CHALMERS

UNIVERSITY OF
GOTHENBURG

Parallelizing computations

Lecture 9 of TDA384/DIT391
Principles of Concurrent Programming

Sandro Stucki

Chalmers University of Technology | University of Gothenburg

SP3 2018/2019

Based on course slides by Carlo A. Furia

Today's menu

Challenges to parallelization

Fork/join parallelism

Pools and work stealing

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.

> *My concurrent program will be so fast, there will be no time to check the answer!*

– Scott West, circa 2010

General approaches to parallelization

In this class, we explore several general approaches to parallelizing computations in multi-processor systems.

A task $\langle F, D\rangle$ consists in computing the result $F(D)$ of applying function F to input data D.
A parallelization of $\langle F, D\rangle$ is a collection $\langle F_1, D_1\rangle, \langle F_2, D_2\rangle, \ldots$ of tasks such that $F(D)$ equals the composition of $F_1(D_1), F_2(D_2), \ldots$.

We first cast the problems and solutions using Erlang's notation and models – message-passing between processes – since it is easier to prototype implementations of the solutions.

Then, we will apply the same concepts and techniques to shared-memory models such as Java threads.

Challenges to parallelization

Challenges to parallelization

A strategy to parallelize a task $\langle F, D \rangle$ 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 $\langle F, D \rangle$ 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 $((1 + 2) + (3 + 4)) + ((5 + 6) + (7 + 8))$

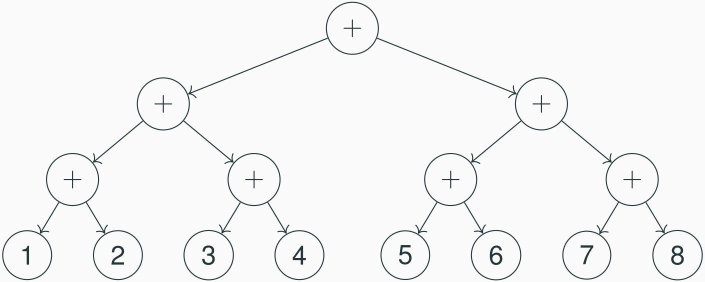
The computations in each group depend on the computations in the previous group, and hence the corresponding tasks must execute after the latter have completed.

The synchronization problems (producer-consumer, dining philosophers, etc.) we discussed in various classes capture kinds of sequential dependencies that may occur when parallelizing.

Dependency graph

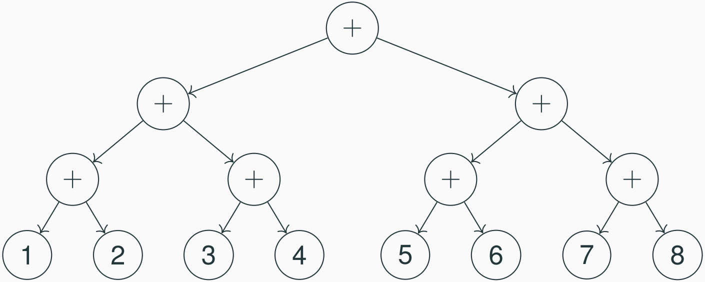
Some steps in a task computation depends on the result of other steps; this creates sequential dependencies where one task must wait for another task to run.

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.



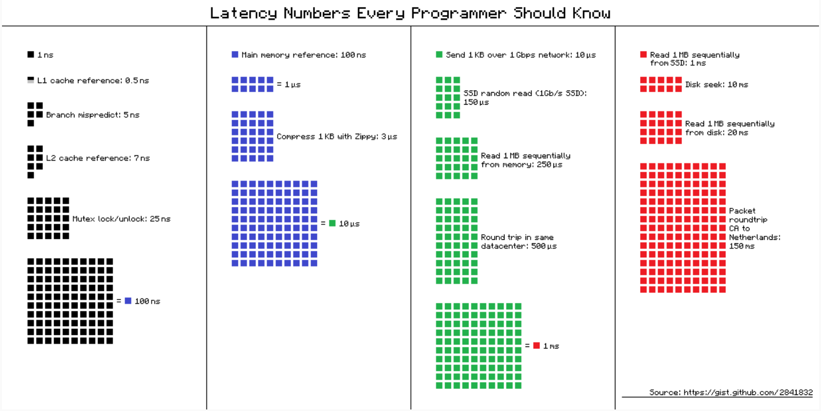
Dependency graph

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.



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



Source: https://gist.github.com/2841832

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 diminishing returns with more spawning.

Error proneness and composability

Synchronization is prone to errors such as data races, deadlocks, and starvation. Message-based synchronization may improve the situation, but it is far for being straightforward and problem free.

Error proneness and composability

Synchronization is prone to errors such as data races, deadlocks, and starvation. Message-based synchronization may improve the situation, but it is far for being straightforward and problem free.

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; }
}
```

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; }
}
```

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

execute atomically

Method `transfer` does not execute atomically: 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

Even if there is no locking with message passing, we still 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

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 class, we present:

- fork/join parallelism techniques, which help naturally capture sequential dependencies
- pools, which help curb the spawning costs

In future classes we will address the remaining problems of reducing synchronization costs and achieving composability.

Fork/join parallelism

Parallel servers

A server’s event loop offers clear opportunities for parallelism:

- each request sent to the server is independent of the others
- instead of serving requests sequentially, a server spawns a new process for every request
- a child processes computes, sends response to the client, and terminates

```
loop(State, Operation) ->
 receive
   {request, From, Ref, Data} ->
     From ! {reply, Ref,
             Operation(Data)},
     loop(new_state(State));
   % other operations...
 end.
```

```
ploop(State, Operation) ->
 receive
   {request, From, Ref, Data} ->
    spawn(fun ()->
     Result = Operation(Data),
     From ! {reply, Ref, Result}
          end),
     loop(new_state(State));
   % other operations...
 end.
```

Parallel recursion

The structure of recursive functions lends itself to parallelization according to the structure of recursion.

Recursion is easier to parallelize when it is expressed in a mostly side-effect free language like sequential Erlang:

- spawn a process for every recursive call
- no side effects means no hidden dependencies – a process's results only depends on its explicit input

Parallel recursion: merge sort

```erlang
merge_sort(List)
  when length(List) =< 1 ->
    List;
merge_sort(List) ->
  Mid = length(List) div 2,
    % split in two halves
  {L, R} = lists:split(Mid, List),
    % recursively sort each half
  SL = merge_sort(L),
  SR = merge_sort(R),
    % merge sorted halves
  merge(SL, SR).
```

Parallel recursion: merge sort

```
merge_sort(List)
  when length(List) =< 1 ->
    List;
merge_sort(List) ->
  Mid = length(List) div 2,
    % split in two halves
  {L, R} = lists:split(Mid, List),
    % recursively sort each half
  SL = merge_sort(L),
  SR = merge_sort(R),
    % merge sorted halves
  merge(SL, SR).
```

```
pmerge_sort(List)
  when length(List) =< 1 ->
    List;
pmerge_sort(List) ->
  Mid = length(List) div 2,
  {L, R} = lists:split(Mid, List),
  Pid = self(),
  spawn(fun ()-> Pid !
     {sl, pmerge_sort(L)} end),
  spawn(fun ()-> Pid !
     {sr, pmerge_sort(R)} end),
  receive {sl, SL} -> sl end,
  receive {sr, SR} -> sr end,
  merge(SL, SR).
```

Parallel recursion: merge sort

```
merge_sort(List)
  when length(List) =< 1 ->
    List;
merge_sort(List) ->
  Mid = length(List) div 2,
    % split in two halves
  {L, R} = lists:split(Mid, List),
    % recursively sort each half
  SL = merge_sort(L),
  SR = merge_sort(R),
    % merge sorted halves
  merge(SL, SR).
```

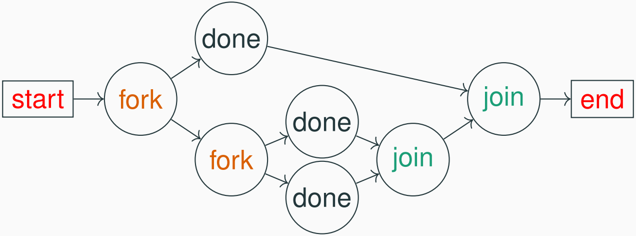
```
pmerge_sort(List)
  when length(List) =< 1 ->
    List;
pmerge_sort(List) ->
  Mid = length(List) div 2,
  {L, R} = lists:split(Mid, List),
  Pid = self(),
  spawn(fun ()-> Pid !
     {sl, pmerge_sort(L)} end),
  spawn(fun ()-> Pid !
     {sr, pmerge_sort(R)} end),
  receive {sl, SL} -> sl end,
  receive {sr, SR} -> sr end,
  merge(SL, SR).
```

cannot be computed inside closure in spawn: must be the parent's pid

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.

Parallel map

Function map's recursive structure lends itself to parallelization.

```
% apply F to all
% elements of list
map(_, [])    -> [];
map(F, [H|T]) ->
  [F(H)|map(F,T)].
```

Parallel map

Function `map`'s recursive structure lends itself to parallelization.

```
% apply F to all
% elements of list
map(_, [])    -> [];
map(F, [H|T]) ->
  [F(H)|map(F,T)].

% wait for all Children
% and collect results in order
gather(Children, Ref) ->
  [receive {Child, Ref, Res}
           -> Res end
     || Child <- Children].
```

```
% parallel map
pmap(F, L) ->
  Me = self(), % my pid
  Ref = make_ref(),
    % for every E in L:
  Children = map(fun(E) ->
      % spawn a process
    spawn(fun() ->
      % sending Me result of F(E)
       Me ! {self(), Ref, F(E)}
          end) end, L),
    % collect and return results
  gather(Children, Ref).
```

Parallel map

Function `map`'s recursive structure lends itself to parallelization.

```
% apply F to all
% elements of list
map(_, [])    -> [];
map(F, [H|T]) ->
  [F(H)|map(F,T)].


% wait for all Children
% and collect results in order
gather(Children, Ref) ->
  [receive {Child, Ref, Res}
           -> Res end
     || Child <- Children].
```

```
% parallel map
pmap(F, L) ->
  Me = self(), % my pid
  Ref = make_ref(),
    % for every E in L:
  Children = map(fun(E) ->
      % spawn a process
    spawn(fun() ->
      % sending Me result of F(E)
       Me ! {self(), Ref, F(E)}
          end) end, L),
    % collect and return results
  gather(Children, Ref).
```

list comprehension ensures results are collected in order

Parallel reduce

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

```
reduce(_, A, []) -> A;
reduce(F, A, [H|T]) ->
  F(H, reduce(F, A, T)).
```

Parallel reduce

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

```
reduce(_, A, []) -> A;
reduce(F, A, [H|T]) ->
  F(H, reduce(F, A, T)).
```

```
preduce(_, A, [])   -> A;
preduce(F, A, [E])  -> F(A, E);
preduce(F, A, List) ->
  Mid = length(List) div 2,
  {L, R} = lists:split(Mid, List),
  Me = self(), % L ++ R =:= Listn
  Lp = spawn(fun() -> % on left half
   Me ! {self(), preduce(F, A, L)} end),
  Rp = spawn(fun() -> % on right half
   Me ! {self(), preduce(F, A, R)} end),
  % combine results of left, right half
  F(receive {Lp, Lr} -> Lr end,
    receive {Rp, Rr} -> Rr end).
```

Parallel reduce

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

```
reduce(_, A, []) -> A;
reduce(F, A, [H|T]) ->
  F(H, reduce(F, A, T)).
```

`preduce(F, A, L)` equals `reduce(F, A, L)` if:

- `F` is associative (`preduce` does not apply `F` right-to-left)
- for every list element `E`: `F(E, A)` = `F(A, E)` = `E` (`preduce` reduces `A` in every base case, not just once)

```
preduce(_, A, [])   -> A;
preduce(F, A, [E])  -> F(A, E);
preduce(F, A, List) ->
  Mid = length(List) div 2,
  {L, R} = lists:split(Mid, List),
  Me = self(), % L ++ R =:= Listn
  Lp = spawn(fun() -> % on left half
   Me ! {self(), preduce(F, A, L)} end),
  Rp = spawn(fun() -> % on right half
   Me ! {self(), preduce(F, A, R)} end),
  % combine results of left, right half
  F(receive {Lp, Lr} -> Lr end,
    receive {Rp, Rr} -> Rr end).
```

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

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()`: wait for termination, and return result if `T != void`
- `T invoke()`: arrange synchronous parallel execution (fork and join), and return 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 {
    private Integer[] data;  // values to be sorted
    private int low, high;   // to be sorted: data[low..high)

    @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
```

The pool takes care of efficiently dispatching work to threads, as we describe in the rest of this class.

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.

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 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
}
```

the forking thread is idle!

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 even need to warm up before it optimizes the parallel code more aggressively.)

Revisited parallel merge sort using fork/join

choose experimentally (at least 1000)

```
protected void compute() {
  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.compute(); // continue work on right in same task
    left.join();     // when done, wait for sorted left half
    merge(mid);      // merge halves
  }
}
```

before joining, do more work in current task

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.

How many processes is lagom?

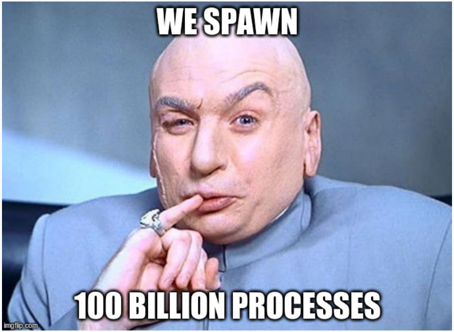
Parallelizing by following the recursive structure of a task is simple and appealing. However, the potential performance gains should be weighted against the overhead of creating and running many processes.

Process creation in Erlang is lightweight: 1 GiB 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 do not improve performance if their number greatly exceeds the number of available physical processors.



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 resources that are available 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 because it actively deals work to workers.

Workers

Workers are servers 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

As soon as a worker completes a work assignments, it sends the result to the pool and goes back to being idle.

```
% create worker for ‘Pool’ computing ‘Function’
init_worker(Pool, Function) ->
  spawn(fun ()-> worker(Pool, Function) end).

worker(Pool, Function) ->
  receive {Pool, Data} ->     % assignment from pool
    Result = Function(Data),  % compute work
    Pool ! {self(), Result},  % send result to pool
    worker(Pool, Function)    % back to idle
  end.
```

Pool state

A pool keeps track of:

- the remaining work – not assigned yet
- the busy workers
- the idle workers

```
-record(pool, {work, busy, idle}).
```

The pool also stores:

- a split function, used to extract a single work item
- a join function, used to combine partial results
- the overall result of the computation that is underway

```
pool(Pool#pool, Split, Join, Result) -> todo.
```

state of record type `pool`

Pool termination

The pool terminates and returns the result of the computation when there are no pending work items, and all workers are busy (thus all work has been done).

```
% work completed, no busy workers: return result
pool(#pool{work = [], busy = []},
     _Split, _Join, Result) ->
  Result;
```

Dealing work

As long as there is some pending work and some idle workers, the pool deals work to some of those idle workers.

```
% work pending, some idle workers: assign work
pool(Pool = #pool{work = Work = [_|_], % matches if Work not empty
                  busy = Busy,
                  idle = [Worker|Idle]},
     Split, Join, Result) ->
    {Chunk, Remaining} = Split(Work),  % split pending work
    Worker ! {self(), Chunk},          % send chunk to worker
    pool(Pool#pool{work = Remaining,
                   busy = [Worker|Busy],
                   idle = Idle},
         Split, Join, Result);
```

Using a function `Split` provides flexibility in splitting work into chunks.

Collecting results

When there are no pending work items or all workers are busy, the pool can only wait for workers to send back results.

```
% work completed or no idle workers: wait for results
pool(Pool = #pool{busy = Busy, idle = Idle},
     Split, Join, Result) ->
         % get result from worker
    receive {Worker, PartialResult} -> ok end,
         % join worker’s result and current result
    NewResult = Join(PartialResult, Result),
    pool(Pool#pool{busy = lists:delete(Worker, Busy),
                   idle = Idle ++ [Worker]},
         Split, Join, NewResult).
```

Note that the condition “no pending work or all workers busy” is implicit because this clause comes last in the definition of `pool`.

Pool creation

Initializing a pool requires a function to be computed, a workload, split and join functions, and a number of worker threads.

```
init_pool(Function, Work, Split, Join, Initial, N) ->
  Pool = self(),
     % spawn N workers for the same pool
  Workers = [init_worker(Pool, Function) || _ <- lists:seq(1, N)],
  [link(W) || W <- Workers], % link workers to pool
     % initially all work is pending, all workers are idle
  pool(#pool{work = Work, busy = [], idle = Workers},
       Split, Join, Initial).
```

The function `link` ensures that the worker processes are terminated as soon as the process running the pool does.

In practice we would set `N` to an optimal number based on the available resources, and export `init_pool` working with that number.

Parallel map with workers

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

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

Parallel map with workers

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

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

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.

Parallel reduce with workers

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

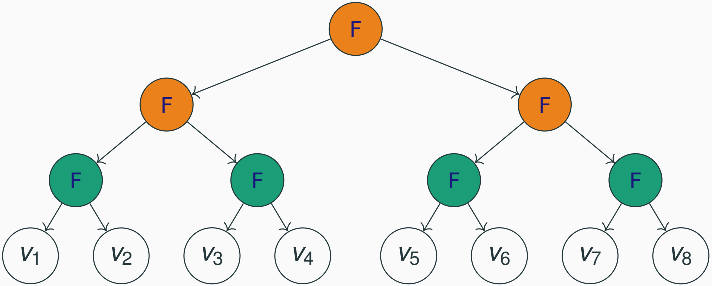
```
preduce(F, I, L, N) ->
  init_pool(fun ({X,Y}) -> F(X,Y) end, % so that a chunk is a pair
            L,   % split: take first two elements
            fun (W) -> chunk_two(I, W) end,
            F,   % join: folding function!
            I, N).

chunk_two(_, [F|[S|R]]) -> {{F,S}, R};
chunk_two(I, [F|R]) -> {{F,I}, R}.
```

This works correctly under the same conditions as the direct recursive version of `preduce` shown before: `F` should be associative, and `I` should be a neutral element under `F`.

The syntax is a bit clunky, but the basic idea is that `preduce` assigns to each worker the reduction of two consecutive input elements.

Joining is working too

In our version of `preduce` using a dealing pool, a lot of reduction work is actually done by the pool process when executing `join` for each result. In the dependency graph, the bottom level is computed by the workers; the upper levels are computed by the pool while joining.



Recursive dealing pools

More generally, the dealing process pool we have designed works well if joining is a lightweight operation compared to computing the work function.

A more flexible solution subdivides work in tasks. Each task consists of a function to be applied to a list of data.

```
-record(task, {function, data}).
```

- The split function extracts a smaller task from a bigger one
- The join function creates a task consisting of computing the join

With this approach, the pool can delegate joining to the workers or do it directly if it is little work. By creating suitable join and split functions we can make a better usage of workers and achieve a better parallelization.

We call this kind of pool recursive (dealing) pool, because it may recursively generate new work while combining intermediate results.

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

This is an outline of the algorithm for work stealing. It assumes that 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();
    } } }
```

Thread pools in Java

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

The **`interface ExecutorService`** provides:

- `void execute(Runnable thread)`: schedule `thread` for execution
- `Future submit(Runnable thread)`: schedule `thread` for execution, and return a `Future` object (to cancel the execution, or wait for termination)

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

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 { // wait for termination
  ft.get(); fu.get(); }
catch (InterruptedException
       | ExecutionException e)
{ System.out.println("Int!"); }
```

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.

These slides’ license