Parallelizing computations

Lecture 10 of TDA383/DIT390 (Concurrent Programming)

Carlo A. Furia

Chalmers University of Technology – University of Gothenburg

SP3 2016/2017

Today's menu

Challenges to parallelization

Fork/join parallelism

Pools and work stealing

Software transactional memory

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 $F(D)$ is the composition of $F_1(D_1), F_2(D_2), \ldots$.

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

However, most of the concepts and techniques apply as well 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 $((1 + 2) + (3 + 4)) + ((5 + 6) + (7 + 8))$

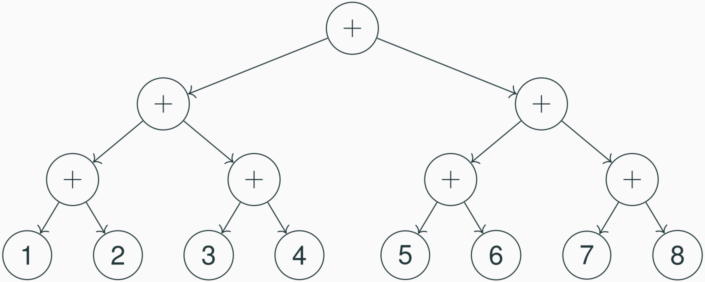
The computations in each group depend on the computations in the previous group, and hence the corresponding tasks must execute after the latter have completed.

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

Dependency graph

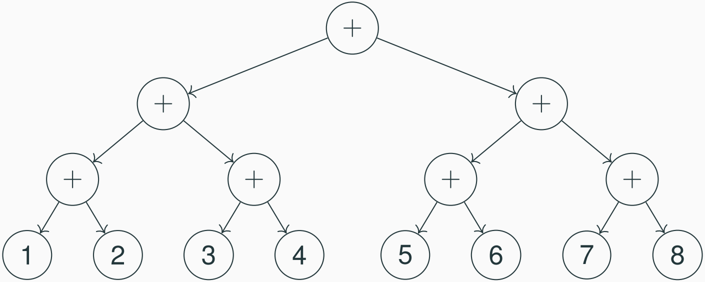
Some steps in a task computation depends on the result of other steps; this creates sequential dependencies where one task must wait for another task to run.

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 overheads and assuming all processes can run in parallel).

Synchronization costs

Synchronization is required to preserve correctness, but it also introduces overheads 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 in 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; }
}
```

execute atomically

```
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 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 amount would look lower than it really 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 simple possible solutions 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.

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

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

```erlang
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

```
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).
```

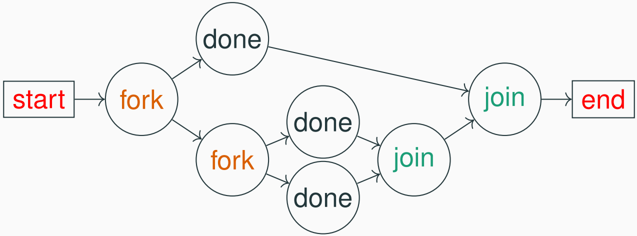
```
pmerge_sort(List)
  when length(List) =< 1 ->
    List;
pmerge_sort(List) ->
  Mid = length(List) div 2,
  {L, R} = lists:split(Mid, List),
  Pid = self(),
  spawn(fun ()-> Pid !
     {sl, pmerge_sort(L)} end),
  spawn(fun ()-> Pid !
     {sr, pmerge_sort(R)} end),
  receive {sl, SL} -> sl end,
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

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

```
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 (low >= high - 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
   }
```

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)).
```

`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

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?

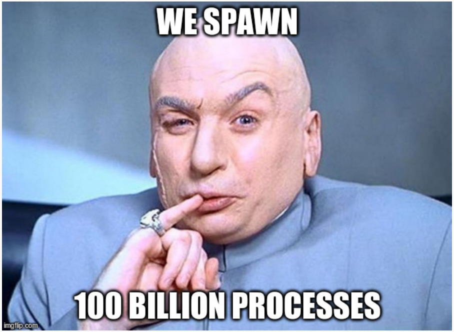
Parallelizing by following the recursive structure of a task is simple and appealing. However, the potential performance gains should be weighted against the overhead of creating and running many processes.

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.

```erlang
% 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).
```

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

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).
```

In practice we would set `N` to an optimal number based on the available resources, and just export a parallel variant of `map`.

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:

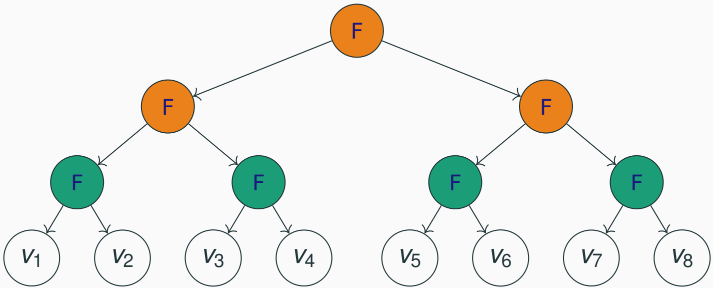
```
preduce(F, I, L, N) ->
  init_pool(fun ({X,Y}) -> F(X,Y) end, % so that a chunk is a pair
            L,   % split: take first two elements
            fun (W) -> chunk_two(I, W) end,
            F,   % join: folding function!
            I, N).
```

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 cumbersome, 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 it 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 offloads new tasks by adding them a worker's queue.

When a worker becomes idle:

- first, it gets the next task from the its 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.

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.

In Lab 4 – Workers, you will implement a simple dealing worker pool following the ideas we have presented in this class.

Software transactional memory

The trouble with locks

Standard techniques for concurrent programming are ultimately based on locks. Programming with locks has several drawbacks:

- performance overheads
- lock granularity is hard to choose:
  - not enough locking: race conditions
  - too much locking: not enough parallelism
- risk of deadlock and starvation
- lock-based implementations do not compose
- lock-based programs are hard to maintain and modify

Message-passing programming is somewhat higher-level, but it still incurs some of the synchronization costs and lack of composability associated with locks.

Breaking free of locks

Lock-free programming takes a fresh look at the problems of concurrency and tries to dispense with using locks altogether:

- lock-based programming is pessimistic: be prepared for the worst possible conditions:

  if things can go wrong, they will

- lock-free programming is optimistic: do what you have to do without worrying about race conditions

  if things go wrong, just try again

Lock-free programming

Lock-free programming relies on:

- using stronger primitives for atomic access
- building optimistic algorithms using those primitives

Compare-and-set operations are an example of stronger primitives:

```
public class AtomicInteger {
  // atomically set to ‘update’ if current value is ‘expect’
  // otherwise do not change value and return false
  boolean compareAndSet(int expect, int update)
}
```

To update an AtomicInteger variable `k`:

```
  do {  // keep trying until no one changes k in between
    int oldValue = k.get();
    int newValue = compute(oldValue);
  } while (!k.compareAndSet(oldValue, newValue));
```

Lock-free vs. wait-free

Two classes of lock-free algorithms, collectively called non-blocking:

lock-free: guarantee system-wide progress: infinitely often, some process makes progress

wait-free: guarantee per-process progress: every process eventually makes progress

Wait-free is stronger than lock-free:

- Lock-free algorithms are free from deadlock
- Wait-free algorithms are free from deadlock and starvation

Lock-free and wait-free algorithms have been developed for a number of problems — in particular, non-blocking data structures atomically accessible in parallel (thread safe), such as those in `java.util.concurrent`.

Transactions

The notion of transaction, which comes from database research, supports a general approach to lock-free programming:

> A transaction is a sequence of steps executed by a single thread, which are executed atomically.

A transaction may:

- succeed: all changes made by the transaction are committed to shared memory; they appear as if they happened instantaneously
- fail: the partial changes are rolled back, and the shared memory is in the same state it would be if the transaction had never executed

Therefore, a transaction either executes completely and successfully, or it does not have any effect at all.

Programming with transactions

The notion of transaction supports a general approach to lock-free programming:

- define a transaction for every access to shared memory
- if the transaction succeeds, there was no interference
- if the transaction failed, retry until it succeeds

Imagine we have a syntactic means of defining transaction code:

```
atomic {
  // transaction code
}
// retry until success
```

```
% execute Function(Arguments)
% as a transaction (retry until success)
atomic(Function, Arguments)
```

Transactions may also support invoking retry and rollback explicitly.

(Note that `atomic` is not a valid keyword in Java or Erlang: we use it for illustration purposes, and later we sketch how it could be implemented as a function in Erlang.)

Transactions are better than locks

Transactional atomic blocks look superficially similar to monitor's methods with implicit locking, but they are in fact much more flexible:

- since transactions do not lock, there is no locking overhead
- parallelism is achieved without risks of race conditions
- since no locks are acquired, there is no problem of deadlocks (although starvation may still occur if there is a lot of contention)
- transactions compose easily

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

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

no locking, so no deadlock is possible!

Transactional memory

A transactional memory is a shared memory storage that supports atomic updates of multiple memory locations.

Implementations of transactional memory can be based on hardware or software:

- hardware transactional memory relies on support at the level of instruction sets (Herlihy & Moss, 1993)
- software transactional memory is implemented as a library or language extension (Shavit & Touitou, 1995)

Software transactional memory implementations are available for several mainstream languages (including Java, Haskell, and Erlang). This is still an active research topic — quality varies!

Implementing software transactional memory

We outline an implementation of software transactional memory (STM) in Erlang.

Each variable in an STM is identified by a name, value, and version:

```
-record(var, {name, version = 0, value = undefined}).
```

Implementing software transactional memory

We outline an implementation of software transactional memory (STM) in Erlang.

Each variable in an STM is identified by a name, value, and version:

```
-record(var, {name, version = 0, value = undefined}).
```

Clients use an STM as follows:

- at the beginning of a transaction, check out a copy of all variables involved in the transaction
- execute the transaction, which modifies the values of the local copies of the variables
- at the end of a transaction, try to commit all local copies of the variables

Implementing software transactional memory

We outline an implementation of software transactional memory (STM) in Erlang.

Each variable in an STM is identified by a name, value, and version:

```
-record(var, {name, version = 0, value = undefined}).
```

The STM's commit operation ensures atomicity:

- if all committed variables have the same version number as the corresponding variables in the STM, there were no changes to the memory during the transaction: the transaction succeeds
- if some committed variable has a different version number from the corresponding variable in the STM, there was some change to the memory during the transaction: the transaction fails

The counter example — with software transactional memory

```
int cnt;
```

thread t	thread u
`int c;`	`int c;`
`atomic {`	`atomic {`
`c = cnt;`	`c = cnt;`
`cnt = c + 1;`	`cnt = c + 1;`
`}`	`}`

The `atomic` translates into a loop that repeats until the transaction succeeds:

1. check out (`pull`) the current value of `cnt`
2. increment the local variable `c`
3. try to commit (`push`) the new value of `cnt`
4. if `cnt` has changed version when trying to commit, repeat the loop

The counter example: a successful run

⟨name: cnt, version: x, value: y⟩

thread t	thread u
`int c;`	`int c;`
`do {`	`do {`
`// check out cnt`	`// check out cnt`
● `c = pull(cnt);`	`c = pull(cnt);` ●
`c = c + 1;`	`c = c + 1;`
`} while (!push(cnt, c));`	`} while (!push(cnt, c));`
`// commit cnt`	`// commit cnt`

The subscript in a variable's value indicates its version:

t's LOCAL	u's LOCAL	STM
$c_t : \bot$	$c_u : \bot$	cnt : 0_3

The counter example: a successful run

⟨name: cnt, version: x, value: y⟩

thread t

```
int c;
do {
 // check out cnt
 c = pull(cnt);
•c = c + 1;
} while (!push(cnt, c));
  // commit cnt
```

thread u

```
int c;
do {
 // check out cnt
 c = pull(cnt); •
 c = c + 1;
} while (!push(cnt, c));
  // commit cnt
```

The subscript in a variable's value indicates its version:

t's LOCAL	u's LOCAL	STM
$c_t : 0_3$	$c_u : \bot$	cnt : 0_3

The counter example: a successful run

$\langle$`name: cnt`, `version:`x, `value:`$y\rangle$

thread t	thread u
`int c;`	`int c;`
`do {`	`do {`
`  // check out cnt`	`  // check out cnt`
`  c = pull(cnt);`	`  c = pull(cnt);` ●
`  c = c + 1;`	`  c = c + 1;`
● `} while (!push(cnt, c));`	`} while (!push(cnt, c));`
`    // commit cnt`	`    // commit cnt`

The subscript in a variable's value indicates its version:

t'S LOCAL	u'S LOCAL	STM
$c_t : 1_3$	$c_u : \bot$	$cnt : 0_3$

The counter example: a successful run

⟨name: cnt, version: x, value: y⟩

thread t	thread u

`int c;`
`do {`
` // check out cnt`
` c = pull(cnt);`
` c = c + 1;`
`} while (!push(cnt, c));`
`  // commit cnt`
	`int c;`
	`do {`
	` // check out cnt`
	` c = pull(cnt); ●`
	` c = c + 1;`
	`} while (!push(cnt, c));`
	`  // commit cnt`

The subscript in a variable's value indicates its version:

t'S LOCAL	u'S LOCAL	STM
success	c_u : ⊥	cnt : 1_4

The counter example: a successful run

⟨`name: cnt`, `version:` x, `value:` y⟩

thread t	thread u
`int c;`	`int c;`
`do {`	`do {`
`// check out cnt`	`// check out cnt`
`c = pull(cnt);`	`c = pull(cnt);`
`c = c + 1;`	`c = c + 1;` ●
`} while (!push(cnt, c));`	`} while (!push(cnt, c));`
`// commit cnt`	`// commit cnt`

The subscript in a variable's value indicates its version:

t'S LOCAL	u'S LOCAL	STM
done	$c_u : 1_4$	$cnt : 1_4$

The counter example: a successful run

$\langle$`name: cnt`, `version:`x, `value:`$y\rangle$

thread t	thread u
`int c;`	`int c;`
`do {`	`do {`
` // check out cnt`	` // check out cnt`
` c = pull(cnt);`	` c = pull(cnt);`
` c = c + 1;`	` c = c + 1;`
`} while (!push(cnt, c));`	`} while (!push(cnt, c));` ●
`  // commit cnt`	`  // commit cnt`

The subscript in a variable's value indicates its version:

t's LOCAL	u's LOCAL	STM
done	$c_u : 2_4$	$cnt : 1_4$

The counter example: a successful run

⟨`name: cnt`, `version:`x, `value:`y⟩

thread t	thread u
`int c;`	`int c;`
`do {`	`do {`
` // check out cnt`	` // check out cnt`
` c = pull(cnt);`	` c = pull(cnt);`
` c = c + 1;`	` c = c + 1;`
`} while (!push(cnt, c));`	`} while (!push(cnt, c));`
`  // commit cnt`	`  // commit cnt`

The subscript in a variable's value indicates its version:

t'S LOCAL	u'S LOCAL	STM
done	success	cnt: 2_5

The counter example: a retry run

⟨name: cnt, version: x, value: y⟩

thread t	thread u
`int c;`	`int c;`
`do {`	`do {`
`  // check out cnt`	`  // check out cnt`
• `c = pull(cnt);`	`c = pull(cnt);` •
`  c = c + 1;`	`  c = c + 1;`
`} while (!push(cnt, c));`	`} while (!push(cnt, c));`
`  // commit cnt`	`  // commit cnt`

The subscript in a variable's value indicates its version:

t's LOCAL	u's LOCAL	STM
$c_t : \bot$	$c_u : \bot$	cnt : 0_3

The counter example: a retry run

⟨name: cnt, version: x, value: y⟩

thread t	thread u
`int c;`	`int c;`
`do {`	`do {`
`  // check out cnt`	`  // check out cnt`
`  c = pull(cnt);`	`  c = pull(cnt); ●`
`● c = c + 1;`	`  c = c + 1;`
`} while (!push(cnt, c));`	`} while (!push(cnt, c));`
`   // commit cnt`	`   // commit cnt`

The subscript in a variable's value indicates its version:

t'S LOCAL	u'S LOCAL	STM
$c_t : 0_3$	$c_u : \bot$	$cnt : 0_3$

The counter example: a retry run

⟨`name: cnt`, `version:`x, `value:`y⟩

thread t	thread u
`int c;`	`int c;`
`do {`	`do {`
`// check out cnt`	`// check out cnt`
`c = pull(cnt);`	`c = pull(cnt);`
● `c = c + 1;`	`c = c + 1;` ●
`} while (!push(cnt, c));`	`} while (!push(cnt, c));`
`// commit cnt`	`// commit cnt`

The subscript in a variable's value indicates its version:

t's LOCAL	u's LOCAL	STM
$c_t : 0_3$	$c_u : 0_3$	cnt : 0_3

The counter example: a retry run

$\langle$`name: cnt`, `version:`x, `value:`$y\rangle$

thread t	thread u
`int c;`	`int c;`
`do {`	`do {`
`// check out cnt`	`// check out cnt`
`c = pull(cnt);`	`c = pull(cnt);`
`c = c + 1;`	`c = c + 1;` ●
● `} while (!push(cnt, c));`	`} while (!push(cnt, c));`
`// commit cnt`	`// commit cnt`

The subscript in a variable's value indicates its version:

t'S LOCAL	u'S LOCAL	STM
$c_t : 1_3$	$c_u : 0_3$	$cnt : 0_3$

The counter example: a retry run

⟨`name: cnt`, `version:`x, `value:`y⟩

thread t	thread u

thread t

```
int c;
do {
 // check out cnt
 c = pull(cnt);
 c = c + 1;
●} while (!push(cnt, c));
  // commit cnt
```

thread u

```
int c;
do {
 // check out cnt
 c = pull(cnt);
 c = c + 1;
} while (!push(cnt, c)); ●
  // commit cnt
```

The subscript in a variable's value indicates its version:

t'S LOCAL	u'S LOCAL	STM
$c_t : 1_3$	$c_u : 1_3$	$cnt : 0_3$

The counter example: a retry run

⟨name: cnt, version: x, value: y⟩

thread t	thread u

`int c;`
`do {`
` // check out cnt`
` c = pull(cnt);`
` c = c + 1;`
`} while (!push(cnt, c));`
`  // commit cnt`
	`int c;`
	`do {`
	` // check out cnt`
	` c = pull(cnt);`
	` c = c + 1;`
	`} while (!push(cnt, c));` ●
	`  // commit cnt`

The subscript in a variable's value indicates its version:

t'S LOCAL	u'S LOCAL	STM
success	$c_u : 1_3$	cnt : 1_4

The counter example: a retry run

⟨name: cnt, version: x, value: y⟩

thread t	thread u

`int c;`
`do {`
` // check out cnt`
` c = pull(cnt);`
` c = c + 1;`
`} while (!push(cnt, c));`
`  // commit cnt`
	`int c;`
	`do {`
	` // check out cnt`
	` c = pull(cnt);`
	` c = c + 1;`
	`} while (!push(cnt, c));`
	`  // commit cnt`

The subscript in a variable's value indicates its version:

t'S LOCAL	u'S LOCAL	STM
done	fail	cnt: 1_4

The counter example: a retry run

⟨`name: cnt`, `version:`x, `value:`y⟩

thread t	thread u
`int c;`	`int c;`
`do {`	`do {`
`// check out cnt`	`// check out cnt`
`c = pull(cnt);`	`c = pull(cnt);` ●
`c = c + 1;`	`c = c + 1;`
`} while (!push(cnt, c));`	`} while (!push(cnt, c));`
`// commit cnt`	`// commit cnt`

The subscript in a variable's value indicates its version:

t's LOCAL	u's LOCAL	STM
done	retry	`cnt`: 1_4

The counter example: a retry run

⟨name: cnt, version: x, value: y⟩

thread t	thread u

`int c;`
`do {`
` // check out cnt`
` c = pull(cnt);`
` c = c + 1;`
`} while (!push(cnt, c));`
`  // commit cnt`
	`int c;`
	`do {`
	` // check out cnt`
	` c = pull(cnt); ●`
	` c = c + 1;`
	`} while (!push(cnt, c));`
	`  // commit cnt`

The subscript in a variable's value indicates its version:

t'S LOCAL	u'S LOCAL	STM
done	$c_u : \bot$	$\texttt{cnt} : 1_4$

The counter example: a retry run

⟨`name: cnt`, `version:`x, `value:`y⟩

thread t	thread u
`int c;`	`int c;`
`do {`	`do {`
`// check out cnt`	`// check out cnt`
`c = pull(cnt);`	`c = pull(cnt);`
`c = c + 1;`	`c = c + 1;` ●
`} while (!push(cnt, c));`	`} while (!push(cnt, c));`
`// commit cnt`	`// commit cnt`

The subscript in a variable's value indicates its version:

t'S LOCAL	u'S LOCAL	STM
done	$c_u : 1_4$	$cnt : 1_4$

The counter example: a retry run

⟨`name: cnt`, `version:` x, `value:` y⟩

thread t	thread u
`int c;`	`int c;`
`do {`	`do {`
`// check out cnt`	`// check out cnt`
`c = pull(cnt);`	`c = pull(cnt);`
`c = c + 1;`	`c = c + 1;`
`} while (!push(cnt, c));`	`} while (!push(cnt, c));` ●
`// commit cnt`	`// commit cnt`

The subscript in a variable's value indicates its version:

t'S LOCAL	u'S LOCAL	STM
done	$c_u: 2_4$	$cnt: 1_4$

The counter example: a retry run

⟨name: cnt, version: x, value: y⟩

thread t	thread u

`int c;`
`do {`
` // check out cnt`
` c = pull(cnt);`
` c = c + 1;`
`} while (!push(cnt, c));`
`  // commit cnt`
	`int c;`
	`do {`
	` // check out cnt`
	` c = pull(cnt);`
	` c = c + 1;`
	`} while (!push(cnt, c));`
	`  // commit cnt`

The subscript in a variable's value indicates its version:

t's LOCAL	u's LOCAL	STM
done	success	cnt: 2_5

The counter example: a retry run

⟨name: cnt, version: x, value: y⟩

thread t	thread u

`int c;`
`do {`
` // check out cnt`
` c = pull(cnt);`
` c = c + 1;`
`} while (!push(cnt, c));`
`  // commit cnt`
	`int c;`
	`do {`
	` // check out cnt`
	` c = pull(cnt);`
	` c = c + 1;`
	`} while (!push(cnt, c));`
	`  // commit cnt`

The subscript in a variable's value indicates its version:

t's LOCAL	u's LOCAL	STM
done	done	cnt: 2_5

STM in Erlang

An STM is a server that provides the following main operations:

- `pull(Name)`: check out a copy of variable with name `Name`
- `push(Name, Vars)`: commit all variables in `Vars`; return `fail` if unsuccessful

Clients read and write local copies of variables using:

- `read(Var)`: get value of variable `Var`
- `write(Var, Value)`: set value of variable `Var` to `Value`

We base the STM implementation on the `gserver` generic server implementation we presented in a previous class.

STM: operations

```
create(Tm, Name, Value) ->
  gserver:request(Tm, {create, Name, Value}).
drop(Tm, Name) ->
  gserver:request(Tm, {drop, Name}).
pull(Tm, Name) ->
  gserver:request(Tm, {pull, Name}).
push(Tm, Vars) when is_list(Vars) ->
  gserver:request(Tm, {push, Vars});
read(#var{value = Value}) ->
  Value.
write(Var = #var{}, Value) ->
  Var#var{value = Value}.
```

STM: server handlers

The storage is a dictionary associating variable names to variables; it is the essential part of the server state.

```
stm(Storage, {pull, Name}) ->
  case dict:is_key(Name, Storage) of
    true ->
      {reply, Storage,
       dict:fetch(Name, Storage)};
    false ->
      {reply, Storage, not_found}
   end;
```

```
stm(Storage, {push, Vars}) ->
  case try_push(Vars, Storage) of
    {success, NewStorage} ->
      {reply, NewStorage, success};
    fail ->
      {reply, Storage, fail}
  end.
```

STM: try to push

Helper function `try_push` determines if any variable to be committed has a different version from the corresponding one in the STM.

```
try_push([], Storage) ->
  {success, Storage};
try_push([Var = #var{name = Name, version = Version} | Vars],
         Storage) ->
  case dict:find(Name, Storage) of
    {ok, #var{version = Version}} ->
      try_push(Vars,
               dict:store(Name,
                          Var#var{version = Version + 1},
                          Storage));
    _ -> fail
  end.
```

Using the Erlang STM

Using the STM to create atomic functions is quite straightforward. For example, here are pop and push atomic operations for a list:

```
% pop head element from 'Name'
qpop(Tm, Name) ->
  Queue = pull(Tm, Name),
  [H|T] = read(Queue),
  NewQueue = write(Queue, T),
  case push(Tm, NewQueue) of
    % push failed: retry!
    fail -> qpop(Tm, Name);
    % push successful: return head
    _ -> H
  end.
```

```
% push 'Value' to back of 'Name'
qpush(Tm, Name, Value) ->
  Queue = pull(Tm, Name),
  Vals = read(Queue),
  NewQueue = write(Queue,
                   Vals ++ [Value]),
  case push(Tm, NewQueue) of
    % push failed: retry!
    fail -> qpush(Tm, Name, Value);
    % push successful: return ok
    _ -> ok
  end.
```

Composable transactions?

The simple implementation of STM we have outlined does not support easily composing transactions:

```
% pop from Queue1 and push to Queue2
qtransfer(Tm, Queue1, Queue2) ->
  Value = qpop(Tm, Queue1),   % another process may interleave!
  qpush(Tm, Queue2, Value).
```

To implement composability, we need to keep track of pending transactions and defer commits until all nested transactions have completed.

See the course's website for an example implementation:

```
% atomically execute Function on arguments Args
atomic(Tm, Function, Args) -> todo.
```