor Services

Characteristics

- Large coarse-grained tasks
- Dependencies are simpler
- Meant to stay there until terminated

Executing “things” in parallel in Java

Java package **`java.util.concurrent`** includes a library for ExecutorServices

Do you need to return a value?

If `m` is a procedure:

- Re-use the `Runnable` interface
- override **`void`** `run()` with `m`’s computation

If `m` is a function:

- Implement the `Callable<T>` interface
- override `T call()` with `m`’s computation
- allowed to throw!

`External to Runnable/Callable:`

- `Future<T>`: handle for waiting for termination, cancelling, returned results, and exception handling
- `fork()/join()`: are not over-ridden!
- `submit()/execute()`: using an appropriate service
- `invokeAll(Collection<Y extends Callable<T>> tasks)`: invoke all tasks in collection and wait for them to terminate

Executor Services – implementing Thread pools

Java offers efficient implementations of thread pools in package `java.util.concurrent`

The `interface ExecutorService` provides:

- Schedule `thread` for execution: `void execute(Runnable thread)`:
- Schedule `thread` for execution, and return a `Future` object (to cancel the execution, or wait for termination): `Future submit(Runnable thread)`
  `Future<T> submit(Callable<T> call)`

Implementations of `ExecutorService` with different characteristics can also be obtained by factory methods of `class Executors`:

- `CachedThreadPool`: thread pool of dynamically variable size
- `WorkStealingPool`: thread pool using work stealing
- `ForkJoinPool`: work-stealing pool for running fork/join tasks – **careful w details!**
- …

Thread pools in Java: example

Without thread pools:

```
Counter counter = new Counter();
// threads t and u

Thread t = new Thread(counter);
Thread u = new Thread(counter);
t.start(); // increment once
u.start(); // increment twice
try { // wait for termination
   t.join(); u.join();
}
catch (InterruptedException e)
{
   System.out.println("Int!");
}
```

With thread pools:

```
Counter counter = new Counter();
// threads t and u

Thread t = new Thread(counter);
Thread u = new Thread(counter);
ExecutorService pool = Executors.newWorkStealingPool();

// schedule t and u for execution
Future<?> ft = pool.submit(t);
Future<?> fu = pool.submit(u);
try {
   ft.get(); fu.get();
}
catch (InterruptedException | ExecutionException e){
   System.out.println("Int!");
}
```

we use "?" since we are not interested in the result but use the future just for the sake of cancelling the task

Parallel map vs executor map

```java
import java.util.function.Function;

public class ParallelMap<X,Y> extends Map<X,Y> {

    ParallelMap(X[] source, … ) …

    public class Applicator implements Runnable {
        …
    }

    public void map() {
        Thread[] threads = new Thread[size];

        for (int i=0 ; i<size ; i++) {
            threads[i] = new Thread(new Applicator(i));
            threads[i].start();
        }
        try {
            for (int i=0 ; i<size ; i++) {
                threads[i].join();
            }
        } catch (InterruptedException e) {}
}   }
}
```

How would you implement it?

Parallel map vs executor map

```
import java.util.function.Function;

public class ParallelMap<X,Y> extends Map<X,Y> {

    ParallelMap(X[] source, … ) …

    public class Applicator implements Runnable {
        …
    }

    public void map() {
        Thread[] threads = new Thread[size];

        for (int i=0 ; i<size ; i++) {
            threads[i] = new Thread(new Applicator(i));
            threads[i].start();
        }
        try {
            for (int i=0 ; i<size ; i++) {
                threads[i].join();
            }
        } catch (InterruptedException e) {}
}   }
}
```

```
public class ParallelMapPool<X,Y> extends Map<X,Y> {

    ParallelMapPool(ArrayList<X> source, … ) …

    public class Applicator implements Runnable {
        …
    }

    public void map() {
        ExecutorService pool = Executors.newCachedThreadPool();

        for (int i=0 ; i<source.size() ; i++) {
            pool.execute(new Applicator(i));
        }
        pool.shutdown();
        try {
            pool.awaitTermination(1, TimeUnit.DAYS);
        }
        catch (InterruptedException e) {
}   }   }
```

More executor maps

```java
public class Applicator1 implements Callable<Y> {
    int loc;
    Applicator1(int loc) {
        this.loc = loc;
    }

    public Y call() {
        return func.apply(source.get(loc));
    }
}
```

```java
public void map1() {
    ExecutorService pool = Executors.newCachedThreadPool();
    ArrayList<Future<Y>> futures = new
                                  ArrayList<Future<Y>>(source.size());
    for (int i=0 ; i<source.size() ; i++) {
        futures.set(i,pool.submit(new Applicator1(i)));
    }

    for (int i=0 ; i<target.size() ; i++) {
        try {
        target.set(i,futures.get(i).get());
        }
        catch (InterruptedException e) {
            // Here we end up if this
            // thread was interrupted

            // You might want to wait again
        }
        catch (ExecutionException e) {
            // Here we end up if the
            // execution of the thread had an exception

            // You might want to run it again
        }
    }
    pool.shutdownNow();
}
```

More executor maps

```java
public class Applicator2 implements Callable<Y> {
    X val;
    Applicator2(X val) {
        this.val = val;
    }

    public Y call() {
        return func.apply(this.val);
    }
}
```

```java
public void map2() {
    ExecutorService pool = Executors.newCachedThreadPool();
    ArrayList<Future<Y>> futures = new
ArrayList<Future<Y>>(source.size());
    for (int i=0 ; i<source.size() ; i++) {
        futures.set(i,pool.submit(new Applicator2(source.get(i))));
    }

    for (int i=0 ; i<source.size() ; i++) {
        try {
        target.set(i,futures.get(i).get());
        }
        catch (InterruptedException e) {
            // Here we end up if this
            // thread was interrupted

            // You might want to wait again
        }
        catch (ExecutionException e) {
            // Here we end up if the
            // execution of the thread had an exception

            // You might want to run it again
        }
    }
    pool.shutdownNow();
}
```

More about the Future …

"A Future represents the result of an asynchronous computation.

Methods are provided to check if the computation is complete, to wait for its completion, and to retrieve the result of the computation.

The result can only be retrieved using method get when the computation has completed, blocking if necessary until it is ready.
Cancellation is performed by the cancel method.
Additional methods are provided to determine if the task completed normally or was cancelled.
Once a computation has completed, the computation cannot be cancelled.
If you would like to use a Future for the sake of cancellability but not provide a usable result, you can declare types of the form Future<?> and return null as a result of the underlying task."

From the Java documentation about "public interface Future<V>"

Process pools in Erlang

Erlang provides some load distribution services in the system module `pool`

These are aimed at distributing the load between different nodes, each a full-fledged collection of processes

Parallel map with workers

We can define a parallel version of `map` using a pool:

map: apply function F to all elements in list L (independently)

```
pmap(F, L, N) -> init_pool( F,  % function to be mapped
                            L,  % workload: list to be mapped
                            fun ([H|T]) -> {H,T} end,    % split: take first element
                            fun (R,Res) -> [R|Res] end, % join: cons with list
                            [],  % initial value
                            N    % number of workers
                          ).
```

Note that the order of the results may change from run to run

It is possible to restore the original order by using a more complex join function